

From Discrepancy to Majority

David Eppstein and Daniel S. Hirschberg

Department of Computer Science, University of California, Irvine*

Abstract. We show how to select an item with the majority color from n two-colored items, given access to the items only through an oracle that returns the discrepancy of subsets of k items. We use $n/\lfloor \frac{k}{2} \rfloor + O(k)$ queries, improving a previous method by De Marco and Kranakis that used $n - k + k^2/2$ queries. We also prove a lower bound of $n/(k-1) - O(n^{1/3})$ on the number of queries needed, improving a lower bound of $\lfloor n/k \rfloor$ by De Marco and Kranakis.

1 Introduction

A large body of theoretical computer science research concerns problems of computing a function using a minimal number of calls to an oracle for another function on small subsets of input values. Such problems include sorting with a minimum number of comparisons, as well as *combinatorial group testing*, in which the goal is to identify the positions of a small set of true values among a larger number of false values using an oracle that returns the disjunction of an arbitrary subset of values [5,6]. Other problems with this flavor include Valiant’s work on computing the majority of n values by shallow circuits of 3-input majority gates [10] and recent work by the authors using two-input disjunctions to identify a small number of slackers among a larger number of workers [7].

De Marco and Kranakis [4] provide another interesting class of such problems. Their input consists of n items, each having one of two colors. The goal is to select an item of the majority color or, if the input is equally balanced between colors, to report that fact rather than returning an item. The algorithm may only access the input by *counting* queries on k -item subsets of the input. If a subset has b black items and $w = k - b$ white items, then the result of the query is $c = \min(b, w)$, the size of the smaller of the two color classes. Equivalently, one may ask for the *discrepancy* $d = |b - w|$ of the query subset; the count can be calculated from the discrepancy or vice versa via the identity $2c + d = k$. The motivating application of De Marco and Kranakis is in fault diagnosis of distributed systems, which requires a majority of processors to be non-faulty. Their queries model tests that examine a small number of processors per test in order to determine whether the fault-free processors are indeed a majority.

The case $k = 2$ of this problem had been previously studied [1,2,9], and optimal bounds are known [1,9]. De Marco and Kranakis [4] provide more general solutions that apply whenever k is sufficiently smaller than n . They show that it is possible to find a majority item for even k using only $n - k + k^2/2$ counting queries,¹ and they prove a lower bound of $\lfloor n/k \rfloor$ on the number of queries that are necessary for this problem for all k .

The upper bound of De Marco and Kranakis for counting queries is greater than the lower bound by a factor of k in its leading term. In this work, we reduce this upper bound by a factor of approximately $k/2$ to $n/\lfloor \frac{k}{2} \rfloor + O(k)$, matching the lower bound to within a constant factor independent of k .

De Marco and Kranakis also considered a more powerful type of query, the *output model*, in which the answer to a query is a partition of the queried set into two monochromatic subsets (not

* David Eppstein was supported in part by NSF grant CCF-1228639.

¹ There is a bug in their method for odd k , in Case 1 of Theorem 4.1, when $i = \lfloor k/2 \rfloor$.

revealing the colors of each subset). For this problem De Marco and Kranakis provided an upper bound of $\lceil (n-1)/(k-1) \rceil$ queries, and showed that the same $\lfloor n/k \rfloor$ lower bound for counting queries applies also to the output model. For odd k , we show that their upper bound is tight by proving a matching lower bound. For even k , we slightly improve their upper bound and prove a new lower bound that is within an additive $O(n^{1/3})$ lower-order term of the upper bound. Our new lower bounds apply both to counting queries and to the output model, and the $n/(k-1)$ leading terms of the new lower bounds improve the n/k leading term of the previously known bound.

Our results can also be interpreted in the framework of *discrepancy theory*, the study of how small the discrepancy of the sets in a set system can be made by choosing an appropriate 2-coloring of the set elements [3]. The first stage in our counting-query algorithm, finding an unbalanced query, is equivalent to constructing a system of k -element sets with discrepancy > 1 , and our results for this stage provide examples of such unbalanced k -set systems.

1.1 Notational conventions and problem statement

We use the following shorthand notation for sets:

- $[m]$ denotes the set $\{1, 2, \dots, m\}$ of the first m positive integers.
- If S is a set, i is an element of S , and j is not an element of S , then S_i^j denotes the set $(S \setminus \{i\}) \cup \{j\}$. That is, we replace i by j in S .
- With the same conventions, if A is a subset of S and B is disjoint from S , then S_A^B denotes the set $(S \setminus A) \cup B$.
- If S and T are two sets of numbers with $|S| \geq |T|$, then $S \triangleleft T$ is the set formed from S by removing the $|T \setminus S|$ largest elements of $S \setminus T$ and replacing them by the elements of $T \setminus S$. The result is a set with the same size as S that forms a subset of $S \cup T$ and a superset of T . By abuse of notation, when t is a number, we write $S \triangleleft t$ as a shorthand for $S \triangleleft \{t\}$.

To avoid confusion with the equality predicate, we use the notation $x := y$ to indicate that a variable x of our algorithm should be assigned the new value y .

An instance of the majority problem may be parameterized by two values, n (the number of input items) and k (the size of queries), with $n > k$. We may represent an input to the problem by an n -tuple X of numbers x_i ($i \in [n]$) where each x_i is a member of the set $\{0, 1\}$. The argument to a query made by a majority-finding algorithm may be represented by a set $Q \subset [n]$ with $|Q| = k$. Then we may define the results of the input queries `count` and `partition` as

$$\begin{aligned} \text{count}(Q) &= \min \left\{ \sum_{i \in Q} x_i, \sum_{i \in Q} (1 - x_i) \right\} \\ \text{partition}(Q) &= \{ \{i \mid x_i = 0\}, \{i \mid x_i = 1\} \}. \end{aligned}$$

By extension, we allow these functions to be applied to any set, not necessarily of cardinality k , with the same definitions.

