Abstract

A sequence avoids a pattern $\pi \in \mathbb{R}^k$ if there is no subsequence that shares the same relative ordering as $\pi$. We develop techniques to understand how many queries it takes to test whether a sequence is $\pi$-avoiding or $\epsilon$-far from $\pi$-avoiding. We initiate the study of searching in $\pi$-avoiding or nearly $\pi$-avoiding sequences, and show that it is possible to efficiently find most elements in a nearly $(2, 1)$-avoiding, i.e. sorted, sequence. Pattern avoidance is a specific instance of hereditary properties, which are closed under taking subsequences. We prove the existence of general properties that require linear queries to test, implying the existence of hereditary properties that require linear queries to test.

1 Introduction

In property testing, we want to be able to determine whether an object has a specific property or is “far” from having the property. The goal is to understand how many queries an algorithm needs to distinguish these two cases with constant probability assuming unbounded time and space.

The objects we consider in this thesis are sequences, typically over the integers or some other ordered alphabet. We define how “far” two sequences are from each other by edit distance, i.e. the number of insertions, deletions, and modifications to get from one sequence to another. A query consists of asking for the value at a particular index in the sequence. The query complexity for a given property is the number of queries needed to distinguish whether any sequence has a property or requires at least $\epsilon n$ edits to attain the property. The query complexity is often dependent on the distance parameter $\epsilon$ and the length of the sequence $n$.

Monotonicity is one natural property for integer sequences. For property testing, this translates to testing whether a sequence is sorted or requires $\epsilon n$ edits to become sorted. It has been shown that testing monotonicity requires $\Theta(\frac{1}{\epsilon} \log n)$ queries [EKK+98, Fis04].

Monotonicity can be generalized to pattern avoidance, which is the main focus of this thesis. A pattern $\pi = (\pi_1, \ldots, \pi_k)$ is a $k$-tuple of real numbers, and we say a sequence contains $\pi$ if there exists a subsequence that shares the same relative ordering as $\pi$. We say a sequence is $\pi$-avoiding if it does not contain $\pi$. For example, monotonicity is equivalent to $(2, 1)$-avoidance, i.e. a sequence is monotone if and only if there is pair of indices such that the first index is greater than the second. We look specifically at $(3, 2, 1)$-avoiding sequences in this thesis, which turns out to be equivalent to being composed of at most 2 monotone increasing subsequences.

Pattern avoidance is a hereditary property of sequences. A hereditary property is one that is closed under the subsequence relation. In other words, if a sequence has a specific hereditary property, then all of its subsequences do as well. For pattern avoidance, if one sequence avoids a
pattern $\pi$, deleting indices will not introduce the pattern into the sequence, so pattern avoidance is indeed a hereditary property.

One question in property testing is if there is a difference between adaptivity and non-adaptivity, i.e. does it matter if queries depend on the responses from previous queries? Newman et al. [NRRS17] show that testing avoidance of monotone patterns of the form $(k, k-1, \ldots, 1)$ can be done with $\text{poly}(\frac{1}{\epsilon}, \log n)$ queries non-adaptively, but testing avoidance of non-monotone patterns, e.g. $(1, 3, 2)$, requires at least $\Omega(n^{1/2})$ non-adaptive queries. However, [NRRS17] also show that $(1, 3, 2)$-avoidance can be tested adaptively using $\text{poly}(\frac{1}{\epsilon}, \log n)$, showing an exponential gap between adaptivity and non-adaptivity. Newman et al. [NRRS17] pose the open problem, can any constant-size permutation pattern be tested adaptively with $\text{poly}(\frac{1}{\epsilon}, \log n)$ queries? In this thesis, we try to make progress on this question.

Standard binary search gives an efficient way to find an element in a sequence using $O(\log n)$ queries given that the sequence is sorted. However using property testing, we can only distinguish between a sequence that is sorted or $\epsilon$-far from being sorted. This raises the question, can you efficiently search in a sequence that is close to being sorted? To the best of our knowledge, this question has not been addressed before, but other variants on randomized binary search have been proposed. Newman et al. [NRRS17] proposed a method to find a value within a range in a nearly sorted sequence. They used this as a subprocedure in adaptively testing $(1, 3, 2)$-avoidance with $\text{poly}(\frac{1}{\epsilon}, \log n)$ queries. Another line of work (see [RMK+80, Pel89] and references therein) deals with a version of binary search where with some fixed probability $p \in (0, 1/2)$ responses are incorrect.

1.1 Our Results

We generalize the intersection search problem defined in [NRRS17], which we call Generalized Intersection Search($\pi, \epsilon$). In Generalized Intersection Search($\pi, \epsilon$), you are given two sequences $A$ and $B$ that are both $\pi$-avoiding and share $\epsilon n$ elements. The goal is to output a pair $(i, j)$ such that $A_i = B_j$. We reduce the Generalized Intersection Search($\pi, \epsilon$) problem to testing general $\tau$-avoidance. This gives a possible lower bound technique for adaptively testing constant-size permutation patterns.

**Theorem 1.** For any pattern $\pi$ of length $k$, if Generalized Intersection Search($\pi, \epsilon$) requires $q$ queries to solve with at least $2/3$ probability, then there exists some pattern $\tau$ of length at most $4k + 6$ that has no one-sided $\epsilon$-tester using fewer than $q$ queries.

