Managing and analyzing huge data sets forces us to reconsider the traditional notions of efficient algorithms. Processing such massive data sets in polynomial time is by far too expensive, and often even linear time algorithms may be too slow. Often we would like to design algorithms that run in sublinear time.

Consider the problem: Given an array of distinct elements, we would like to know whether it is monotonic increasing. Clearly, this problem can be solved in linear time. Can we design an algorithm that runs in less than linear time? Observe that to fulfill requirement “sublinear time”, the algorithm can not even read its input entirely. Any sublinear time algorithm can look only a portion of the input. In such scenarios it might be too ambitious to solve the problem “exactly”, we can only hope to solve an “approximate” version of the problem.

**Definition:** A list of size \(n\) is \(\epsilon\)-close to sorted if by changing at most \(\epsilon n\) elements, the list becomes sorted.

**Observation:** If a list \(\epsilon\)-close sorted if and only if it has an increasing sub-sequence of length \((1 - \epsilon)n\).

If the list is \(\epsilon\)-close sorted, then by changing at most \(\epsilon n\) elements, the list becomes sorted. This means that elements that are not changed must be in increasing order. Thus there is an increasing subsequence of length at least \((1 - \epsilon)n\).

Consider a list that has an increasing subsequence of length at least \((1 - \epsilon)n\). There are at most \(\epsilon n\) elements that do not belong to this subsequence. By changing these elements, we can make the list sorted. Thus the list is \(\epsilon\)-close to sorted.

Our goal is to find an algorithm \(A\) with following properties. If the list is sorted, then the algorithm accepts with probability one. If the list is \(\epsilon\)-away from sorted, then the algorithm rejects with high probability. We require that the algorithm runs in \(O(\log n)\) time.

As a first attempt, consider the following algorithm: Randomly pick \(i \in \{1, 2, ..., n\}\). If \(a[i] > a[i + 1]\), then reject. else accept.

If the list is sorted, then the algorithm accepts with probability one.

Consider the following list:

Consider a list of \(n\) elements having \(\sqrt{n}\) number of subsequences, each of length \(\sqrt{n}\) whose elements placed as in the following sequences.

\[
a_1 \quad - \quad - \quad - \quad b_1, \quad a_2 \quad - \quad - \quad - \quad b_2, \quad ..., \quad a_{\sqrt{n}} \quad - \quad - \quad - \quad b_{\sqrt{n}}
\]
with \( b_2 < a_1, \ b_3 < a_2, \text{ etc.} \)

Every increasing subsequence of the list is of size \( \sqrt{n} \). Thus the list is
\[
(1 - \frac{1}{\sqrt{n}})
\]
away from being sorted. The algorithm rejects this list, only when
\( a[i] \) is one of \( b_1, \ldots b_{\sqrt{n}} \). Thus the algorithm rejects with probability at most
\( 1/\sqrt{n} \). To make this probability significant, we have to repeat the above
algorithm for \( O(\sqrt{n}) \) steps.

Now consider the following algorithm: Randomly pick \( i,j \) (say \( i < j \)). If
\( a[i] > a[j] \) then reject, else accept. Clearly, this algorithm rejects the above
list with high probability. However, we can find another list that is far away
from being sorted and yet this algorithm accepts with high probability.

The first algorithm compares elements and consecutive indices, whereas the
second algorithm (with high probability) compares elements at far away
indices. We arrive at a correct algorithm by following a mixed strategy:

1. Repeat Steps 1, 2, and 3 for \( m \) times.
2. Randomly pick \( i \in \{1, 2, \ldots, n\} \)
3. For \( k = 0 \) to \( \log i \), repeat following for \( \ell \) steps.
   (a) Randomly pick \( j \in \{1, 2, \ldots, 2^k\} \)
   (b) If \( a[i - j] > a[i] \) then reject
4. For \( k = 0 \) to \( \log(n - i) \), repeat the following for \( \ell \) times.
   (a) Randomly pick \( j \in \{1, 2, \ldots, 2^k\} \)
   (b) If \( a[i + j] < a[i] \) then reject
5. accept

Running time of this algorithm is \( m(l \log i + l \log(n - i)) \leq 2ml \log n \). It
is obvious that if the given list is already sorted, then this algorithm accepts
with probability one.

Now assume that the list is \( \epsilon \)-away from being sorted.

**Definition:** An index \( i \) is heavy if for every \( k \) more than \( \frac{3}{4}2^k \) elements
from \( [i, \ldots i + 2^k] \) are bigger than \( a[i] \) and more than \( \frac{3}{4}2^k \) elements from \( [i -
2^k, \ldots, i] \) are less than \( a[i] \).