For odd k it will be convenient to partition the set $[n]$ into two complementary subsets, M and L . M is the set of indices i whose associated values x_i equal the majority value of $[k]$. (This may differ from the majority of $[n]$.) Similarly, L is the set of indices i whose associated values x_i equal the minority value in $[k]$.

We say that a query set Q is *homogeneous* if all of its elements have the same value; that is, it is homogeneous when $\text{count}(Q) = 0$ and when $\text{partition}(Q) = \{\emptyset, Q\}$. We say that a query

is *inhomogeneous* if it is not homogeneous. We say that a query set is *balanced* if it is equally partitioned between elements of the two values (or as near to equal as possible when k is odd). That is, Q is balanced when its discrepancy is at most 1 or when $\text{count}(Q) = \lfloor k/2 \rfloor$. We say that Q is *unbalanced* when it is not balanced.

1.2 New results

We prove the following new results.

- A majority element may be found by making $n/\lfloor \frac{k}{2} \rfloor + O(k)$ count queries. The best previous bound, by De Marco and Kranakis [4], was $n - k + k^2/2$.
- When n is odd, a majority element may be found by making $\lceil (n-2)/(k-1) \rceil$ partition queries. This improves for some values of k the best previous upper bound, by De Marco and Kranakis [4], of $\lceil (n-1)/(k-1) \rceil$.
- Determining the majority element requires at least $\lceil (n-1)/(k-1) \rceil$ queries, for odd k , and $n/(k-1) - O(n^{1/3})$ queries, for even k , regardless of whether the queries are of type count or partition. The best previous lower bound for both these query types, by De Marco and Kranakis [4], was $\lfloor n/k \rfloor$.

In addition our methods prove the following discrepancy-theoretic result:

- For even k , there exists a family of at most $2 \log_2 k + 1$ sets, each having k elements, that cannot be 2-colored to make every set balanced. For odd k , there exists a family of at most $k + 3 \log_2 k + 4$ sets with the same property.

2 Upper bounds for counting

For our new upper bounds for counting we use an algorithm with the following four stages:

1. Find an unbalanced query U .
2. Use U to find a homogeneous query H .
3. Use H to determine $\text{count}([n])$.
4. Based on the value of $\text{count}([n])$, find the result of the majority problem.

We describe these four stages in the following four subsections.

2.1 Finding an unbalanced query

Throughout this section, when a subroutine discovers that a set U is unbalanced, we will abort the subroutine and its callers, and pass U on to the next stage of the algorithm. To indicate that this action is not simply returning to the subroutine’s caller, we describe it using the Java-like pseudocode “throw U ”.

For even k , we do not need to find an unbalanced set, as our algorithm for finding a homogeneous set does not require it. However, the solution below serves as a warmup for the odd- k case. It maintains a homogeneous subset H of a balanced set B , repeatedly doubling H until it is too large to be a subset of a balanced set. To double H , we query a set B_H^Q ; if it is balanced, then Q and H have the same composition and the doubled set $H \cup Q$ is homogeneous.

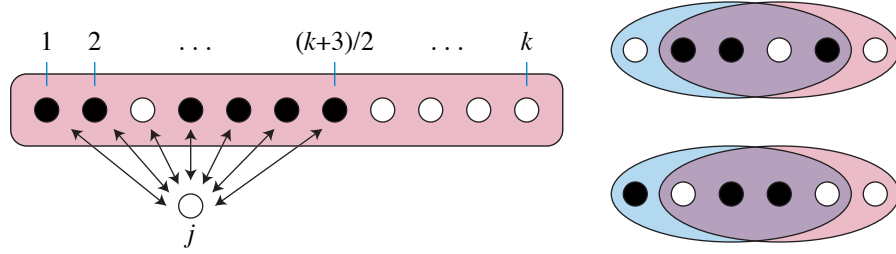


Fig. 1. Left: the arrows connect pairs of elements swapped into and out of the queries made by $\text{star}(j)$. Right: if two overlapping queries (shown as ellipses) differ in a single element, and are both balanced, then either the two swapped elements have equal values (top) or they are unequal but both are in the majority for their query (bottom).

Subroutine 1 to find an unbalanced set when k is even:

1. Set $B := [k]$ and $H := \{1\}$.
2. Repeat the following steps:
 - (a) If B is unbalanced, throw B .
 - (b) Let Q be a set disjoint from B with $|Q| = |H|$.
 - (c) If B_H^Q is unbalanced, throw B_H^Q .
 - (d) Set $H := H \cup Q$ and then set $B := B \triangleleft H$.

Lemma 1. *Subroutine 1 always throws an unbalanced set after at most $2 \log_2 k + 1$ queries.*

Proof. Throughout the loop, H remains homogeneous, and doubles in size at each iteration. The loop terminates on or before the iteration for which $k/2 < |H| \leq k$, after at most $2 \log_2 k + 1$ queries, because substituting such a large homogeneous set into B will always produce an unbalanced set. Thus, $|H|$ cannot grow larger than k and cause B_H^Q to become undefined. \square

For the subroutine to work correctly, we must have $n \geq 3k/2$ to ensure that a large enough subset Q disjoint from B can be chosen in step 2(b).

When k is odd we use a similar idea, doubling the size of a small unbalanced seed set until it overwhelms the whole set, but the details are more complicated. In the first place, the seed set for the doubling routine in the even case is always the set $\{1\}$, found without any queries, but in the odd case we choose our seed more carefully to have the form $\{j, j'\}$ where $\{j, j'\} \subset L$. To construct this seed, we choose j and j' to be arbitrary indexes disjoint from $[k]$ and then verify that they both belong to L by using the following subroutine:

Subroutine 2 $\text{star}(j)$ (for $j > k$) verifies that $j \in L$ or finds an unbalanced set:

1. If $[k]$ is unbalanced, throw $[k]$.
2. For $i := 1, 2, \dots, (k+3)/2$, if $[k]_i^j$ is unbalanced, throw $[k]_i^j$.

The subroutine name refers to the fact that the pairs (i, j) defining the queries form the edges of a star graph (Figure 1, left).