We look at Generalized Intersection Search($\pi, \epsilon$) in the first non-trivial case when $\pi = (3, 2, 1)$. We discuss some insights for this problem and propose a possible algorithm showing that Generalized Intersection Search may be easy for $\pi = (3, 2, 1)$. Motivated by trying to solve Generalized Intersection Search($(3, 2, 1), \epsilon$), we develop a technique for searching in an almost sorted sequence. We give a randomized version of binary search where with some fixed probability $p \in (0, 1/2)$ responses are incorrect.

**Theorem 2.** If a sequence $S$ is $\alpha$-close to being sorted, then the nearly sorted search procedure from Algorithm 2 uses $O(\log n \log \log n)$ queries and finds at least a $1 - O(\alpha)$ fraction of the elements in $S$ with at least $2/3$ probability.

Finally, we give a probabilistic argument for why there exist properties of sequences that require $\Omega(n)$ queries. Together with a reduction (in a concurrent work) from testing arbitrary properties to
testing hereditary properties, this implies that hereditary properties of sequences cannot be tested with sublinear query complexity. This solves another open question by Newman et al. [NRRS17] negatively.

**Theorem 3.** Fix $\epsilon = 1/100$. There exists a property such that for any two-sided $\epsilon$-tester requires at least $n/6000$ queries.

1.2 Overview of Techniques

In this thesis, we have three main technical contributions: a reduction from Generalized Intersection Search to testing $\pi$-avoidance; a randomized binary search algorithm for searching in a nearly sorted sequence; and a lower bound for arbitrary properties. We overview the main techniques for each contribution.

**Generalized Intersection Search Reduction.** We give a reduction showing that if Generalized Intersection Search$(\pi, \epsilon)$ requires $q(n, \epsilon)$ queries, then there is some pattern $\tau$ such that testing $\tau$-avoidance requires $q(n, \epsilon)$ queries. We first define an alternating pattern, which has the property that if you replace each element in a $\pi$-avoiding sequence by disjoint blocks of $\pi$, then the only $\pi$ patterns occur in the disjoint blocks. Then to solve Generalized Intersection Search$(\pi, \epsilon)$ for sequences $A$ and $B$ where $\pi$ is alternating, we can replace all elements in $A$ and $B$ by disjoint $\pi$ blocks, shift all elements in $B$ up by $1/2$, and then concatenate the two sequences. Then a one-sided tester for the pattern $\tau = (\pi_1, \ldots, \pi_k, \pi_1 + \frac{1}{2}, \ldots, \pi_k + \frac{1}{2})$ for the concatenated sequence will find two blocks that correspond to matching elements in $A$ and $B$. Thus, if it is hard to solve Generalized Intersection Search for an alternating pattern $\pi$ with at least $2/3$ probability, the corresponding $\tau$ is hard to test. Furthermore, every pattern is contained in some alternating pattern, so it suffices to show Generalized Intersection Search is hard for any pattern $\pi$ to prove that there is some pattern $\tau$ that is hard to test.

**Nearly Sorted Search.** In Algorithm 2, we present a randomized binary search procedure for efficiently searching in a nearly sorted sequence $S$. In each recursive call of the algorithm, we randomly sample $O(\log \log n)$ elements from the middle block of 10% of the elements. If the majority of sample elements are greater than what we are searching for, we recursively search to the left, including the middle 10% block. Otherwise we search to the right. We show that if the middle 10% block mostly consists of elements from the longest increasing subsequence of $S$, then we will make progress with good probability to finding most elements. In fact if $S$ is $\alpha$-close to being sorted, at most $O(\alpha)$ elements won’t be found as long as we always choose the right direction to continue searching. Thus in a sequence $\alpha$-close to being sorted, with constant probability, we find a $1 - O(\alpha)$ fraction of elements using $O(\log n \log \log n)$ queries.

**Existence of Hard Properties.** We then show that there exist properties that require $\Omega(n)$ queries to test by the probabilistic method. We draw a random property $P$ that contains $2^{n/20}$ binary sequences of length $n$, and bound the probability that there exists a tester for $P$. We use Yao’s principle to restrict our attention to deterministic testers with error measured relative to a fixed input distribution for each $P$. We bound the number of adaptive deterministic testers making $q$ queries by equating them to binary trees of depth $q$. Then we show the probability any
particular tester is good over a randomly chosen $P$ is much smaller than the number of testers, when $q \leq n/6000$. Thus, most such properties do not have any tester using at most $n/6000$ queries.

2 Notation

A sequence of length $n$ over an alphabet $\Sigma$ is a function $S : [n] \rightarrow \Sigma$, often written as $(S_1, \ldots, S_n)$. A property $P$ is a set of sequences, and we say a particular sequence $S$ has property $P$ if $S$ is in $P$. A property $P$ is hereditary if for all sequences $S$ in $P$, every subsequence of $S$ is also in $P$. We say that a sequence $S$ is $\epsilon$-far from $P$ if for all $x \in P$, $d(S, x) > \epsilon n$ for some distance measure $d$. In this paper we consider edit distance, i.e. $d(x, y)$ is the minimum number of symbol deletions, insertions, or substitutions needed to transform $x$ into $y$.

An $\epsilon$-tester for a property $P$ is a randomized algorithm that on an input sequence $S$ queries a set of indices of $S$ (possibly adaptively) and accepts with probability at least $2/3$ if $S \in P$ and rejects with probability at least $2/3$ if $S$ is $\epsilon$-far from $P$. Such a tester is said to have two-sided error. If the tester is instead required to accept with probability 1 on all inputs in $P$, we say that the tester has one-sided error.

We say that a property $P$ is testable with $q(n, \epsilon)$ queries if for every $\epsilon > 0$ there is an $\epsilon$-tester for $P$ using at most $q(n, \epsilon)$ queries on sequences of length $n$ with two-sided error.

