DETERMINING MEMBERSHIP WITH 2 SIMULTANEOUS QUERIES

DAVID HOWARD

ABSTRACT. *Alice* and *Bob* are playing a cooperative game in which *Alice* must devise a scheme to store n elements in an array from a universe U of size m. Her goal is to store in such a way that for every $x \in U$ *Bob* can observe the values of two positions (dependent on x) in the array and determine whether x is in the array or not. *Alice* may share her storage scheme with *Bob* and they win if such an arrangement is made. The question is how large can the universe U be in terms of n so that *Alice* and *Bob* can win? In this paper we give upper and lower bounds on this question for general n and the special case when $n = 3$. We also pose conjectures and further questions for research.

1. INTRODUCTION

Sorting and searching are fundamental operations that we use everyday in many different ways. Our assumptions change from problem to problem and we are frequently concerned with having these operations done as quickly as possible. We are sometimes concerned with how long an operation will take us to complete in the worst case, other times knowing the average amount of time to complete such an operation is important. There is a large amount of research in this area and for an encyclopedic but by no means comprehensive discussion in the area we refer the reader to [4].

Let us look at a particular but normal instance of sorting and searching. Suppose *Alice* has a list of n objects taken from a Universe U and we must choose a good storing scheme so that if *Bob* wants to determine whether a particular element x is among our n objects he doesn't need to search the entire string to decide whether x is in the list or not. Well immediately questions come to mind:

- Is *Bob* aware of the storage scheme?
- What do we know about the universe?
  - Do we know what all the possible items are in the universe?
  - Are the items comparable with each other in some way (i.e. can they be ordered)?
  - Is the universe finite and if so how does it compare to n?

Well the answers to both knowing the storage scheme and comparability must be yes otherwise *Bob* may have to traverse the whole list on some searches. Given these two assumptions perhaps the most natural thing to do is to look at the storage scheme that completely sorts the n element input. Now when *Bob* wants to determine if a particular element is in the list he can perform a binary search of the list which runs in a logarithmic number of steps. Knowing nothing else about U this strategy has the best possible average run time. That being said it is often the case that we have additional assumptions on U. For example, perhaps we have knowledge of the elements in U and that $|U|$ is finite. Now choosing to sort the elements in order and having *Bob* perform binary search may not be the best method of quickly finding membership among the list.

Let us suppose we have knowledge of the elements of the universe and that the universe is finite. Knowing the elements of the set means that we can assign numerical values to each element and impose an order. Thus, we can assume our universe is the set of integers from 1 to some integer m. Without any knowledge of the value of m compared to n, sorting in order and using a binary search to determine membership in our list may seem like a reasonable strategy. What if, however, $m = n + 1$? In this case in-order sorting and performing binary search is a rather poor strategy. If *Alice*, as a storage scheme, places the integer above (modular n) the missing element in the first position of the list, then *Bob* by looking at the value stored in the first position can determine exactly who is in the list. *Bob* in this case needs to query a position in the list just once to complete his goal. This naturally begs the question: How big can m be so that only one query is needed to determine membership of any particular value from the list?

This question has been solved [16] and the answer is $m = 2n - 2$ for $n > 2$ (Note: for $n = 2$ the answer is 3). To show that $2n - 2$ is possible we list the storage scheme from [16].

Let the universe be the set $[2n - 2]$ and our storage list be denoted as rooms with labels $1, 2, \ldots, n$. We say that each element $p \in [2n - 2]$ prefers to stay in the room ($p \mod n$) (In the case of 0 choose room n instead). We also denote the first n elements as lower elements and the last $n-2$ elements as upper elements. We say room r is *full* if all members who prefer room r from the universe are among the n elements to be stored and *empty* if no members prefer room r. Note there are always at least two full rooms and rooms $n - 1$ and n are either full or empty whereas the other rooms may be neither. Storage is as follows:

- Any element that prefers a room that is not full is stored in that room.
- If an upper element prefers a room that is full it is stored in a room that is empty.
- If a lower element prefers a room that is full it is stored in a different room that is full.

To determine membership for an element $p \in [2n - 2]$ *Bob* queries room $p' = (p \mod n)$ (In the case of 0 choose room n instead). If p' contains p or another lower element that does not prefer p' as a room then p is in one of the rooms, otherwise it's not.