Lemma 2. *If $\text{star}(j)$ terminates without finding an unbalanced set, then $j \in L$.*

Proof. There are two different possible ways that the sets $[k]$ and $[k]_i^j$ queried by the algorithm can both be balanced (Figure 1, right): either $x_i = x_j$ (the upper case in the figure), or $i \in M$ and $j \in L$ (the lower case). The first of these two possibilities, that $x_i = x_j$, can happen only for $\lceil k/2 \rceil$ choices of i , for otherwise too many of the members of $[k]$ would be equal to x_j (and each other) for $[k]$ to be balanced. However, $\text{star}(j)$ tests a larger number of pairs than that. Therefore, if it tests all of these pairs and fails to find an unbalanced set, then it must be the case that $j \in L$. \square

We define a set S with even cardinality to be L -heavy if a majority of S belongs to L , and L -balanced if S is either balanced or L -heavy. Because we assume $|S|$ is even, an L -heavy set must contain at least $1 + |S|/2$ elements of L , and an L -balanced set must contain at least $|S|/2$ elements of L .

Lemma 3. *The disjoint union of an L -heavy and an L -balanced set is L -heavy.*

Proof. If X and Y are disjoint with X containing at least $1 + |X|/2$ elements of L and $|Y|$ containing at least $|Y|/2$ elements of L , then $X \cup Y$ contains at least $1 + |X|/2 + |Y|/2 = 1 + |X \cup Y|/2$ elements of L . \square

Our algorithm for the odd case of stage 1 depends on the following result, which lets us determine an L -heavy set of size double that of a previously known L -heavy set using $O(1)$ queries.

Lemma 4. *Suppose that S and T are sets disjoint from $[k]$, that S is L -heavy, that $|S| = |T| \leq k$, and that $[k]$, $[k] \triangleleft S$, and $[k] \triangleleft T$ are all balanced. Then T is necessarily L -balanced.*

Proof. Let U be the set of the largest $|S|$ elements of $[k]$; this is the subset of $[k]$ removed to make way for S in the set $[k] \triangleleft S$ (Figure 2). For $[k]$ and $[k] \triangleleft S$ to be balanced, U can have at most one more member of M than S does; that is, U is L -balanced. Again, for $[k]$ and $[k] \triangleleft T$ to be balanced, T must have at least as many members of L as U does; therefore, T is also L -balanced. \square

Based on Lemma 4, we define a second subroutine $\text{multiply}(P, m)$ that transforms an L -heavy set P into a larger L -heavy set of cardinality $m|P|$. It takes as input an L -heavy set P , where P has even size and is disjoint from $[k]$, and a positive integer m with $m|P| \leq k$. It either finds an unbalanced set U (aborting the subroutine) or returns as output an L -heavy set of cardinality $m|P|$. We assume as a precondition for this subroutine that $[k]$ has already been determined to be balanced. The subroutine uses the binary representation of m to find its return value in a small number of doublings.

Subroutine 3 $\text{multiply}(P, m)$ (where m and P are as described above) finds an unbalanced set or returns an L -heavy set disjoint from $[k]$ of size $m|P|$:

1. If $m = 1$, return P .
2. If $[k] \triangleleft P$ is unbalanced, throw $[k] \triangleleft P$.
3. Choose Q disjoint from both P and $[k]$ with $|Q| = |P|$.
4. If $[k] \triangleleft Q$ is unbalanced, throw $[k] \triangleleft Q$.
5. Set $R := \text{multiply}(P \cup Q, \lfloor m/2 \rfloor)$.
6. If m is even, return R .
7. Choose S disjoint from R and from $[k]$ with $|S| = |P|$.
8. If $[k] \triangleleft S$ is unbalanced, throw $[k] \triangleleft S$.
9. Return $R \cup S$.

Lemma 5. *If $\text{multiply}(P, m)$ does not throw an unbalanced set, it returns an L -heavy set of cardinality $m|P|$. Regardless of whether it throws an unbalanced set or returns an L -heavy set, it performs at most $3 \log_2 m$ queries.*

Proof. By Lemma 4, if multiply does not find an unbalanced set, then Q and S must both be L -balanced, and their disjoint union with an L -heavy set is another L -heavy set. Therefore, the set returned by this subroutine is L -heavy, and (by induction on the number of recursive calls) has the desired cardinality. The number of levels of recursion (counting only levels that can perform queries) is $\lfloor \log_2 m \rfloor$; at each level it performs either two or three queries, depending on whether m is even or odd. Therefore, in the worst case, it performs at most $3 \log_2 m$ queries. \square

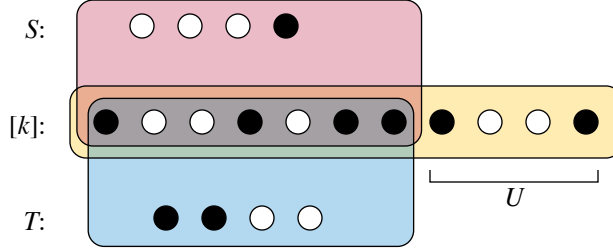


Fig. 2. The sets S (top), T (bottom), and U (middle right), and the query sets $[k]$ (yellow), $[k] \triangleleft S$ (red), and $[k] \triangleleft T$ (blue), used in the proof of Lemma 4.

Putting **star** and **multiply** together, we have the following algorithm to find an unbalanced set when k is odd. It uses **star** twice to find a two-element L -heavy set Y , then uses **multiply** to expand this set to an L -heavy set of $k - 1$ elements. If this L -heavy set together with one element $i \in [k]$ remains unbalanced, it must be the case that $i \in M$. After we identify two members of M , we can replace them with the two known members of L to obtain an unbalanced set.