3 Generalized Intersection Search

We provide a potential technique for showing that for some $\pi$ of constant length $k$, testing a sequence for $\pi$-avoidance may require more than poly($\frac{1}{\epsilon}, \log n$) queries. We define the Generalized Intersection Search problem, generalizing the lower bound technique in [NRRS17].

Definition 4 (Generalized Intersection Search($\pi, \epsilon$)). Fix a pattern $\pi$. Let $A$ and $B$ be sequences of length $n$ that are $\pi$-avoiding and share $\epsilon n$ elements. Assume also that the elements of $A$ are all distinct, and the elements of $B$ are all distinct. Find $(i, j)$ such that $A_i = B_j$.

Note that the intersection search problem from [NRRS17] is identical to the above for $\pi = (2, 1)$ and $\epsilon = 1/3$. We will show that there is a reduction from Generalized Intersection Search to the problem of testing $\pi$-avoidance.

Theorem 1. For any pattern $\pi$ of length $k$, if Generalized Intersection Search($\pi, \epsilon$) requires $q$ queries to solve with at least $2/3$ probability, then there exists some pattern $\tau$ of length at most $4k + 6$ that has no one-sided $\epsilon$-tester using fewer than $q$ queries.

The reduction requires the following definition.

Definition 5. Let $\pi = (\pi_1, \ldots, \pi_k)$ be a pattern of length $k \geq 4$. We say that $\pi$ is alternating if the following conditions hold:

- $\pi_1 < \pi_k < \pi_2$
For all $i \notin \{1, 2, k\}$, $\pi_i < \pi_1$ if $i$ even, and $\pi_i > \pi_2$ if $i$ is odd.

The following is the key property satisfied by alternating patterns.

**Lemma 6.** Suppose that $\pi$ is an alternating integer permutation pattern. Let $S$ be a $\pi$-avoiding sequence of integers consisting of distinct elements. Form the sequence $\hat{S}$ by replacing each entry $S_i$ of $S$ with the sequence $U_i = (2kS_i + \pi_1, \ldots, 2kS_i + \pi_k)$. Then the only copies of $\pi$ occurring in $\hat{S}$ are the $U_i$’s.

*Proof.* Suppose that $(\hat{S}_{i_1}, \ldots, \hat{S}_{i_k})$ form a pattern $\pi$ in $\hat{S}$. If $i_1$ and $i_2$ both occur inside some block $U_j$, then it follows that $i_k$ also occurs inside $U_j$ since $\hat{S}_{i_1} < \hat{S}_{i_k} < \hat{S}_{i_2}$ and all blocks are disjoint. However, $U_j$ has length $k$, so it must be the case that $U_j = (\hat{S}_{i_1}, \ldots, \hat{S}_{i_k})$.

Now consider the case where $i_1$ and $i_2$ lie in different blocks. We claim that for all $i$, $i_\ell$ and $i_{\ell+1}$ lie in different blocks. When $\ell \leq k - 1$, this follows from the fact that $\pi$ is alternating since one of $\hat{S}_{i_\ell}$ and $\hat{S}_{i_{\ell+1}}$ must be greater than $\hat{S}_{i_2}$ and the other must be smaller than $\hat{S}_{i_1}$. For $i = k - 1$, we note that $\hat{S}_{i_{k-1}} \notin (\hat{S}_{i_1}, \hat{S}_{i_2})$, while $\hat{S}_{i_k} \in (\hat{S}_{i_1}, \hat{S}_{i_2})$. Our construction then forces $i_{k-1}$ and $i_k$ to lie in different blocks. Now since each block consists of consecutive elements, we have that for all $1 \leq \ell \leq k$, $\hat{S}_{i_\ell}$ are distinct. However, this implies that the the original sequence $S$ contains $\pi$, which is a contradiction.

Therefore, the only copies of $\pi$ in $\hat{S}$ are in the $U_j$ blocks from the transformation of $S$.  

**Lemma 7.** Fix an alternating integer permutation pattern $\pi$. Let $\tau$ be the pattern $(\pi_1, \ldots, \pi_k, \pi_1 + \frac{1}{2}, \ldots, \pi_k + \frac{1}{2})$. If there exists a one-sided $(\epsilon/2)$-tester for $\tau$-avoidance making $q$ queries, then we can solve Generalized Intersection Search($\pi, \epsilon$) with constant probability by making at most $q$ queries.

*Proof.* Suppose we are given two $\pi$-avoiding lists $A$ and $B$ of length $n$. We will construct a sequence $X$ such that if $A$ and $B$ share at least $en$ elements in common, then $X$ is $\epsilon/2$-far from avoiding $\tau$ and if $A$ and $B$ share no elements, $X$ is $\tau$-avoiding.

We first construct length $kn$ sequences $\hat{A}, \hat{B}$ as in lemma 6 where each element is replaced with a block containing $\pi$ and all blocks are disjoint. Then we set $X$ to be the concatenation of $\hat{A}$ and $\hat{B} + \frac{1}{2}$, where we add $\frac{1}{2}$ to every element in $\hat{B}$. Note that querying into $X$ may be done by making a single query into either $A$ or $B$.

It is clear from the construction that each pair of elements shared between $A$ and $B$ gives rise to a $\tau$-shaped subsequence. We argue that all $\tau$ patterns in $X$ arise from common elements of $A$ and
Consider an arbitrary $\tau$-shaped subsequence $X_\tau$ in $X$. Either the first half of $X_\tau$ occurs in the first half of $X$, or the last half of the $X_\tau$ occurs in the last half of $X$.

