# Revisiting Time-Space Tradeoffs for Function Inversion

**Spencer Peters**

Noah S.D.

Siyao Guo

Sasha Golovnev

# Function Inversion

- Given a function

$$\{1, 2, \dots, N\} =: [N] \longrightarrow \boxed{f} \longrightarrow [N]$$

# Function Inversion

- Given a function, and a point $y$ in its range,

$$\{1, 2, \ldots, N\} =: [N] \longrightarrow \boxed{f} \longrightarrow y \in [N]$$

# Function Inversion

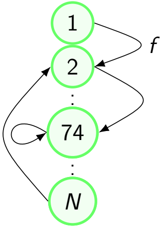
- Given a function, and a point $y$ in its range, find $x$ with $f(x) = y$.

$$\{1, 2, \dots, N\} =: [N] \ni x \longrightarrow \boxed{\quad f \quad} \longrightarrow y \in [N]$$

# Function Inversion

- Given a function, and a point $y$ in its range, find $x$ with $f(x) = y$.

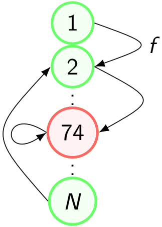
$$\{1, 2, \dots, N\} =: [N] \ni x \longrightarrow \boxed{f} \longrightarrow y \in [N]$$



# Function Inversion

- Given a function, and a point $y$ in its range, find $x$ with $f(x) = y$.

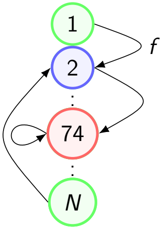
$$\{1, 2, \ldots, N\} =: [N] \ni x \longrightarrow \boxed{f} \longrightarrow y \in [N]$$



# Function Inversion

- Given a function, and a point $y$ in its range, find $x$ with $f(x) = y$.

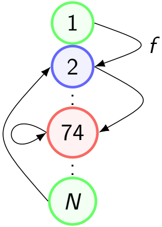
$$\{1, 2, \dots, N\} =: [N] \ni x \longrightarrow \boxed{f} \longrightarrow y \in [N]$$



# Function Inversion

- Given a function, and a point $y$ in its range, find $x$ with $f(x) = y$.

$$\{1, 2, \ldots, N\} =: [N] \ni x \longrightarrow \boxed{f} \longrightarrow y \in [N]$$

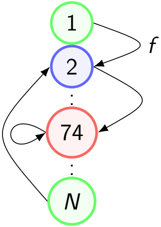


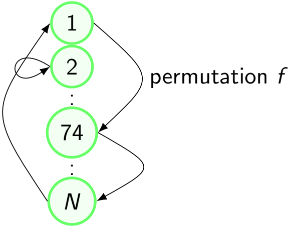

- The study of this *black-box* function inversion problem was initiated by Martin Hellman  in 1980 [Hel80].

# Function Inversion

- Given a function, and a point $y$ in its range, find $x$ with $f(x) = y$.

$$\{1, 2, \ldots, N\} =: [N] \ni x \longrightarrow \boxed{f} \longrightarrow y \in [N]$$

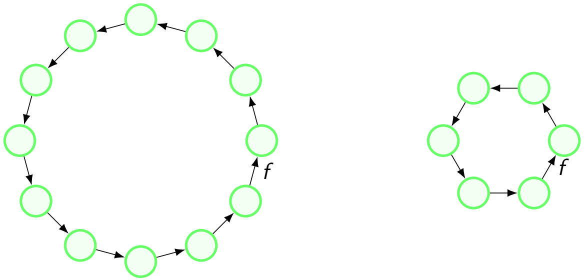



- The study of this *black-box* function inversion problem was initiated by Martin Hellman  in 1980 [Hel80].

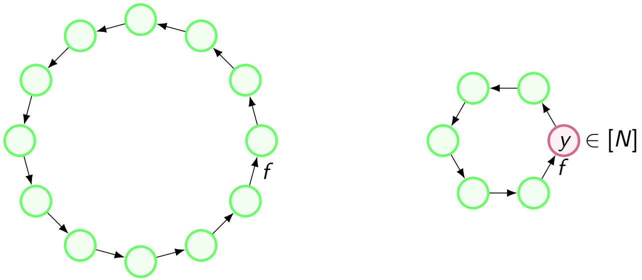
# Hellman's algorithm

- If $f$ is a permutation, its *graph* is a disjoint union of cycles.