Subroutine 4 finds an unbalanced set when k is odd:

1. Call **star**($k + 1$) and **star**($k + 2$), and set $Y := \{k + 1, k + 2\}$.
2. Set $Z := \text{multiply}(Y, (k - 1)/2)$, an L -heavy set of $k - 1$ elements.
3. If $Z \cup \{1\}$ or $Z \cup \{2\}$ is unbalanced, throw the unbalanced set.
4. Throw $[k]_{\{1,2\}}^Y$.

Lemma 6. *Subroutine 4 always throws an unbalanced set after at most $k + 3 \log_2 k + 3$ queries.*

Proof. The two calls to **star** (after eliminating the shared query of set $[k]$) take a total of $k + 4$ queries. The call to **multiply** takes at most $3(\log_2 k - 1)$ queries. The remaining steps of the algorithm use at most two queries. Therefore, the total number of queries made in this stage of the algorithm is at most $k + 3 \log_2 k + 3$. \square

In order to work, this algorithm needs n to be at least $2k - 1$ so that it can find enough elements in the disjoint sets that it chooses.

For the algorithms in this stage, the sequence of queries made by the algorithm is non-adaptive: whenever a query finds an unbalanced set, the algorithm terminates, so the sequence of queries can be found by simulating the algorithm using an oracle that knows nothing about the input and always returns a balanced result. Eventually, the algorithm will determine that some particular set is unbalanced without querying it. The sequence of query sets together with the final unqueried and unbalanced set form a family of k -sets with the property that, no matter how their elements are colored, at least one set in the family will be unbalanced. This proves the following result:

Theorem 1. *When k is even, there exists a family of at most $2 \log_2 k + 1$ sets, each having k elements, that cannot be 2-colored to make every set in the family be balanced. When k is odd, there exists a family of at most $k + 3 \log_2 k + 4$ sets with the same property.*

These bounds are not tight for many values of k . When $k \equiv 2 \pmod{4}$, three k -sets with pairwise intersections of size $k/2$ cannot all be balanced. And for many odd values of k our bound can be improved by using optimal addition chains. However, such improvements would make our algorithms more complex and would affect only a low-order term of our overall analysis.

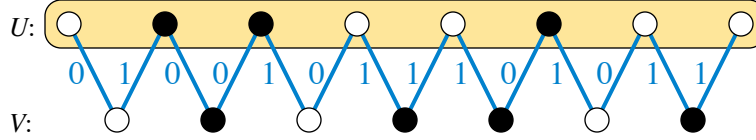


Fig. 3. Finding a homogeneous query. Given an unbalanced k -element query U (top, yellow), we find a disjoint set V of $k - 1$ elements (bottom), and construct a spanning tree of the complete bipartite graph that has U and V as its two vertex sets (blue edges). We then query each set U_i^j for each spanning tree edge ij and use the result to label each edge 0 (if $x_i = x_j$) or 1 (otherwise). Any two elements of $U \cup V$ have the same value if and only if the spanning tree path connecting them has even label sum.

2.2 Finding a homogeneous query

After the previous stage of the algorithm, we have obtained an unbalanced query U . We may also assume that we know the result of the query $\text{count}(U)$, for the algorithm of the previous stage will either query this number itself or it will find an unbalanced query U for which $\text{count}(U)$ can be determined without making a query. Our algorithm for finding a homogeneous query is based on the principle that, for any two indices i and j with $i \in U$ and $j \notin U$, we can test whether $x_i = x_j$ in a single additional query, by testing whether $\text{count}(U_i^j) = \text{count}(U)$. If $x_i = x_j$ then the count stays the same, clearly. However, with U unbalanced, it is not possible for the two indices to have different values while preserving the count.

Subroutine 5 to find a homogeneous query:

1. Let V be a set of $k - 1$ elements disjoint from U .
2. Construct a spanning tree T of the complete bipartite graph $K_{k,k-1}$ having U and V as the two sides of its bipartition.
3. For each edge (i, j) of T , with $i \in U$ and $j \in V$, query $\text{count}(U_i^j)$. Label the edge with the number 1 if the query value is different from $\text{count}(U)$ and instead label the edge with the number 0 if the two query values are equal.
4. Define two elements of $U \cup V$ to be equivalent when the path connecting them in T has even label sum, and partition $U \cup V$ into the two equivalence classes X and Y of the resulting equivalence relation.
5. Return a subset of k elements from the larger of the two equivalence classes.

The algorithm is illustrated in Figure 3. This stage performs $2k - 2$ queries and requires that $n \geq 2k - 1$.

2.3 Finding the count

We next use the known homogeneous query H to compute $\text{count}([n])$.

Subroutine 6 to compute $\text{count}([n])$, given a homogeneous set H :

1. Partition $[n] \setminus H$ into $(n - k) / \lfloor \frac{k}{2} \rfloor$ subsets S_1, S_2, \dots , each having at most $\lfloor \frac{k}{2} \rfloor$ elements.
2. For each subset S_i of the partition, query $\text{count}(H \triangleleft S_i)$. Since $S_i \leq k/2$ and the remaining elements of $H \triangleleft S_i$ are homogeneous, this query determines the number of elements of S_i that are not the same type as H .
3. Let c be the sum of the query values, and return $\min(c, n - c)$.

As well as computing $\text{count}([n])$, the same algorithm can determine whether H is in the majority (according to whether c or $n - c$ is the minimum) and, if not, find an inhomogeneous query I for which $|H \cap I| \geq k/2$ (any of the queries with a nonzero query value). The number of queries it needs is

$$\frac{n - k}{\lfloor k/2 \rfloor} \leq \frac{n}{\lfloor k/2 \rfloor} - 2.$$

2.4 Finding the majority

After the previous three stages of the algorithm, we have the following information:

- A homogeneous query H .
- The number $\text{count}([n])$.
- Whether the elements of H are in the majority.
- An inhomogeneous query I (if H is not in the majority), with $|H \cap I| \geq k/2$.