In the first case, the first half of $X_\tau$ must coincide with a single block $U_i$ from $A$ by lemma 6. Then the $k + 1$st element of $X_\tau$ must be between the first two elements of $X_\tau$. This means the second half of $X_\tau$ must start in $\hat{B}$ since all blocks in $A$ are disjoint. However, this implies that the second half of $X_\tau$ is a block in $\hat{B}$. These two blocks from the first and second halves intersect. Therefore, they correspond to positions $i, j$ with $A_i = B_j$. A similar argument applies when the last half of $X_\tau$ occurs in the last half of $X$.

Therefore if $A$ and $B$ share $\epsilon n$ elements, then $X$ has $\epsilon n$ disjoint copies of $\tau$ in a length $2n$ sequence, so $X$ is $\epsilon/2$-far from avoiding $\tau$. Also, if $A$ and $B$ share no elements, then there are no $\tau$ patterns in $X$.

It follows that an $(\epsilon/2)$-tester for $\tau$-avoidance will reject with constant probability if $A$ and $B$ share $\epsilon n$ elements. Now recall that if a one-sided $\tau$-avoidance tester rejects, then there must be a $\tau$-shaped subsequence among the entries that it queried, so we can figure out the corresponding indices $(i, j)$ such that $A_i = B_j$. Thus the $\tau$-avoidance tester solves Generalized Intersection Search($\pi, \epsilon$) with constant probability.

We now complete the proof of Theorem 1.

\textit{Proof.} From any pattern $\pi$ of length $k$, we can create an alternating pattern $\pi'$ of length $2k + 3$ that contains $\pi$ as a subpattern. The idea is to start with a $(1, 2)$ pair at the beginning. Then put arbitrary values less than the first index in the odd locations, and the different values of $\pi$ shifted to be greater than the second index in the even locations. Then the $2k + 3$rd element will be in between the first two indices. The $\tau$ pattern given in lemma 7 is twice the length of $\pi'$, which is $4k + 6$. Also, we can set $\tau_i = \ell + 1$, where $\ell$ is the number of elements less that $\tau_i$ in $\tau$, to make $\tau$ an alternating integer permutation pattern.

Thus, if Generalized Intersection Search($\pi, \epsilon$) requires $q$ queries, then Generalized Intersection Search($\pi', \epsilon$) also requires at least $q$ queries since $\pi$ is contained in $\pi'$. But this implies that a one-sided $\epsilon$-tester for $\tau$-avoidance requires at least $q$ queries by Lemma 7.

\hfill $\Box$

4 Searching in $\pi$-Avoiding Sequences

In this section, we generalize searching in a sorted sequence in two different ways. First, we look at searching in a $(3, 2, 1)$-avoiding sequence. Secondly, we consider searching in a nearly $(2, 1)$-avoiding sequence.

Searching in a $(3, 2, 1)$-avoiding sequence is closely related to Generalized Intersection Search for $\pi = (3, 2, 1)$. We propose a possible algorithm and show how this could be used to solve Generalized Intersection Search($(3, 2, 1), \epsilon$) using poly($\frac{1}{\epsilon}, \log n$) queries. Note that the goal of Generalized Intersection Search was to prove lower bounds rather than upper bounds. However, we believe that the observations we make can give insights to the problem in general, and it may give evidence to why testing the avoidance of constant length permutation patterns can be done with poly($\frac{1}{\epsilon}, \log n$) queries.

As part of our proposed algorithm for searching in a $(3, 2, 1)$-avoiding sequence, we need to search in a nearly $(2, 1)$-avoiding sequence. We believe that this problem is interesting in its own right when trying to search on sorted data that may have been corrupted in some way. We prove
that if it most $\alpha(n)$ elements have been corrupted, then with constant probability we can find all but $O(\alpha)$ elements with a randomized binary search procedure.

### 4.1 Searching in $(3, 2, 1)$-Avoiding Sequences

Let $S$ be a $(3, 2, 1)$-avoiding sequence of length $n$ and $x$ the element you are searching for. We make use of the following simple observation. Suppose $(S_i, S_j)$ is a $(2, 1)$ pair in $S$, then one of the following three cases must hold:

- if $S_i > S_j > x$, then $x$ occurs to the left of $S_j$; (Case 1)
- if $x > S_i > S_j$, then $x$ occurs to the right of $S_i$; or (Case 2)
- if $S_i > x > S_j$, then $x$ does not occur between $S_i$ and $S_j$. (Case 3)

Similar to the standard binary search algorithm, we can use this observation to restrict the search space for $x$ every time a $(2, 1)$ pair is found in $S$. In order to make sure we restrict the search space by a constant factor for each $(2, 1)$ pair, we cut our sequence $S$ into five equal blocks. We search for $(2, 1)$ pairs in the second and fourth blocks. If any $(2, 1)$ pair is in Case 1, we can at least rule out that $x$ is in the first block. If any $(2, 1)$ pair is in Case 2, we can at least rule out that $x$ is in the fifth block. Lastly, if there are $(2, 1)$ pairs in Case 3 in both the second and fourth blocks, then we can at least rule out that $x$ is in the third block. This covers all cases in which we find a $(2, 1)$ pair in either the second or fourth blocks.

If we do not make progress in restricting the search space by finding a $(2, 1)$ pair, then we know either the second or fourth block is far from being sorted. This means that we can use a search algorithm for a nearly sorted sequence to check if $x$ is in that block. We propose Algorithm 2 as a way to do this in the next section.