We leave it to the reader to verify that this scheme will work. That being said perhaps the next logical question is: How big can m be if you allow two queries of the list to determine membership? Well, this question isn't well-defined because it isn't clear whether *Bob* can use the information obtained by one query to help decide where in the list *Bob* makes the second query. In an upcoming paper Paul Horn, Adam Jobson, Andre Kezdy, and Jake Wildstrom [5] give an adaptive membsership algorithm (i.e. *Bob* can use the information from the first query) , which is similar to Yao's algorithm, where $m = 3n - 4$. At the time of writing there has been *no* upper bound found for m regarding an adaptive strategy. The main results of this paper give upper and lower bounds for the non-adaptive question (*Bob* apriori must decide which two positions in the list he will query and then based on these answers determine whether a particular element from the universe is in the list).

We make one other point of note from Yao's article that given special additional arrangements two queries is enough to determine membership under the adaptive scheme. In fact, in this special case when Bob determines that someone (Person x) *is* among the stored rooms then the member seen by the second query will actually be Person x. Specifically, given an arbitrarily large universe and an additional room which is occupied with whomever Alice chooses, Bob can determine membership with only two queries. Bob determines who Alice has put in the additional room and this person acts as an index (Alice has many different storage schemes and the additional person tells Bob which scheme to choose from). Bob can then query a specific room which will either contain Person x or Bob will claim that Person x is not among the stored list.

2. Related Work and Previous Results

Despite the simplicity of the question "Is x in one of the rooms?" from a computer science standpoint the question can quickly become quite difficult to even ask:

- How is the information stored (e.g. A linked list of integers? A zero-one array which has the size of my universe? etc.)
- How many rooms do I have?
- What type of answer do I get from my queries (numerical/boolean)?
- How many such queries am I entitled to? What is my universe size?
- Are my queries adaptive?

In this article, we assume that queries give a numerical answer (which for a computer means the size of the answer is logarithmic in the size of the finite universe). This model is known as the cell-probe model. In general there are a variety of parameters that one can adjust for this question about membership. A very popular approach to this problem is for the queries to give only true/false answers, known as the bit-probe model.

Under this boolean query assumption, a variety of results have approached the problem from a different angle [1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. Namely, given a fixed set of rooms n and universe $[m]$ with t queries, how large (in terms of bits) does a data structure need to be to for any element from $[m]$ to be able to determine membership using at most t bit-probes? Furthermore, are these probes adaptive or non-adaptive. In [3, 14] the authors even add the complexity of bit-probes having a probability of returning a false answer. This question regarding size of the data structure is well-defined as it should be clear that m is always an upper bound in terms of storage (One can store a 0-1 vector where a 1 in the ith position indicates membership for value i. Thus, only one query of this structure is needed).

Much of the previous work on membership is on adaptive queries, which one can argue seems a more natural question. That being said non-adaptive schemes have been studied in other works [1, 3, 12, 13]. Non-adaptive questions in some sense are simpler to study because analyzing the possible decisions for adaptability are difficult to do (we again note no upper bound is yet known in the adataptive version of this problem [5]). Results of this nature also provide lower bounds for their adaptive counterparts.

We direct the reader to [6, 7] for a good survey of research pertaining to the question of membership.

3. Bounds for the non-adaptive two query problem

Let m, n be integers. Let $\binom{[m]}{n}$ denote the n element sets of $[m]$ and for every set $Q \in \binom{[m]}{n}$ denote $Q[i]$ as the i-th smallest element in Q. Now let F be the family of functions that take $\binom{[m]}{n}$ to a permutation on n elements. Thus, for $f \in F$, we say $[f(Q)](i)$ represents where the permutation $f(Q)$ maps the i-th element. We shall call the functions from F *storage schemes*. For each $f \in F$ and $Q \in \binom{[m]}{n}$ we can define a function $g_{f,Q}$ which is an injective map from $[n]$ to Q such that $g_{f,Q}([f(Q)](i)) = Q[i]$ (or $g_{f,Q}(j) = Q[[f(Q)]^{-1}(j)]$). In layman's terms f is *Alice*'s storage scheme, Q is the set to be stored, and $g_{f,Q}(i)$ represents the element stored in position i among the n positions. Finally, define G_f as the set of all functions $g_{f,Q}$ over all $Q \in \binom{[m]}{n}$. Thus, G_f represents all the different storage mappings corresponding to the different n sets of $[m]$.