If $\text{count}([n]) = n/2$, we report that there is no majority. If H is a subset of the majority, we may return any element of H as the majority element. In the remaining case, we find an element of I that is not of the same type as the elements of H , using binary search:

Subroutine 7 uses binary search to find a majority element:

1. Let $U := I \setminus H$, a set containing an element not the same type as H .
2. Let $c := \text{count}(I)$, the number of majority elements in U already determined in stage three of the algorithm.
3. While $|U| > c$, do the following steps:
 - (a) Let $V :=$ any subset of $\lfloor |U|/2 \rfloor$ elements of U .
 - (b) Query $\text{count}(H \triangleleft V)$.
 - (c) If the result of the query is nonzero, let $U := V$ and let $c :=$ the query result. Otherwise, let $U := U \setminus V$ and leave c unchanged.
4. Return any element of the remaining set U .

Lemma 7. *Subroutine 7 finds a majority element using at most $\log_2 k$ additional queries.*

Proof. By induction, for a given set U , this algorithm uses at most $\lfloor 1 + \log_2(|U| - 1) \rfloor$ queries. The worst case occurs when $|U|$ is one plus a power of two and the query result is zero, resulting in a case of the same type in the next step. Since initially $|U| \leq k/2$, it follows that the total number of queries for this stage of the algorithm is less than $\log_2 k$. \square

This bound can be improved by making a more careful choice of the set I to ensure that the initial values in the algorithm satisfy $c > |U|/2$, but this improvement is unnecessary for our results.

2.5 Counting analysis

By adding together the numbers of queries made in the four stages of our algorithm we obtain the following result.

Theorem 2. *Let k and n be given integers with $n \geq 2k - 1$ and $k > 1$. Then it is possible to find a majority element of a set of n 2-colored elements, or to report that there is no majority, using at most $n/\lfloor \frac{k}{2} \rfloor + 3k + 4\log_2 k$ count queries on subsets of k elements.*

In the next section we remove the constraint that $n \geq 2k - 1$ by providing substitute algorithms for the case that $k < n < 2k - 1$, using $O(k)$ queries.

3 Upper bounds for small n

Our previous algorithm for finding the majority using counting queries requires that $n \geq 2k - 1$. Here we show how to relax that assumption by giving separate algorithms for the remaining possible values of n , using only $O(k)$ queries. We first consider the case that $k + 1 < n < 2k - 1$; we will handle the case that $n = k + 1$ in a later subsection. Our algorithm for $k + 1 < n < 2k - 1$ has the following outline:

1. If k is odd then find an unbalanced query U ; otherwise set $U = [k]$.
2. Use U to determine the result of the majority problem.

We detail these stages in the following two subsections.

3.1 Finding an unbalanced query when k is odd

We begin in a manner similar to what we did for the case in which $n \geq 2k - 1$ and k is odd. As we already observed, the subroutine $\text{star}(j)$ used in that case can fail to find an unbalanced set, but if it does then we know that $j \in L$. As in the earlier algorithm, we use the pseudocode “throw U ” to indicate that the unbalanced set U should be passed to the next stage of the algorithm, terminating the subroutine and any of its callers.

Subroutine 8 to find an unbalanced query for odd k and any $n \geq k + 2$:

1. Call $\text{star}(k + 1)$ and $\text{star}(k + 2)$ to verify that $\{k + 1, k + 2\} \subset L$.
2. Let $Q := \{k + 1, k + 2\}$.
3. Choose $\lfloor k/2 \rfloor$ disjoint pairs of elements in $[k]$. For each pair P , if $[k]_P^Q$ is unbalanced, throw $[k]_P^Q$.
4. Let $m :=$ the remaining unpaired element of $[k]$, and let $\{i, j\}$ be any one of the chosen pairs.
5. If $[k]_{\{i, m\}}^Q$ is unbalanced, throw $[k]_{\{i, m\}}^Q$.
6. Throw $[k]_{\{j, m\}}^Q$.

If any one of the sets $[k]_P^Q$ queried by the algorithm is balanced, then P must contain a member of L . If all $\lfloor k/2 \rfloor$ of these sets are balanced, then (since there are only $\lfloor k/2 \rfloor$ members of $L \cap [k]$ that can be included in the pairs) each pair must consist of exactly one member of L and one member of M . The remaining unpaired element m must also belong to M . Therefore, one of the two final pairs $\{i, m\}$ or $\{j, m\}$ must be a subset of M , and replacing it with Q produces an unbalanced tuple.

This procedure works for all n with $k + 1 < n < 2k - 1$ and requires at most $(3k + 3)/2$ queries.

3.2 Partitioning the input using an unbalanced query

Our algorithm for completing the problem of finding a majority, given an unbalanced query U (when k is odd, or an arbitrary query U when k is even) is very similar to Subroutine 5 for finding a homogeneous set in the case that n is large.

Subroutine 9 to find a majority element from an unbalanced query U :

1. Let $V := [k] \setminus U$.
2. Construct a spanning tree T on the complete bipartite graph $(U, V, U \times V)$.
3. For each edge (i, j) of T , with $i \in U$ and $j \in V$, query $\text{count}(U_i^j)$ and label the edge with 0 or 1 as in Subroutine 5.
4. Partition $U \cup V$ into two subsets of vertices such that, within each subset, each pair of vertices is connected by a path in T with an even label sum.
5. Return the difference in sizes of the two subsets of vertices.

After finding a U with a known value of $\text{count}(U)$ (unbalanced in the case that k is odd), this method uses $n - 1 \leq 2k - 3$ additional queries.

3.3 Finding a majority element when $n = k + 1$

When $n = k + 1$ there are only n possible queries to make: for each element, there is one query that omits that element. However, we can find a majority element using even fewer queries. The principle our algorithm uses is that, unless $[n]$ is equally split, a query that omits a minority element will have a strictly smaller count than a query that omits a majority element. So to find a majority element, we need only try enough queries to ensure that a majority element will be one of the omitted ones, and compare the query values.