In order to search for a $(2, 1)$ pair, we can use any one-sided error monotonicity tester since it can only reject after seeing a violating pair. In particular, we can use a $(\epsilon / \log n)$-tester to distinguish between a sequence being sorted and $(\epsilon / \log n)$-far from being sorted with constant probability using $O(\epsilon \log^2 n)$ queries. If we repeat this tester $O(\log \log n)$ times and don’t see a $(2, 1)$ pair, then we are confident with probability $1 - 1/O(\log n)$ that the sequence is $\epsilon$-close to being sorted.

We summarize the above idea in Algorithm 1.

**Lemma 8.** Let $S$ be a $(3, 2, 1)$-avoiding sequence. The 321Search procedure from Algorithm 1 uses $O(\frac{1}{2} \log^3 n \log \log n)$ queries.

**Proof.** We reduce the size of the search space by at least $1/5$ in each recursive call, so the algorithm will terminate after $O(\log n)$ iterations. In each iteration, the $(2, 1)$ tester uses $O(\frac{1}{2} \log^2 n)$ queries, and we repeat it $O(\log \log n)$ times to improve the failure probability. Also, we may use $O(\log n \log \log n)$ queries to perform the randomized binary search by Theorem 2. Therefore in each iteration, we make at most $O(\frac{1}{2} \log^2 n \log \log n)$ queries, so the total query complexity is $O(\frac{1}{2} \log^3 n \log \log n)$. \qed

**Correctness Idea.** We discuss the possible correctness of the 321Search procedure of Algorithm 1 and what issues we currently face in the analysis.

Because we assume the original $S$ is $(3, 2, 1)$-avoiding, we never rule out $x$ in a recursive call after finding valid $(2, 1)$ pairs. The only time we may miss $x$ if it exists is during the randomized
Algorithm 1 Proposed Algorithm for (3,2,1)-Avoiding Search

Precondition: $S$ is a (3,2,1)-avoiding sequence of length $n$, $x$ is the element to search for

1: function 321Search($S, x$):
2: Split $S$ into 5 equal sized blocks, $S = S_1 S_2 S_3 S_4 S_5$
3: Search for a (2,1) pair using an $\left(\epsilon / \log n\right)$-tester $O(\log \log n)$ times in $S^2$
4: Set (2,1) pair to $(S_i^2, S_j^2)$ if found
5: if $x$ is found then
6: return corresponding index for $x$
7: if $S_i^2 > S_j^2 > x$ then
8: return 321Search($S_1 S_2^2, x$)
9: if $x > S_i^2 > S_j^2$ then
10: return 321Search($S_2 S_3^2 S_4 S_5, x$)
11: if no $(S_i^2, S_j^2)$ found then
12: NearlySortedSearch($S_2^2, x$)
13: if $x$ is found then
14: return corresponding index for $x$
15: else
16: return 321Search($S_1 S_3^2 S_4 S_5^2, x$)
17: Search for a (2,1) pair using an $\left(\epsilon / \log n\right)$-tester $O(\log \log n)$ times in $S^4$
18: Set (2,1) pair to $(S_i^4, S_j^4)$ if found
19: if $x$ is found then
20: return corresponding index for $x$
21: if $S_i^4 > S_j^4 > x$ then
22: return 321Search($S_1 S_2^4 S_3^2 S_4, x$)
23: if $x > S_i^4 > S_j^4$ then
24: return 321Search($S_1 S_2^2, x$)
25: if no $(S_i^4, S_j^4)$ found then
26: NearlySortedSearch($S_4^4, x$)
27: if $x$ is found then
28: return corresponding index for $x$
29: else
30: return 321Search($S_1 S_2^2 S_3^4 S_5^2, x$)
31: # It must be the case that $S_i^2 > x > S_j^2$ and $S_i^4 > x > S_j^4$
32: return 321Search($S_1 S_2^4 S_4 S_5^2, x$)
Algorithm 2 Randomized Binary Search Algorithm for Nearly Sorted Sequences

**Precondition:** $S$ is a sequence of length $n$, $x$ is the element to search for

1: function NearlySortedSearch($S, x$):
2: Sample $O(\log \log n)$ indices from $S_{9n/20}, \ldots, S_{11n/20}$
3: if $x$ is found then
4: return corresponding index for $x$
5: else if a majority of queried values are greater than $x$ then
6: return NearlySortedSearch($(S_1, \ldots, S_{11n/20}), x$)
7: else # a weak majority of queried values are less than $x$
8: return NearlySortedSearch($(S_{9n/20}, \ldots, S_n), x$)

binary search procedure in the nearly sorted list. With constant probability, we will be correct in claiming that either $S^2$ or $S^4$ are $(\epsilon/\log n)$-close to being sorted for every iteration because we repeat the $(\epsilon/\log n)$-tester $O(\log \log n)$ times. Assuming this is the case, we will fail to find at most a $O(\epsilon/\log n)$ fraction of elements in each recursive call. Naively, this looks like we should be able to find at least $n - O(\epsilon n)$ elements with constant probability since there are only $O(\log n)$ iterations. However, the sequences on which we search for $(2, 1)$ pairs depend on $x$. It could be the case that each $x$ naturally leads itself to a bad configuration where it is difficult to find using the randomized binary search procedure of Algorithm 2.

There are two possible cases here: 1) not many values of $x$ get cornered into a bad position like this, so some analysis would show that Algorithm 1 works; or 2) this idea is inherently flawed in some way. We hope 1) actually holds, and we are working towards figuring out a better way to analyze this algorithm. However, we are also looking at new approaches in the scenario where case 2) holds.