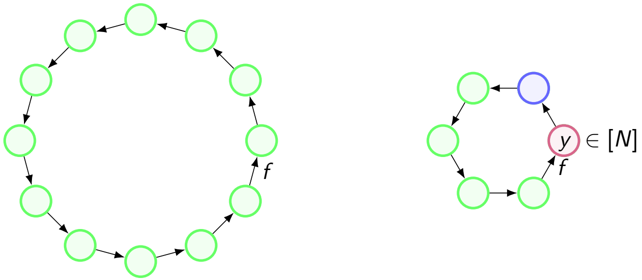
# Hellman's algorithm

- If $f$ is a permutation, its *graph* is a disjoint union of cycles.



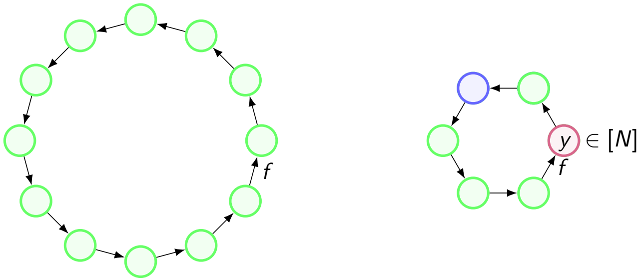
# Hellman's algorithm

- If $f$ is a permutation, its *graph* is a disjoint union of cycles.



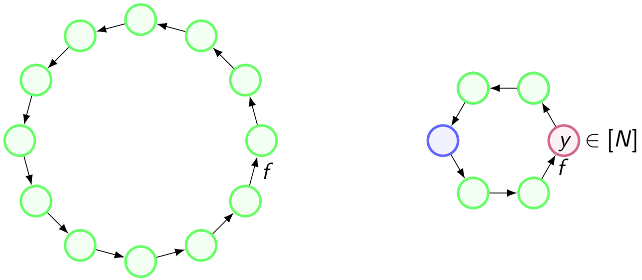
# Hellman's algorithm

- If $f$ is a permutation, its *graph* is a disjoint union of cycles.



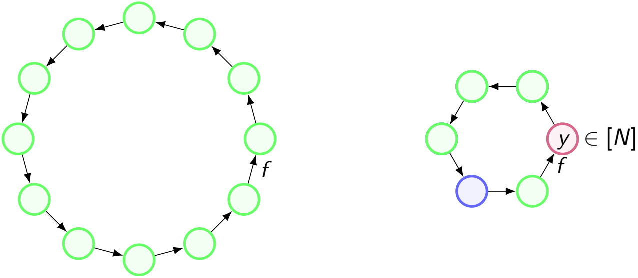
# Hellman's algorithm

- If $f$ is a permutation, its *graph* is a disjoint union of cycles.



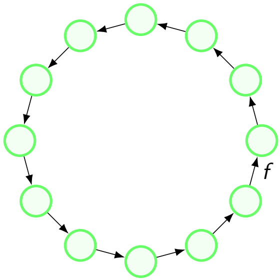
# Hellman's algorithm

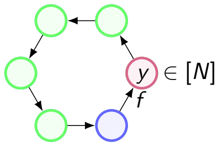
- If $f$ is a permutation, its *graph* is a disjoint union of cycles.



# Hellman's algorithm

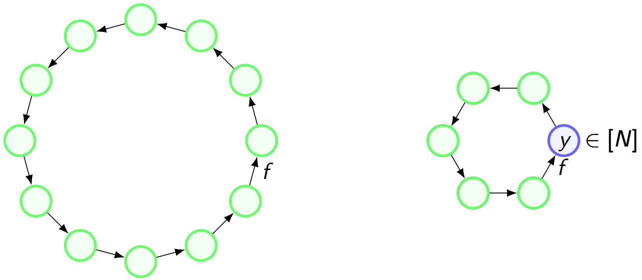
- If $f$ is a permutation, its *graph* is a disjoint union of cycles.