Subroutine 10 to find a majority element when $n = k + 1$:

1. Let $q := \text{count}([k])$.
2. For $i := 1, 2, \dots, q + 1$, let $q_i = \text{count}([k]_i^n)$.
3. If all $q_i = q$, return that there is no majority.
4. Choose j with $q_j \neq q$.
5. If $q_j > q$ then return j .
6. If $q_j < q$ then return n .

Since $q \leq k/2$ and this algorithm makes $q + 2$ queries, its worst case number of queries is $(k + 4)/2$.

4 Lower bounds

In contrast to our upper bounds for counting queries, our lower bounds are simpler and tighter in the case that k is odd, so we begin with that case first.

4.1 Odd query size

Our lower bound for odd k uses **partition** queries, as they are the most powerful and can simulate **count** queries: if it is impossible to find the majority using a given number of **partition** queries, it is also impossible with the same number of queries of the other types. We prove our lower bound by an adversary argument: we design an algorithm for answering queries that, unless enough queries are made, will be able to force the querying algorithm into making a wrong choice of answer to the majority problem.

At any point during the interaction of the querying algorithm and adversary, we define the *query graph* to be a bipartite graph that has the n given set elements on one side of its bipartition and the queries made so far on the other side of the bipartition. We make each query be adjacent to the elements in it. As a shorthand, we use the word *component* to refer to a connected component of the query graph. The querying algorithm can be assumed to know the results of applying the **partition** and **count** functions to any subset of elements within a single component, as those results can be inferred from the queries actually performed within the component. Note also that, if any component C has discrepancy zero, the querying algorithm may safely ignore that component for the rest of the querying process, as removing its elements from the problem will not change the majority.

To simplify the task of the adversary, we restrict the querying algorithm to make only *reasonable queries*, which we define as queries that never include elements from components with zero discrepancy, and that (unless the result of the query leaves at most one nonzero-discrepancy component) never include more than one element from the same pre-query component. It follows from these properties that the querying algorithm must stop making queries, and choose an output for the majority problem, if it ever reaches a state where at most one component has nonzero discrepancy.

Lemma 8. *Any lower bound for an algorithm that makes only reasonable queries will be valid as a lower bound for all querying algorithms.*

Proof. An arbitrary querying algorithm can be transformed into one that makes only reasonable queries by skipping any query whose elements belong to one component, removing query elements that come from zero-discrepancy components or that duplicate the component of another element, and replacing the removed elements by elements from new components. This modification produces components that are supersets of the original ones, from which the results of the original queries can be inferred. \square

By induction, with only reasonable queries for k odd, if more than one component remains, then all components have odd cardinality and therefore odd discrepancy. We design an adversary that maintains for each odd component a partition of its elements into two subsets (consistent with previous answers) that has discrepancy one. If a query produces a single component of even cardinality, we allow the adversary to choose any partition consistent with previous answers. If a query merges multiple discrepancy-one components, then (by choosing slightly more than half of the input components to have a majority that coincides with the majority of the merged component, and slightly fewer than half of the input components to have a majority that falls into the minority of the merged component) we can always find a consistent partition with discrepancy one. Therefore, by induction, the adversary can always achieve the goals stated above.

Lemma 9. *If a querying algorithm that makes reasonable queries does not reduce the input to a single component before producing its output, then the adversary described above can force it to compute an incorrect answer.*

Proof. Unless there is one component, more than one answer to the majority problem is consistent with the choices already made by the adversary.

In particular, if there are evenly many odd components of discrepancy one, then by choosing the majorities of all components to be the majority of the whole input, it is possible to cause the whole input to have a majority. But by choosing half of the components to have a majority of value 0 and half of the components to have a majority of value 1, it is also possible to cause the whole input to be evenly split between the two values and have no majority. Thus, regardless of whether the querying algorithm declares that there is no majority or whether it chooses a majority element, it can be made to be incorrect.

If there are an odd number of odd components, then a majority always exists. We may achieve discrepancy one for the whole input set of elements by choosing slightly more than half of the components to have majority value 1 and slightly fewer than half to have majority value 0; however, each component can be either on the majority 1 or majority 0 side, so each element can be either in the majority or in the minority. Regardless of which element the querying algorithm determines to belong to the majority, it can be made to be incorrect. \square

Theorem 3. *When k is odd, any algorithm that always correctly finds the majority of n elements by making partition or count queries must use at least $\lceil (n-1)/(k-1) \rceil$ queries.*

Proof. As above, the algorithm can be assumed to make only reasonable partition queries, and must make enough queries to reduce the query graph to a single component. This graph initially has n components, and each query reduces the number of components by at most $k-1$, from which the result follows. \square

4.2 Even query size

De Marco and Kranakis showed that the majority problem on n elements may be solved using $\lceil (n-1)/(k-1) \rceil$ partition queries on subsets of k elements, matched by the lower bound of Theorem 3. For odd n , this bound may be improved to $\lceil (n-2)/(k-1) \rceil$ by applying it only to the first $n-1$ elements, and either returning the result (if it is a majority) or the final element (if the first $n-1$ elements have no majority). However, this modification to their algorithm can reduce the number of queries only when $k-1$ evenly divides $n-2$, which only happens when k is even. Therefore, this improvement does not contradict Theorem 3. When $k=2$ a similar improvement can be continued recursively by pairing up elements, eliminating balanced pairs, and recursively finding the majority of a set of representative elements from each pair. The resulting algorithm uses $n-b$ queries, where b is the number of nonzero bits in the binary representation of n , and a matching lower bound is known [9]. Again, this does not contradict Theorem 3 because $k=2$ is even. These improvements to the upper bound of De Marco and Kranakis raise the question of whether the majority can be found with significantly fewer queries whenever k is even. However, we show in this section that the answer is no. An adversary strategy similar to the odd- k strategy but more complicated than it can be used to prove a lower bound of $n/(k-1) - O(n^{1/3})$ on the number of queries.