We now briefly argue why being able to searching in a $\pi$-avoiding sequence solves Generalized Intersection Search($\pi, \epsilon$) with constant probability.

**Lemma 9.** If there exists an algorithm that finds at least $n - \epsilon n/2$ elements with constant probability in a $\pi$-avoiding sequence using $q(n, \epsilon)$ queries, then Generalized Intersection Search($\pi, \epsilon$) can be solved with $O(\frac{1}{\epsilon}q(n, \epsilon))$ queries.

**Proof.** Let $A$ and $B$ be the two $\pi$-avoiding sequences for the Generalized Intersection Search problem. There are at least $n - \epsilon n/2$ elements that can be found in $B$ by assumption, so at least $\epsilon n/2$ of the elements in common between $A$ and $B$ can be found in $B$. Thus, after searching for $O(1/\epsilon)$ random elements from $A$ in $B$, one of them should be able to be found. This element will be found in $B$ with constant probability, solving Generalized Intersection Search($\pi, \epsilon$).

Assuming Algorithm 1 is correct, the previous lemma implies that Generalized Intersection Search($\pi, \epsilon$) can be solved using $O(\frac{1}{\epsilon^2} \log^3 n \log \log n) = \text{poly}(\frac{1}{\epsilon}, \log n)$ queries.

### 4.2 Searching in Nearly Sorted Sequences

We consider a variant of binary search where instead of choosing a single pivot from a sequence $S$ deterministically, we sample $O(\log \log n)$ indices at random from the middle 10% block of elements
in $S$. If we are searching for $x$, we check whether or not the majority of sampled indices are greater than $x$. If so, we restrict our search to the first 55% of the sequence. Otherwise, we restrict our search to the last 55% of the sequence. We repeat until we find $x$ in the set of sampled points.

Algorithm 2 outlines the ideas above.

**Theorem 2.** If a sequence $S$ is $\alpha$-close to being sorted, then the nearly sorted search procedure from Algorithm 2 uses $O(\log n \log \log n)$ queries and finds at least a $1 - O(\alpha)$ fraction of the elements in $S$ with at least $2/3$ probability.

**Proof.** Query Complexity. In each recursive call of randomized binary search, the sequence is restricted to $S_1, \ldots, S_{11n/20}$ or $S_{9n/20}, \ldots, S_n$, so the length of the new sequence is at most $11/20$ the original size. Therefore, there can be at most $2 \log n$ recursive calls made. In each call, $O(\log \log n)$ indices are queried, so at most $O(\log n \log \log n)$ indices are queried.

Correctness. We assume that the initial sequence $S$ is $\alpha$-close to being sorted. This means there exists a set of bad elements $B$ of size at most $\alpha n$ such that the deletion of all of the corresponding indices in $B$ makes $S$ sorted.

We say that a recursive call is BAD if at least $1/3$ of the elements from $S_{9n/20}, \ldots, S_{11n/20}$ are in $B$, otherwise we say the recursive call is GOOD. We will say that an element is searchable if every possible recursive call on $S$ containing $x$ is GOOD, otherwise we say that is it unsearchable.

We first show that at least $n - O(\alpha n)$ elements are searchable. We then show that randomized binary search will find each searchable element with probability at least $2/3$, which will complete the proof of correctness.

**Claim 10.** At most $60\alpha$ elements in $S$ are unsearchable.

**Proof.** We will think of all paths in our randomized binary search algorithm as a binary decision tree, where each node represents a possible recursive call. Any time we encounter a BAD recursive call, we will cut off the binary tree at that node and label all of the elements in that recursive call as unsearchable.

First we claim that all elements in $B$ are unsearchable. Let $x$ be an arbitrary element of $B$. If we did not stop in the binary decision tree at any BAD recursive calls, then there must be some call where $x$ is the lone element in the middle block that we sample indices from. This is clearly a BAD recursive call since $x$ is in $B$. Either this recursive call is reachable in the binary decision tree, or some ancestor of this node is BAD. In either case, $x$ is part of some BAD recursive call, so $x$ is unsearchable.

Next we claim that for every element in $B$, there are at most 60 other unsearchable elements. We do this by showing it is possible to assign at most 60 unsearchable elements to each $x$ in $B$ in a way that every unsearchable element is assigned some element in $B$.

Let $x$ be an element in $B$, and consider all BAD recursive calls for which $x$ is in the middle block. If there are no such calls, we do not assign any elements to $x$. If there is one such call, then of the middle 10%, we know at least $1/3$ of the elements are in $B$. Thus, there are at most 30 times as many unsearchable elements in this recursive call than elements in $B$. We assign 30 unsearchable elements to $x$, and the rest of the unsearchable elements will be assigned to other elements in $B$ to cover the sequence for that recursive call.

If $x$ is in more than one BAD recursive call where it is in the middle block, we first consider the calls of minimal depth in the binary decision tree. From the previous argument, at most 30
elements will be assigned to $x$ for each BAD recursive call. However, we claim that we only need to consider two such BAD recursive calls: the one where $x$ is furthest left in the middle block and the one where it is furthest right in the middle block. All unsearchable elements in other BAD recursive calls will be covered by these two cases. For each of these cases, by the argument for a single BAD recursive call, we can assign at most 30 elements to $x$. So $x$ will be assigned at most 60 unsearchable elements, and all unsearchable elements in that range can be covered by other elements in $B$.

Lastly note that any BAD recursive calls at a greater depth in the tree that $x$ is in the middle block are completely contained in the ones of minimal depth, so all the unsearchable elements are already assigned to element in $B$.