**Claim:** If \( i \) and \( j \) are heavy and \( i < j \), then \( a[i] < a[j] \).

**Proof:** Let \( i \) and \( j \) be heavy indices with \( i < j \). There exists \( l \) such that
\( 2^l \leq j - i < 2^{l+1} \).
Divide the interval \([j - 2^{l+1}, i + 2^{l+1}]\) into five disjoint subintervals as follows: \(A = [j - 2^{l+1}, i], B = [i, j - 2^l], C = [j - 2^l, i + 2^l], D = [i + 2^l, j]\) and \(E = [j, i + 2^{l+1}]\).

Case 1: Let \(j - i - 2^l \leq \frac{1}{2} 2^l \). (2)
Recall that \(i\) and \(j\) are heavy. So

1. More than \(\frac{3}{4} 2^l\) from \([i, ..., i + 2^l]\) are bigger than \(a[i]\).
2. More than \(\frac{3}{4} 2^l\) form \([j - 2^l, ..., j]\) are less than \(a[j]\).

The lengths of \(B\) and \(D\) be \(j - i - 2^l\) and length of \(C\) is \(2^{l+1} - (j - i)\). Consider all elements that satisfy Property 1. In the worst-case, all elements from \(D\) have this property. Similarly all elements from \(B\) have property 2. Thus \(C\) must contain at least \(\frac{3}{4} 2^l - (j - i - 2^l)\) elements that are bigger than \(a[i]\) and at least \(\frac{3}{4} 2^l - (j - i - 2^l)\) elements that are less than \(a[j]\). However the length of \(C\) is \(2^{l+1} - (j - i)\). Thus, by pigeon hole principle, there exists an element in \(C\) that is bigger than \(a[i]\) and less than \(a[j]\). Thus \(a[i] < a[j]\).

The case when \(j - i - 2^l > \frac{1}{2} 2^l\) can be handled via similar arguments.

**Corollary:** If the list has greater than \((1 - \epsilon)n\) heavy elements, then it has an increasing subsequence of length greater than \((1 - \epsilon)n\). So it is \(\epsilon\)-close to being sorted.

Conversely, if the list is \(\epsilon\)-away from sorted then list has less than \((1 - \epsilon)n\) heavy elements.

Now our task is to show that if the given array is \(\epsilon\)-away from sorted, then the above algorithm accepts with probability less than \(\delta\).

If the list is \(\epsilon\)-away from sorted then the algorithm accepts when one of the following happens.

- All \(i'\)'s picked are heavy.

- A non-heavy \(i\) will survive both inner loops.

The probability of the first event is at most \((1 - \epsilon)^m\). If \(i\) is non heavy then at least \(\frac{1}{4} 2^k\) elements from \([i, ..., i + 2^k]\) are less than \(a[i]\). Thus the probability of second event is at most \((\frac{3}{4})^l\).

If we set \(1 - \epsilon)^m \leq \delta/2\) and \((\frac{3}{4})^l \leq \delta/2\), then we obtain \(m = \frac{1}{\epsilon} \log \frac{1}{\delta}\) and \(l = \log \frac{1}{\delta}\).

Consider the following question: Given an undirected, \(d\)-regular graph, is it connected? Assume adjacency list of the graph is given.

**Definition:** A graph \(G\) is said to be \(\epsilon\)-away from connected if it can be made connected by adding at least \(\epsilon dn\) edges to it.
Our goal is to design an algorithm that accepts with probability one when the graph is connected, and rejects with high probability when the graph is $\epsilon$-away from being connected.

Observe that if the graph is $\epsilon$-away from being connected, then its has at least $\epsilon d n$ components. Call a component “small” if the number of vertices in the component is at most $\frac{2}{\epsilon d}$. The number of small component is at least $\epsilon d n / 2$. Since each component has at least one vertex, total number of vertices in all small components is at least $\epsilon d n / 2$. Now consider the following algorithm.

- Randomly pick $v \in V$
- Do a BFS for $\frac{2}{\epsilon d}$ steps. If the BFS takes back to $v$ REJECT.
- Accept.

The total running time for this algorithm $< \frac{2}{\epsilon d} d = \frac{2}{\epsilon}$. If the graph is connected, then the algorithm accepts with probability. If the graph is $\epsilon$-away from connected, the with probability at least $1/2$ it picks a a vertex from a small component. When this happens, the algorithm rejects.