For odd k , we proved our lower bound using an adversary that (for reasonable queries) always chooses a partition of each query that gives the resulting component discrepancy exactly one. For even k , we do not wish to create components of discrepancy zero (because that would allow the querying algorithm to eliminate all the elements of the component from future consideration) but, without creating components of discrepancy zero it is not possible to bound the discrepancy that may be needed. In particular, if the querying algorithm makes queries that have the structure of a complete k -ary tree of height h , then the adversary will be forced either to create a component of discrepancy zero or to use discrepancy values as large as 2^h . And it is not always a good strategy for the adversary to choose a nonzero discrepancy for every query, for in the case that $k=2$ and $n=2^h-1$, a complete binary tree strategy against such an adversary can succeed in answering majority queries with only $(n-1)/2$ queries by creating a homogeneous component that is large enough to overwhelm the remaining unqueried elements.

Instead, we will parameterize our adversary by a threshold value τ , and have it follow the following strategy for each query. If it is possible to partition the query elements consistently with previous queries so that the resulting component has discrepancy at most τ , choose the partition that results in as small a nonzero discrepancy as possible. Otherwise, if this is not possible, choose a partition that results in discrepancy zero. The ability to follow this strategy is ensured by the following lemmas. Here, we define the query graph and its components in exactly the same way as in the odd- k lower bound, and (as in that bound) we assume that all queries are reasonable.

Lemma 10. *Suppose that a given reasonable query combines elements from a collection of components $C_i (i = 1, 2, \dots)$ that have nonzero discrepancies d_i . Then it is possible to answer the query consistently with previous queries, to achieve total discrepancy $\sum \sigma_i d_i$ for any choice of $\sigma_i = \pm 1$.*

Proof. Let $C = \bigcup C_i$ be the component resulting from the query. If $\sigma_i = \sigma_1$ choose a partition of C in which the majority elements of C_i are on the same side as the majority elements of C_1 , and if $\sigma_i \neq \sigma_1$ choose a partition of C in which the majority elements of C_i are on the opposite side as the majority elements of C_1 . Then, answer the query by restricting this partition of C to the query elements. \square

Lemma 11. *Suppose that a given reasonable query combines elements from a collection of components C_i that have nonzero discrepancies d_i , and that at least two of the discrepancies d_i are*

different from each other. Then it is possible to answer the query in such a way that the discrepancy of the resulting component is nonzero and at most $\max d_i$.

Proof. Again, let C be the union of the query components; by the previous lemma we may choose any combination of signs for the discrepancies of these components. We prove the lemma using induction on the number of components to be combined. We may assume without loss of generality that $d_1 = \max d_i$. If some two components different from C_1 (without loss of generality C_2 and C_3) have different discrepancies from each other, then we may combine them with opposite signs to each other (that is, we choose $\sigma_2 \neq \sigma_3$), effectively replacing them by a single component with discrepancy $|d_2 - d_3|$ which is nonzero and different from d_1 ; the result follows by induction. If on the other hand all discrepancies other than d_1 are smaller than d_1 and equal to each other, we may cancel them in pairs (again by choosing values of σ_i with opposite signs). If this cancellation leaves no components other than C_1 , then the discrepancy of C equals d_1 . If there is a single component C_2 left after this cancellation, with discrepancy $d_2 < d_1$, then by choosing opposite signs for σ_1 and σ_2 we may cause the discrepancy of C to equal $d_1 - d_2$. \square

Lemma 12. *Suppose that a given reasonable query combines elements from a collection of components C_i that all have the same nonzero discrepancies d . Then if there are an odd number of components in the query, it is possible to answer the query in such a way that the discrepancy of the resulting component is exactly d . If there are an even number of components, it is always possible to achieve discrepancy zero, and it is also always possible to achieve discrepancy $2d$, but it is not possible to achieve any discrepancy between those two values.*

Proof. To achieve discrepancy zero or d , we choose signs σ_i that are as evenly balanced as possible between $+1$ and -1 . To achieve discrepancy $2d$, we start with a balanced set of signs and then flip one of them. \square

Combining these lemmas, we have the following description of the adversary's behavior.

Lemma 13. *For any positive even integer τ , it is possible for an adversary to answer reasonable partition queries in such a way that:*

- Each query answer is consistent with the previous answers.
- At all times, each nonzero discrepancy is at most τ .
- Whenever a query results in a nonzero discrepancy d , then either at least one of the components combined by the query also has discrepancy at least d , or all the pre-query components combined by the query have discrepancy exactly $d/2$.
- Whenever a query results in a zero discrepancy, then all the pre-query components combined by the query have equal discrepancy that is greater than $\tau/2$.

We have the following bound on the sizes of the zero-discrepancy components produced by this adversary:

Lemma 14. *If an adversary follows the strategy described by Lemma 13, and a reasonable query creates a component with discrepancy d that is not the single remaining component of nonzero discrepancy, then the number of elements in the component is at least $k^{\log_2 d} = d^{\log_2 k}$.*

Proof. For each query Q of discrepancy d , we can find k earlier component queries of discrepancy at least $d/2$, either directly as the ones containing the elements of the component or indirectly as the set of k earlier queries of the component combined by Q that has discrepancy at least d . By continuing recursively, we can find a complete k -ary tree of queries, of height $\log_2 d$, whose elements are all combined in query Q . There must be at least one element of Q for each leaf of this tree. \square

Corollary 1. *If an adversary follows the strategy described by Lemma 13, and a reasonable query creates a component with discrepancy zero, then that component must contain $\Omega(\tau^{\log_2 k})$ elements.*

To complete the lower bound argument, we must also bound the sizes of the components remaining whenever a querying algorithm has compiled enough information to correctly determine the majority.