# Hellman's algorithm

- If $f$ is a permutation, its *graph* is a disjoint union of cycles.

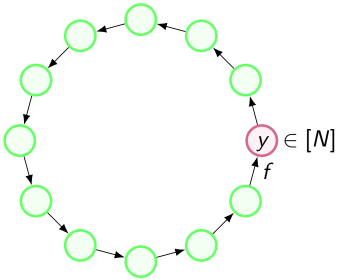


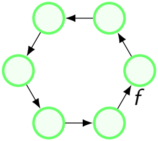
# Hellman's algorithm

- What if $y$ is on a large cycle?

# Hellman's algorithm

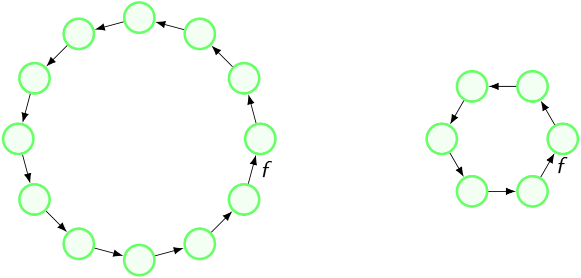
- What if $y$ is on a large cycle?



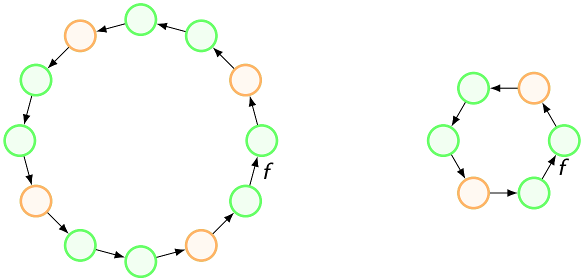
# Hellman's algorithm

- What if $y$ is on a large cycle?



# Hellman's algorithm

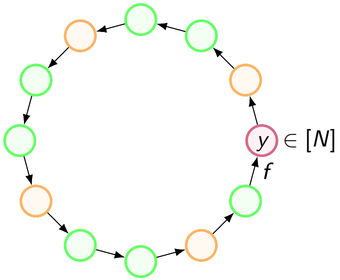
- What if $y$ is on a large cycle?




- In a preprocessing step, store points uniformly spaced around each cycle.

# Hellman's algorithm

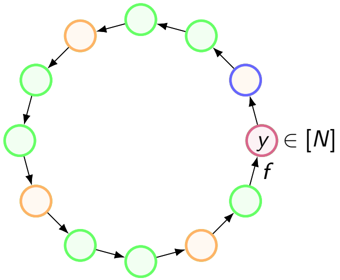
- What if $y$ is on a large cycle?




- In a preprocessing step, store points uniformly spaced around each cycle.

# Hellman's algorithm

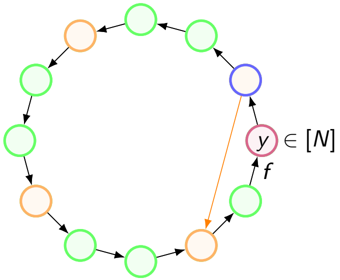
- What if $y$ is on a large cycle?

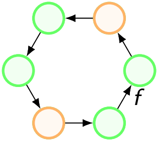



- In a preprocessing step, store points uniformly spaced around each cycle.

# Hellman's algorithm

- What if $y$ is on a large cycle?

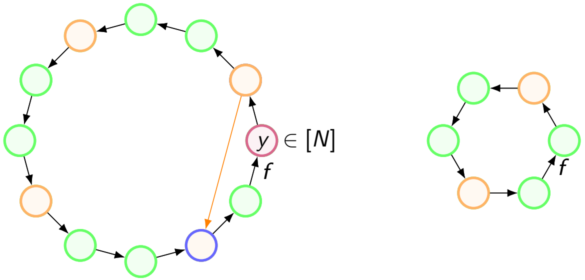



- In a preprocessing step, store points uniformly spaced around each cycle.