**Claim 11.** The randomized binary search algorithm succeeds in finding each searchable element with probability at least $2/3$.

**Proof.** Let $x$ be an arbitrary searchable element. We first show that after every GOOD recursive call, $x$ is still in the search space with probability $1 - 1/6\log n$.

There are three cases to consider: 1) $x$ is in the middle block $S_{9n/20}, \ldots, S_{11n/20}$, 2) $x$ is to the left of the middle block, or 3) $x$ is to the right of the middle block.

In case 1), $x$ will always remain in the search space since both recursive calls contain the elements $S_{9n/20}, \ldots, S_{11n/20}$ in the next subsequence to search.

In cases 2) and 3), every element not in $B$ will be a correct witness for which direction to search for $x$. This is because the elements not in $B$ form an increasing subsequence, and since $x$ is searchable, it is also not in $B$. Then if $x$ is to the left of the middle block, every queried element not in $B$ will be greater than $x$, and similarly if $x$ is to the right of the middle block, every queried element not in $B$ will be less than $x$.

Let $X$ be a random variable counting the number of correct responses after querying $k$ indices. Because we assume it is a GOOD recursive call, $E[X] > 2k/3$. Using a Chernoff bound, the probability that less than a majority give the correct response, i.e. $X \leq k/2$, is at most

$$\Pr[E[X] - X > k/6] \leq e^{-2(1/6)^2k} < 2^{-k/18}.$$  

Furthermore, if $k \geq 18\log\log(6n)$, then $x$ is still in the search space with probability at least $1 - 1/(6\log n)$.

This implies that the algorithm will find each searchable elements if all GOOD recursive calls succeed. Since the indices chosen are all independent in each round, the probability this occurs is at least

$$(1 - 1/(6\log n))^{2\log n} > 2/3.$$  

**5 Existence of Properties Requiring Linear Queries**

We shift our focus now to show that in general, there exist properties that require $\Omega(n)$ queries. In a concurrent work, we show that if all hereditary properties can be solved with $q(n, \epsilon)$ queries, then all properties can be solved with $q(n, \epsilon/2)$ queries. Therefore, this result implies that there must be
hereditary properties of sequences requiring $\Omega(n)$ queries. However, the pattern avoiding properties we consider are a subset of hereditary properties of sequences, so it is still possible these can all be tested using sublinear queries.

In a similar vein to [BFNR03], we show that most properties of random binary sequences require $\Omega(n)$ queries. In [BFNR03], they consider the quantum property testers with Hamming distance. We consider classical property testers using edit distance, for which we can give a more elementary proof.

**Theorem 3.** Fix $\epsilon = 1/100$. There exists a property such that for any two-sided $\epsilon$-tester requires at least $n/6000$ queries.

**Proof.** Our definition of an $\epsilon$-tester requires the algorithm is correct with probability at least $2/3$. We note that by repeating the tester $k$ times and taking the majority decision, we can amplify the correctness probability to any constant. For this argument, we use the constant $9/10$. A Chernoff bound shows that we can amplify a tester that is correct with probability $2/3$ to $9/10$ by running the $2/3$ tester $k \geq 60$ times.

Let $P$ be a random property of $|P| = 2^{n/20}$ binary sequences of length $n$. We want to show there exists a property $P$ such that there is no (possibly adaptive) randomized tester for $P$ using $q = n/100$ queries that is correct with $9/10$ probability. To do this, we will show that

$$\Pr_P[\exists \text{ randomized tester for } P] < 1.$$

If there exists a randomized tester for the property $P$, then Yao’s principle tells us that for any fixed distribution $X_P$ over inputs to the tester, there exists a deterministic tester for $P$ which succeeds with probability at least $9/10$ on inputs drawn from $X_P$. We will take $X_P$ to be the following distribution:

$$X_P = \begin{cases} \text{uniform } x \in P & \text{w.p. } 1/2 \\ \text{uniform } x \epsilon \text{-far from } P & \text{w.p. } 1/2 \end{cases}$$

Now we have

$$\Pr_P[\exists \text{ randomized tester for } P] \leq \Pr_P[\exists \text{ deterministic tester } T : \left( \Pr_{x \sim X_P} [T(x) \text{ is correct}] > 9/10 \right)]$$

$$\leq (\# \text{ deterministic testers}) \cdot \max_T \Pr_P \left[ \Pr_{x \sim X_P} [T(x) \text{ is correct}] > 9/10 \right]$$

We first bound the number of deterministic adaptive testers using at most $q = n/100$ queries. The state space of a deterministic adaptive property testing algorithm making $q$ queries can be represented by a binary tree of depth $q$. The nodes represent the queries and the edges represent the two possible state transitions from the response for each query. We label each node with the query the algorithm makes at that state. Accounting for the decision that the algorithm makes to accept or reject after $q$ queries, the total number of deterministic testers is

$$2^{2q} \cdot \prod_{i=1}^{q} (n - i + 1)^{2i} \leq 2^{2q/50}.$$

Note that any tester using fewer than $q$ queries is equivalent to a tester that makes exactly $q$ queries but ignores some responses.
Next we bound the second term from the union bound above. Note that

\[
\Pr_{x \sim X_P} [T(x) \text{ is correct}] = \frac{1}{2} \cdot \Pr_{x \sim X_P} [T \text{ accepts } | x \in P] + \frac{1}{2} \cdot \Pr_{x \sim X_P} [T \text{ rejects } | x \text{ is } \epsilon \text{-far from } P].
\]

For this probability to be greater than \(9/10\), the probability that \(T(X)\) is correct must be at least \(4/5\) both when \(x \in P\) and \(x\) is \(\epsilon\)-far from \(P\).