Example 3.1. Let $n = 3$ and $m = 5$. There are $\binom{5}{3} = 10$ sets of size 3 and 6 possible permutations. Namely:

Sets	
Q_1	$\{1, 2, 3\}$
Q_2	$\{1, 2, 4\}$
Q_3	$\{1, 2, 5\}$
Q_4	$\{1, 3, 4\}$
Q_5	$\{1, 3, 5\}$
Q_6	$\{1, 4, 5\}$
Q_7	$\{2, 3, 4\}$
Q_8	$\{2, 3, 5\}$
Q_9	$\{2, 4, 5\}$
Q_{10}	$\{3, 4, 5\}$

Permutations	$\sigma(1), \sigma(2), \sigma(3)$
σ_1	$1, 2, 3$
σ_2	$1, 3, 2$
σ_3	$2, 1, 3$
σ_4	$2, 3, 1$
σ_5	$3, 1, 2$
σ_6	$3, 2, 1$

G_f	$g(1), g(2), g(3)$
g_{f,Q_1}	$1, 2, 3$
g_{f,Q_2}	$1, 4, 2$
g_{f,Q_3}	$2, 1, 5$
g_{f,Q_4}	$4, 1, 3$
g_{f,Q_5}	$3, 5, 1$
g_{f,Q_6}	$5, 4, 1$
g_{f,Q_7}	$2, 3, 4$
g_{f,Q_8}	$2, 5, 3$
g_{f,Q_9}	$4, 2, 5$
$g_{f,Q_{10}}$	$5, 3, 4$

We define $f(Q_i) = \sigma_{i \, mod \, 6}$ (In the case of 0 use 6 instead) to get the third table. So, $[f(Q_{10})](2) = 3$ and $g_{f,Q_{10}}(3) = Q_{10}[2] = 4$ for example.

So for given m and n values *Alice* and *Bob* must choose a storage scheme f for *Alice* to use. Now for each element $j \in [m]$ *Bob* must determine two *queries* $q_j^1, q_j^2 \in [n]$ such that for any $g_1 = g_{f,Q_1}, g_2 = g_{f,Q_2} \in G_f$ if $g_1(q_j^1) = g_2(q_j^1)$ and $g_1(q_j^2) = g_2(q_j^2)$ then either ($j \in Q_1$ and $j \in Q_2$) or ($j \notin Q_1$ and $j \notin Q_2$). If such a set of query assignments exists we shall call G_f *consistent*. Plainly, G_f is consistent if element j has query positions q_j^1, q_j^2 and the values of $g(q_j^1)$ and $g(q_j^2)$ determine whether j is a member of the input set or not for all j.

The above example is indeed consistent because by looking at the first two elements stored in the list one can immediately determine what is the 3rd element in the list. Thus, for each number let the 2 queries be the first two elements of the list. There are, however, bad choices for the queries that would not insure being consistent. For example, if *Bob* decided the membership queries for element 2 to be the 2nd and 3rd positions in the storage list then he could not tell the difference between Q_7 and Q_{10} being stored. In Q_7 2 is in the list and in Q_{10} 5 is in the storage list, thus determining membership for element 2 can not be done under this choice of queries. That being said element 1 could happily query the second and third positions in the storage list. Even though seeing the elements 3 and 4 (respectively) in the last two positions does not determine the element in position 1, in all such cases position 1 does not contain element 1. Thus, membership for element 1 can be determined.

Theorem 3.2. *For $n \geq 30$, if $|U| = m \geq 6\binom{n}{2}\log_2 n$ then there is no $f \in F$ such that G_f is consistent.*

Proof. It is enough to prove for $m = \lceil 6\binom{n}{2}\log_2 n \rceil$ because we can always restrict the input domain for larger m. Suppose not and that there does exist an f such that G_f is consistent. Therefore there exists a choice of queries for each of the m elements that shows G_f is consistent. By the pigeonhole principle there exists two positions that are queried for at least $\lceil 6\log_2 n \rceil$ elements. Without loss of generality we shall assume these positions are 1 and 2 and let S be a set of size $\lceil 6\log_2 n \rceil$ elements that queries positions 1 and 2. This implies that by looking at the first two positions of the list one can determine membership for the subset of elements in S among any given input set from $\binom{[m]}{n}$.