# Hellman's algorithm

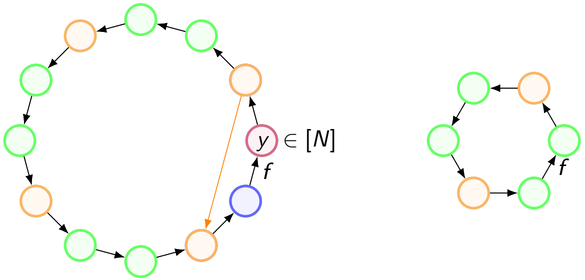
- What if $y$ is on a large cycle?




- In a preprocessing step, store points uniformly spaced around each cycle.

# Hellman's algorithm

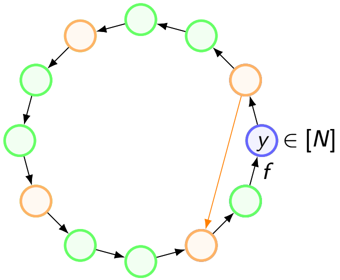
- What if $y$ is on a large cycle?

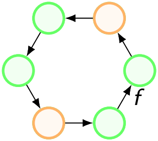



- In a preprocessing step, store points uniformly spaced around each cycle.

# Hellman's algorithm

- What if $y$ is on a large cycle?



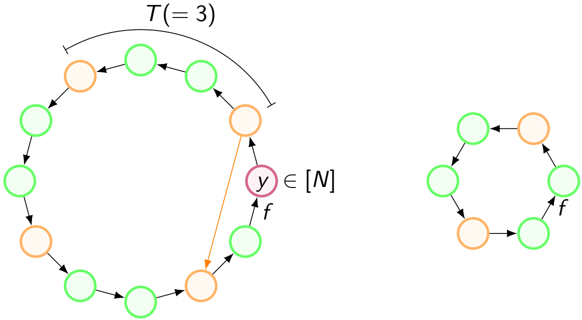

- In a preprocessing step, store points uniformly spaced around each cycle.

# Analysis

- If we space the stored points $T$ hops apart:

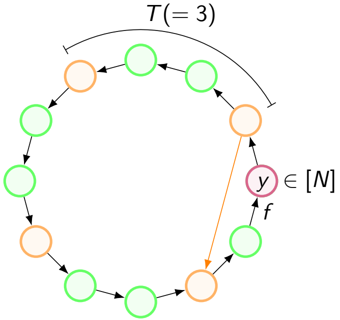
# Analysis

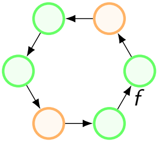
- If we space the stored points $T$ hops apart:



# Analysis

- If we space the stored points $T$ hops apart:

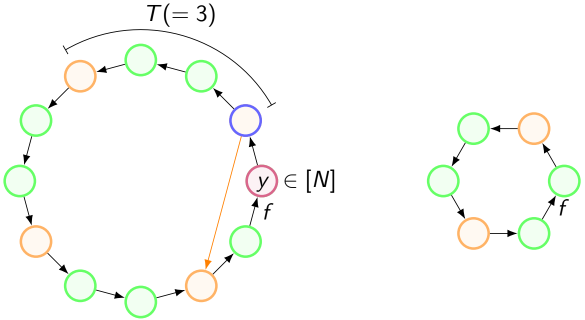



- We need $T$ evaluations of $f$ to invert $y$.

# Analysis

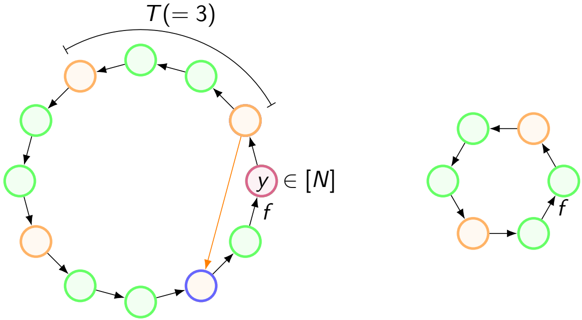
- If we space the stored points $T$ hops apart:




- We need $T$ evaluations of $f$ to invert $y$.