Lemma 15. *If an algorithm for finding the majority makes a sequence of queries that leaves a set of components with the property that each nonzero component discrepancy is at most half of the total component discrepancy, then it is impossible for the algorithm to correctly choose a majority element.*

Proof. If one component C_1 has exactly half of the remaining discrepancy, then by setting all of the remaining component's signs σ_i equal to each other, it is possible either to make no majority (when those signs are different from σ_1) or a majority (when those signs are equal to σ_1). Thus, regardless of whether the algorithm determines that there is or is not a majority, it may be made incorrect.

If, on the other hand, all components have discrepancy that is less than half of the total, then any element i can be made to be a majority element. If i belongs to any component (with zero or nonzero discrepancy), we choose equal signs for all other nonzero-discrepancy components in order to force a majority to exist, and then choose a sign for the component containing element i as desired. Thus, regardless of whether the algorithm chooses that no majority exists or chooses a particular element as a representative of the majority, it can again always be made incorrect. \square

Corollary 2. *If an adversary follows the strategy described by Lemma 13, and a querying algorithm returns its answer after making a sequence of queries that leads to components with total discrepancy at least 2τ , then the adversary can choose a partition of the elements consistent with its previous answers that makes the querying algorithm incorrect.*

For large values of τ , Corollary 1 implies that the components with zero discrepancy are also large, and therefore that there cannot be many such components. For small values of τ , Corollary 2 implies that (when a querying algorithm is capable of determining a correct answer) the components with nonzero discrepancy have a small sum of discrepancies, and therefore that there cannot be many such components. By choosing a value of τ that balances the numbers of components of both types, we can force any correct querying algorithm to leave only a small number of components of either type. To do this, it must make a large number of queries.

Theorem 4. *When k is even, any algorithm that always correctly finds the majority of n elements by making partition or count queries must use at least $n/(k-1) - O(n^{1/(1+\log_2 k)})$ queries.*

Proof. We set $\tau = n^{1/(1+\log_2 k)}$, and follow the adversary strategy described above. Then by Corollary 1 each component with zero discrepancy left at the end of the querying algorithm must have at least $\Omega(\tau^{\log_2 k})$ vertices, from which it follows that there are at most $n/\Omega(\tau^{\log_2 k}) = O(n^{1/(1+\log_2 k)})$ such components. By Corollary 2 the total discrepancy of the nonzero components must be less than 2τ ; since each such component has discrepancy at least one, the number of such components is also less than $2\tau = O(n^{1/(1+\log_2 k)})$.

The query graph initially has n components, and ends with $O(n^{1/(1+\log_2 k)})$ components. Each query reduces the number of components by at most $k-1$, from which the result follows. \square

In particular, for $k \geq 4$ we need at least $n/(k-1) - O(n^{1/3})$ queries. Combining this bound with the known $n - \log_2 n$ lower bound for the $k=2$ case [9] shows that a lower bound of $n/(k-1) - O(n^{1/3})$ is valid for all k .

5 Conclusions

We have provided new bounds for the majority problem, for count and partition queries. For partition queries with odd query size, our bounds are tight, and for even query size we achieve a matching leading term in our upper and lower bounds. However, for count queries, our upper and lower bounds are separated from each other by a factor of two. Reducing this gap remains open.

Recently, Gerbner et al. have given bounds for the majority problem for a different type of query that returns an element of the majority of a three-tuple [8]. It would be of interest to extend their results to k -tuples as well.

Our work also raises the discrepancy-theoretic question of how many sets are needed in a family of k -element sets that cannot be balanced. In this, also, our bounds are not tight and further improvement would be of interest.

References

1. L. Alonso, E. M. Reingold, and R. Schott. Determining the majority. *Inform. Process. Lett.* 47(5):253–255, 1993, doi:10.1016/0020-0190(93)90135-V, MR1245142.
2. L. Alonso, E. M. Reingold, and R. Schott. The average-case complexity of determining the majority. *SIAM J. Comput.* 26(1):1–14, 1997, doi:10.1137/S0097539794275914, MR1431242.
3. J. Beck and W. W. L. Chen. *Irregularities of distribution*. Cambridge Tracts in Mathematics 89. Cambridge University Press, Cambridge, 2008, MR2488272.
4. G. De Marco and E. Kranakis. Searching for majority with k -tuple queries. *Discrete Math. Algorithms Appl.* 7(2):1550009, 2015, doi:10.1142/S1793830915500093, MR3349882.
5. D.-Z. Du and F. K. Hwang. *Combinatorial Group Testing and its Applications*. Ser. Appl. Math. 12. World Scientific, 2nd edition, 2000, MR1742957.
6. D. Eppstein, M. T. Goodrich, and D. S. Hirschberg. Improved combinatorial group testing algorithms for real-world problem sizes. *SIAM J. Comput.* 36(5):1360–1375, 2007, doi:10.1137/050631847, MR2284085.
7. D. Eppstein, M. T. Goodrich, and D. S. Hirschberg. Combinatorial pair testing: distinguishing workers from slackers. *Proc. 13th Int. Symp. Algorithms and Data Structures (WADS 2013)*, pp. 316–327. Springer, Lecture Notes in Comput. Sci. 8037, 2013, doi:10.1007/978-3-642-40104-6_28, MR3126368.
8. D. Gerbner, B. Keszegh, D. Pálvölgyi, B. Patkós, M. Vizer, and G. Wiener. Finding a majority ball with majority answers. *Proc. 8th Eur. Conf. Combinatorics, Graph Theory, and Applications (EuroComb 2015)*, pp. 345–351. Elsevier, *Elect. Notes Discrete Math.* 49, 2015, doi:10.1016/j.endm.2015.06.047.
9. M. E. Saks and M. Werman. On computing majority by comparisons. *Combinatorica* 11(4):383–387, 1991, doi:10.1007/BF01275672, MR1137770.
10. L. G. Valiant. Short monotone formulae for the majority function. *J. Algorithms* 5(3):363–366, 1984, doi:10.1016/0196-6774(84)90016-6, MR756162.