At most there exists $m(m-1)$ different ordered pairs of values seen in positions 1 and 2. Meanwhile there are $2^{|S|}$ possible subsets of S and further there are members of $\binom{[m]}{n}$ that witness each possible subset ($\lceil 6\log_2 n \rceil \leq n$). Thus $m(m-1)$ needs to be larger than $2^{|S|}$. However,

$$m(m-1) < \frac{36}{4}n^4(\log_2 n)^2 < n^6 \leq 2^{\lceil 6\log_2 n \rceil} = 2^{|S|}$$

□

We note here that Theorem 3.2 can be improved in the following way:
For $\epsilon > 0$ there exists an n_ϵ such that if $n > n_\epsilon$ and $m \geq (4+\epsilon)\binom{n}{2}\log_2 n$ then there is no $f \in F$ such that G_f is consistent. This is true since for large enough n:

$$m(m-1) < \frac{(4+\epsilon)^2}{4}n^4(\log_2 n)^2 < n^{4+\epsilon} \leq 2^{\lceil (4+\epsilon)\log_2 n \rceil} = 2^{|S|}$$

It is the author's opinion that this upper bound is far from the actual truth for where the threshold of consistent G_f's arise.

Conjecture 3.3. *There exists a constant k such that for $n \geq 3$ and $m \geq kn$ then there is no $f \in F$ such that G_f is consistent.*

The conjecture stems from the attractiveness of the lower bound and the complicated simultaneous intersection of queries that would need to be dealt with if the threshold were $\omega(n)$. For example in the proof of Theorem 3.2 we said positions 1 and 2 must determine the membership of the set S however meanwhile we may argue there is another set S' that queries positions 2 and 3 and yet another set S'' that queries positions 1 and 3. Each of these queries must simultaneously transmit the information of membership of S,S' andS'' without in some sense getting in each other's way. The lower bound for the threshold of consistency is linear in n because this construction does not have any cross-over among queries. Additionally, the lowerbound threshold for adaptive queries (The second query can be decided as a result of the information received from the first query) is also linear in n and the threshold for this setting is an upperbound on the non-adaptive setting.

Theorem 3.4. *For $n \geq 4$, if $|U| = m \leq \lfloor \frac{5}{2}n \rfloor - 3$ there exists an f such that G_f is consistent.*

Proof. For this proof we give an explicit construction of such an f given $m = \lfloor \frac{5}{2}n \rfloor - 3$. To illustrate this construction we will first assume that n is even and we will split the queries into $\frac{n}{2}$ pairings of queries. Thus, elements 1 through 5 will query positions 1 and 2, elements 6 through 10 will query positions 3 and 4, etc. Note that only elements $m-1$ and m query the final two positions.

We will illustrate the storage function for the first five elements in Table 1, and claim an analogous storage function for the other elements with the exception of $m-1$ and m which will be dealt with differently.

We repeat this storage function in 5 integer increments analogously for each pair of positions to nearly completely define our storage scheme with the exception of the last 2 positions.

This storage function is almost perfect in that it is almost entirely self-reliant since almost all that matters to the scheme is what subset of the 5 numbers is to be stored. The only thing that is occasionally needed are outside elements that *MUST* be available. So why does this argument fails for $m = \frac{5}{2}n$ and n even? Well suppose we want to store the set $\{1,3\} \bigcup \{8,10,13,15,18,20,23,25,\ldots,m-2,m\}$. The storage function

Table 1. Storage assignment for all but final positions (addition is modulo 5)

Subset to be stored	**Ordered pair stored**
All numbers 1 through 5	$\{1,3\}$
Set of size 4 with $i-1$ removed	$\{i+1,i\}$
$\{i,i+1,i+2\}$	$\{i,i+1\}$
$\{i-1,i,i+2\}$	$\{i+2,i\}$
$\{i,i+2\}$ (not $\{1,3\}$)	$\{i,i+2\}$
$\{i,i+1\}$	$\{i,X\}$ (where X is not an element 1 through 5)
$\{1,3\}$	$\{X,1\}$ (where X is not an element 1 through 5)
$\{i\}$ (not $\{1\}$)	$\{X,i\}$ (where X is not an element 1 through 5)
$\{1\}$	$\{X_2,X_1\}$ (where $X_1 < X_2$ and X_1,X_2 are not elements from 1 through 5)
$\emptyset$	$\{X_1,X_2\}$ (where $X_1 < X_2$ and X_1,X_2 are not elements from 1 through 5)