Let \(T_A = \{x : T(x) \text{ accepts}\}\). We show below in Lemma 12 that the size of an \(\epsilon\)-ball in this space is at most \((20/\epsilon)^{2\epsilon n} \leq 2^{n/4}\). Therefore, at least \(2^n - |P| \cdot |B_\epsilon| \geq 2^n - 2^{n/2}\) sequences are \(\epsilon\)-far from \(P\). This means that we must reject at least \(\frac{4}{5} \cdot (2^n - 2^{n/2})\) sequences, so \(|T_A| \leq 2^n - \frac{4}{5} \cdot (2^n - 2^{n/2}) \leq 2^n/4\). Therefore the probability that \(T\) accepts a random sequence is at most \(1/4\). We want to use this to bound the probability that \(T\) accepts at least \(4/5\) \(|P|\) sequences for a random property \(P\). In this case, a Chernoff bound shows that

\[
\Pr_P \left[ \Pr_{x \sim X_P} [T(x) \text{ is correct}] > 4/5 \right] < e^{-2(4/5 - 1/4)^2 |P|} < 2^{-|P|/2}
\]

Since \(|P| = 2^{n/20}\), this implies that

\[
\max_T \Pr_P \left[ \Pr_{x \sim X_P} [T(x) \text{ is correct}] > 9/10 \right] < 2^{-\frac{1}{2} \frac{2^n}{20}}.
\]

Finally, we get

\[
\Pr_P [\exists \text{ randomized tester for } P] \leq 2^{2^{n/50}} \cdot 2^{-\frac{1}{2} \frac{2^n}{20}} < 1.
\]

It remains to bound the number of sequences \(\epsilon\)-far from a sequence in \(P\).

**Lemma 12.** The size of an \(\epsilon\)-ball in the space of binary sequences of length \(n\) under edit distance is at most \((20/\epsilon)^{2\epsilon n}\) as long as \(\epsilon n > 25\).

**Proof.** Let \(|B_\epsilon|\) be the maximum size of an \(\epsilon\)-ball for binary sequences of length \(n\) under edit distance. We note that edit distance allows for insertions, substitutions, and deletions, but a single substitution can be done using a deletion and an insertion. Thus, \(|B_\epsilon|\) is bounded by a ball of radius \(2\epsilon n\) using only insertions and deletions. For a fixed distance \(d\), we think of all \(\delta\) deletions happening first followed by \(d-\delta\) insertions. There are \(\binom{n}{\delta}\) possible positions to delete and \(\binom{n+d-2\delta}{d-\delta}\) ways to insert \(d-\delta\) values into the remaining sequence. Summing over the possible distances and number
of deletions, we get

\[ |B_\epsilon| \leq \sum_{d=0}^{2\epsilon n} \sum_{\delta=0}^{d} \left( \begin{array}{c} n \\ \delta \end{array} \right) \cdot \left( \begin{array}{c} n + d - 2\delta \\ d - \delta \end{array} \right) \cdot 2^{d-\delta} \]

\[ \leq \sum_{d=0}^{2\epsilon n} d \cdot \left( \begin{array}{c} n \\ d \end{array} \right) \cdot \left( \begin{array}{c} n + d \\ d \end{array} \right) \cdot 2^d \]

\[ \leq (2\epsilon n)^2 \left( \begin{array}{c} n \\ 2\epsilon n \end{array} \right) \cdot \left( \begin{array}{c} n + 2\epsilon n \\ 2\epsilon n \end{array} \right) \cdot 2^{2\epsilon n} \]

\[ \leq (2\epsilon n)^2 \left( \frac{2e}{\epsilon} \right)^{2\epsilon n} \]

\[ \leq (20/\epsilon)^{2\epsilon n} \]

\[ \square \]

6 Conclusions and Open Problems

We propose Generalized Intersection Search as a possible technique to prove lower bounds for pattern avoidance, and look at this problem for \((3, 2, 1)\). One obvious question is if the proposed algorithm for Generalized Intersection Search\((3, 2, 1), \epsilon\) can be proven correct at all? Also, can similar ideas be used to solve Generalized Intersection Search for other patterns, possibly non-monotone like \((1, 3, 2)\)? If so, then that may be evidence that constant-size patterns can be tested adaptively with \(\text{poly}(\frac{1}{\epsilon}, \log n)\) queries, an open question originally posed by \[NRRS17\].

Our randomized binary search procedure given in Algorithm 2 uses \(O(\log n \log \log n)\) queries, but it seems feasible that only \(O(\log n)\) queries may be necessary. Is it possible to modify our approach to use only a constant number of queries in each recursive call while still cutting the search space by a constant factor in expectation?

Our final result implies that hereditary properties of sequences require \(\Omega(n)\) queries in general. It is possible that order-based hereditary properties, i.e. those that depend only on the relative order of the elements, can be tested using sublinear queries.

It is known that any specific constant length pattern can be tested with sublinear queries. However, Spielman and Bóna \[SB00\] showed that there exist infinite antichains of permutation patterns. Could testing a sequence for the avoidance of all patterns in an infinite antichain require linear queries?

7 Acknowledgements

The author gives a huge thanks to Prof. Eric Price for advising this thesis and to William Swartworth for all of the collaboration on this work. Also, the author thanks the other thesis committee members, Prof. Anna Gal and Prof. Vijaya Ramachandran, for taking the time to evaluate this work. Lastly, the author wants to thank Prof. Calvin Lin for all of the work he does for the Turing Scholars program at UT Austin.
References