Table 2. Storage Assignment for final pair in even case

Subset to be stored	**Ordered pair stored**
$\{m-1,m\}$	{m-1,m} OR {m,X}
$\{m-1\}$	{X,m-1} (where X is not m or $m-1$)
$\{m\}$	{X,m} (where X is not m or $m-1$)
$\emptyset$	{X,Y} (where X,Y are not m or $m-1$)

wants to look like $[X,1,8,10,13,15,18,20,23,25,\ldots,m-2,m]$ but 3 doesn't get stored anywhere and X cannot be taken from anywhere.

A similar problem occurs with Yao's original problem in that $2n$ is almost possible except for the case when there is exactly one full room and one empty room. The full room does not have access to a lower element from another room. Thus, just like for Yao's original problem we need a clean-up crew (namely the final two positions) which is given in Table 2.

We claim that we can now put all these storage functions together in such a way that we can build a storage scheme. The first observation is the only problem we may face is needing a value from the outside positions when none are available and thus can't use the storage function in its original form. We also notice that the amount of excess values (overflow from a paired position) is equal to the amount of shortage (shortage of values for a paired position). Sometimes a paired position has both excess and shortage (e.g. if 1 is the only element from the first 5 to be stored 1 does not appear in the first pair of positions but requires two outside values). Ultimately any pairing of shortage values with excess values will work except matching a paired position's shortage with its own excess. We note this can only happen corresponding with a paired position behaving like lines 6, 7, or 9 from Table 1. We give explicit instructions for how the extra storage values will be stored (a cyclical assignment). Starting at the values that query positions 1 and 2 among values, if there are an excess of values give these values to the first available positions that need it in increasing order of positions (i.e. if all numbers 1 through 5 are to be stored somewhere in the array positions 1 and 2 are given the values 1 and 3, the excess values are 2, 4, and 5) . Now repeat this process for the values that query positions 3 and 4 and give any excess values to the first available positions that need outside values in increasing order (cycling around back to positions 1 and 2 if necessary). Continue this process until the $n-1$ and n position cycling the excess around if necessary. The only problem that will exist is if the excess values cycle all the way back around to where they came from. Explicitly, this will necessarily mean that there is only one pair of positions that has a shortage of values and this pair will also have excess values as well. For example something like $\{1,3\}\bigcup\{8,10,13,15,18,20,23,25,\ldots,m-4,m-2,m-1,m\}$ or $\{1,8,9,10\}\bigcup\{13,15,18,20,23,25,\ldots,m-4,m-2,m-1,m\}$). We note in both cases positions 1 and 2 produce both a shortage and an excess and there is no other pair that produces a shortage and an excess. Additionally, in both cases only one additional outside value is needed. In such cases both $m-1$ and m must be in the set and in these special cases $m-1$ can be taken from the final pair of positions and used as an excess for the position pair with a shortage and the final two stored values will be $\{m,X\}$ where X is

Table 3. Storage Assignment for final three positions when n is odd

Subset to be stored	**Ordered triple to be stored (X, Y, Z are elements outside the last four)**
$\emptyset$	$\{X, Y, Z\}$
$\{m-1\}$	$\{X, Y, m-1\}$
$\{m\}$	$\{X, Y, m\}$
$\{m-1, m\}$	$\{X, m-1, m\}$
$\{m-3\}$	$\{m-3, X, Y\}$
$\{m-2\}$	$\{m-2, X, Y\}$
$\{m-3, m-1\}$	$\{m-3, X, m-1\}$)
$\{m-3, m\}$	$\{m-3, X, m\}$
$\{m-2, m-1\}$	$\{m-2, X, m-1\}$
$\{m-2, m\}$	$\{m-2, X, m\}$
$\{m-3, m-1, m\}$	$\{m-3, m-1, m\}$ or $\{m-3, m, X\}$
$\{m-2, m-1, m\}$	$\{m-2, m-1, m\}$ or $\{m-2, m, X\}$
$\{m-3, m-2\}$	$\{m-3, m-2, X\}$
$\{m-3, m-2, m-1\}$	$\{m-3, m-2, m-1\}$ or $\{X, m-2, m-1\}$
$\{m-3, m-2, m\}$	$\{m-3, m-2, m\}$ or $\{X, m-2, m\}$
$\{m-3, m-2, m-1, m\}$	$\{m-2, m-3, m\}$

the excess value from the paired position with the shortage. So for the first example we would have $X = 3$ and the second example we would have $X = 1$.

The final piece to clean up is when n is odd. In this case the final three positions are special. For the final three positions we look at the final four elements to be stored namely $\{m-3, m-2, m-1, m\}$.

If at most one of $m-3$ and $m-2$ are to be stored then reduce to the even case of n by either storing the one variable in the third to last position or it needs a value from somewhere else but doesn't provide any excess values in which case no excess values will cycle back to their original position where a value is needed.

If both $m-3$ and $m-2$ are to be stored and exactly one of m and $m-1$ are to be stored the storage function is either:
$\{m-3, m-2, m \text{ (or } m-1)\}$ or $\{X, m-2, m \text{ (or } m-1)\}$ where X is a value not among the last four elements. The first storage function is used in all cases except if an excess value is needed so as to avoid an excess value cycling around to its original pair of queried positions.

If both $m-3$ and $m-2$ are to be stored and neither m or $m-1$ appears then store the last 3 positions as $\{m-3, m-2, X\}$ where X is an outside value.

Finally, if all 4 are present store it as $\{m-2, m-3, m\}$.

The storage scheme is summarized in Table 3.

Elements $m-3$ and $m-2$ query the $n-2$ and $n-1$ positions meanwhile $m-1$ and m still query the final two positions. It is clear that in all cases but one $m-3$ and $m-2$ will be able to witness themselves to test membership (the one case being $X, m-2$ in which case it is still clear). As for $m-1$ and m by seeing an $m-3$ in the second to last position implies that both $m-1$ and m are in the input set whereas $m-2$ in the second to last position means only the final position determines membership of $m-1$ and m (at most one being in the set). In all other cases the membership of m and $m-1$ behave like the even case for n. We take note of the four cases that can be stored in two different ways. How these sets are stored again depends if there is exactly one position that has a shortage and this pair also produces an excess. This case is handled identically as it is in the even case. If an additional excess value is needed then the second storage option is chosen and the argument for consistency is the same as in the even case. Thus, we find this scheme to be consistent. $\square$

We note here that it is unlikely that a better storage scheme exists that does not include overlapping of queried positions among the elements (i.e. element X queries positions one and two meanwhile element Y queries positions two and three for example). The scheme is tight in the sense that *Bob* when looking at positions one and two cannot distinguish between outside elements other than which is larger than the other. If *Bob* could in some way use the outside elements more to his advantage in which ones are stored in the first two positions it may be possible to improve this bound but given that the outside elements are in some sense arbitrary for large values of n it may be impossible to use their actual value for information. The $\frac{5}{2}$ fraction is rather tight since by looking at the two elements in the first position *Bob* must determine membership for all values 1 through 5 among the n stored values from U. Well there are $2^5 = 32$ such subsets of 1 through 5. Meanwhile there are $5 * 4$ ways of putting ordered pairs of values from one through 5. Then there are $5 + 5$ ways of storing an outside element with an element of 1 through 5. Finally there are two ways to store outside elements using only comparison between them and that is $\{X_1, X_2\}$ or $\{X_2, X_1\}$. Thus we see that $32 = 5 * 4 + (5 + 5) + 2$ exactly. Letting 6 numbers look at the queried positions 1 and 2 will give $2^6 = 64$ possibilities to worry about meanwhile in some sense $6 * 5 + (6 + 6) + 2$ possible storage pairs which is less than 64.

4. Special Case $n = 3$

In this section we look at the very particular case of $n = 3$ and how large U can be so that with 2 simultaneous queries (dependent on each element) one can determine membership from an input set. This is different from 2 queries in which one could adapt based on the answer to the first queried position. In such an adaptive search $|U|$ is unbounded since a simple sort of the input set by *Alice* and a binary search by *Bob* of the sorted set can verify membership. In the simultaneous setting this comparison of the elements is not allowed to determine the second query and as a result the maximum size of $|U|$ is somewhere between 15 and 18 inclusive.

Theorem 4.1. *For $n = 3$, if $|U| = m > 18$ then there is no storage scheme f such that G_f is consistent. If $|U| = m \leq 15$ there exists a storage scheme f such that G_f is consistent.*

As a warm-up before the proof we will give a quick argument why 25 is too large to have a storage scheme f such that G_f is consistent. If $m \geq 25$ then there must exist 9 elements who each query the same two positions among the 3 possible positions. Well looking at the inputs from these nine elements there are $\binom{9}{3}$ possible input sets of size 3. Meanwhile when restricted to these 9 elements the maximum number of possible different ordered pairs of queries *Bob* can see is $9 * 8 = 72 < \binom{9}{3} = 84$. However, *Bob* from looking at these two positions must be able to tell which of these 9 elements are in the set. There is not enough possibilities to have a consistent storage scheme and hence $m \leq 24$. Below we give a tighter but similar argument to show 19 is not possible.

Proof. We will prove first the upperbound and then give a construction for the lower bound. Let us suppose $m \geq 19$. This means that at least seven elements must query the same two positions among the three by the pigeonhole principle. Let us take seven such elements and call this set S. We first note that if f is a storage scheme such that G_f is consistent, then *Bob* by looking at these two queries positions (without loss of generality we will assume positions one and two) must be able to determine the subset of S that are among the three selected elements from U. From S there are $\binom{7}{3} = 35$ input sets of size 3 meanwhile there are $7 * 6 = 42$ options available for an ordered pair of queried information which means at most seven of these are used to encode a doubleton input pair among the $\binom{7}{2} = 21$ sets from S. Thus, there are at least 14 sets of size 2 from S that are stored with one of the two elements not in the first two positions. Let T be the set of these doubleton sets and let X_1 and X_2 be two distinct elements not from S. There are at least 28 possible input sets that consist of a pair from T and one of the variables X_1 and X_2. Meanwhile there are exactly 28 possibilities of remaining representations for the storage scheme to represent these input sets of T along with one of X_1 or X_2. This implies that all sets of size 1 from S must be represented with the one element being in the third position. This is a problem since when *Bob* sees $\{X_1, X_2\}$ or $\{X_2, X_1\}$ he must be able to determine the singleton element from S that is stored which is impossible since there are 7 such possibilities. Thus, G_f is not consistent.

In the second half of the proof we establish the lower bound by constructing a storage scheme f such that G_f is consistent. Let U be the values 1 through 15. *Bob* to determine membership of the values 1 through 5 (Group A) will query the first two stored positions, for 6 through 10 (Group B) the second and third positions and finally for values 11 through 15 (Group C) *Bob* will query the first and third positions. *Alice* will store the input set keeping the following two ideas in mind. 1.) If only one member of A, B or C shows up in an input set it will always be placed in one of the two queried positions for each group. 2.) If 2 or 3 members of a particular group (say A) are in an input set then both queried positions for that group (in this case 1 and 2) will contain members of that group (A). So as examples the input set $2, 6, 15$ will be stored either as $\{2, 6, 15\}$ or $\{15, 2, 6\}$ meanwhile $\{2, 3, 15\}$ will be stored either as $\{2, 3, 15\}$ or $\{3, 2, 15\}$. This implies that *Bob* if he queries a value in group A and at most one of the two queried positions is in A then the third position (the unseen one) is not a member of A and thus what he sees from his queries determines what is stored from A. We take note here that when *Alice* must store one value from each group storing $\{"A", "B", "C"\}$ or $\{"C", "A", "B"\}$ is ok for the storage scheme. From here we describe the storage scheme for when two or three members of A are to be placed in the three positions and claim that the other two groups B and C store analogously.

First note that there are $\binom{5}{3}$ sets of size 3 from A and $\binom{5}{2}$ sets of size 2 from A. Meanwhile there are $5 * 4$ possible combinations of ordered pairs from posistions one and two that can be seen by *Bob*. Luckily, $\binom{5}{3} + \binom{5}{2} = 5 * 4$ and thus showing a matching from each set to each observed pair is sufficient. One can make a Hall's matching type argument but in this case it's quicker just to show a possible matching of sets to pairs (below the arithmetic is modulo 5):

Subset to be stored	**Ordered pair stored in first two positions**
$\{i, i+1, i+2\}$	$\{i, i+2\}$
$\{i, i+1, i+3\}$	$\{i, i+1\}$
$\{i, i+1\}$	$\{i+1, i\}$
$\{i, i+2\}$	$\{i+2, i\}$

Thus, if we define f analogously for the cases where B and C have two or three members among the input set then we see that G_f is consistent. □

This storage scheme in some sense feels tight and so we conjecture the following.

Conjecture 4.2. *For $n = 3$, if $|U| = m > 15$ then there is no storage scheme f such that G_f is consistent.*

5. Open Questions and Conclusion

As mentioned in the introduction there has been work [5] on the adaptive version of this problem (where the position of the second query can be decided after viewing the result of the first query). In this preliminary manuscript they devise a storage scheme for a set of size n from a universe of size at most $3n - 4$ and an adaptive 2-query scheme to determine membership. Between [5] and this article there are a variety of questions that can be asked. In both cases (adaptive and non-adaptive) there is still work to be done to find the maximum size of the universe given n and as stated in Conjecture 3.3 the author conjectures that for at least the non-adaptive case the maximum size of the universe is $O(n)$.

One final question:

Question 5.1. *For a fixed number of queries k (adaptable or not), how large (asymptotically) can the universe be in terms of n and k? The arguments in section 3 with $k = 2$ can easily be extended to give an upperbound of $O(\binom{n}{k} \log_2 n)$.*

References

[1] N. Alon and U. Feige, On the power of two, three, and four probes. Proc. of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pg 346-354. SIAM, 2009.

[2] A. Brodnik and J. Munro. Membership in Constant Time and Almost-Minimum Space. SIAM Journal on Computing, 28(5):1627-1640, 1999.

[3] H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? SIAM Journal on Computing, 31(6):1723-1744, 2002.

[4] D. E. Knuth, The Art of Computer Programming. Volume 3: Sorting and Searching. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Co., Reading, Mass.-London-Don Mills, Ont., 1973.

[5] Andre Kezdy, University of Louisville, *Personal Communication.*

[6] P. K. Nicholson, V. Raman, and S. S. Rao. Data Structures in the Bitprobe Model. Space Efficient Data Structures, Streams and Algorithms, volume 8066 of LNCS, pages 303-318. Springer, 2013.

[7] Space Efficient Data Structures in the Word-RAM and Bitprobe Models. PhD thesis, University of Waterloo, 2013.

[8] R. Pagh. Low Redundancy in Static Dictionaries with Constant Query Time. SIAM Journal on Computing, 31(2):353-363, 2001.

[9] R. Pagh. On the cell probe complexity of membership and perfect hashing. Proc. of the 33rd Annual ACM Symposium on Theory of Computing (STOC), pages 425-432. ACM, 2001.
[10] M. Pătraşcu. Succincter. Proc. of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pages 305-313. IEEE Computer Society, 2008.
[11] J. Radhakrishnan, V. Raman, and S. S. Rao. Explicit deterministic constructions for membership in the bitprobe model. Proc. of the 9th Annual European Symposium on Algorithms (ESA), volume 2161 of LNCS, pages 290-299. Springer, 2001.
[12] J. Radhakrishnan, S. Shah, and S. Shannigrahi. Data Structures for Storing Small Sets in the Bitprobe Model. European Symposium on Algorithms (ESA) (2), volume 6347 of LNCS, pages 159-170. Springer, 2010.
[13] S. Srinivasa Rao. Succinct Data Structures. PhD thesis, Institute of Mathematical Sciences, Madras University, 2001.
[14] Ta-Shma, A.: Storing information with extractors. Information Processing Letters 83(5), 267-274 (2002)
[15] E. Viola. Bit-probe lower bounds for succinct data structures. SIAM Journal on Computing, 41(6):1593-1604, 2012
[16] Yao, A. C. C., Should Tables be Sorted, J. Assoc. Comput. Mach. 28 (1981), no. 3, 615-628.

Department of Mathematics, Colgate University, Hamilton, NY, 13346
E-mail address: `David Howard: dmhoward@colgate.edu`