# Analysis

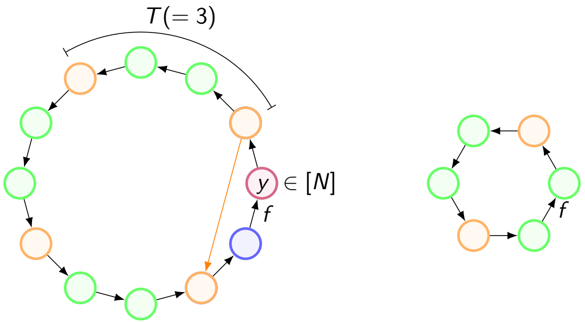
- If we space the stored points $T$ hops apart:




- We need $T$ evaluations of $f$ to invert $y$.

# Analysis

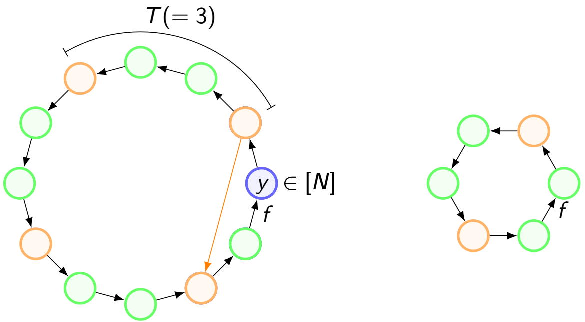
- If we space the stored points $T$ hops apart:




- We need $T$ evaluations of $f$ to invert $y$.

# Analysis

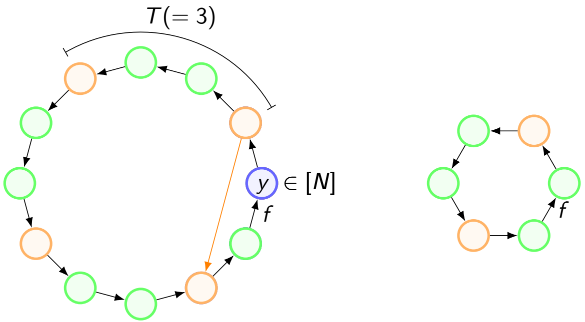
- If we space the stored points $T$ hops apart:




- We need $T$ evaluations of $f$ to invert $y$.

# Analysis

- If we space the stored points $T$ hops apart:




- We need $T$ evaluations of $f$ to invert $y$.
- We need to store about $N/T$ points total.

# What did we just do?

- We have a preprocessing algorithm $\mathcal{P}$ and an online algorithm $\mathcal{A}$. Preprocessing makes a $S$-bit data structure $\alpha = \mathcal{P}(f)$.

# What did we just do?

- We have a preprocessing algorithm $\mathcal{P}$ and an online algorithm $\mathcal{A}$. Preprocessing makes a $S$-bit data structure $\alpha = \mathcal{P}(f)$.
- $\mathcal{A}^f(\alpha, y)$ uses $\alpha$ to invert $y$. It treats $f$ as a *black box* and evaluates it $\leq T$ times.

# What did we just do?

- We have a preprocessing algorithm $\mathcal{P}$ and an online algorithm $\mathcal{A}$. Preprocessing makes a $S$-bit data structure $\alpha = \mathcal{P}(f)$.
- $\mathcal{A}^f(\alpha, y)$ uses $\alpha$ to invert $y$. It treats $f$ as a *black box* and evaluates it $\leq T$ times.
- Hellman's algorithm uses $S = O((N \log N)/T)$.

# What did we just do?

- We have a preprocessing algorithm $\mathcal{P}$ and an online algorithm $\mathcal{A}$. Preprocessing makes a $S$-bit data structure $\alpha = \mathcal{P}(f)$.
- $\mathcal{A}^f(\alpha, y)$ uses $\alpha$ to invert $y$. It treats $f$ as a *black box* and evaluates it $\leq T$ times.
- Hellman's algorithm uses $S = O((N \log N)/T)$.
- More generally, **the goal** is to design a preprocessing algorithm $(\mathcal{P}, \mathcal{A})$ such that for all $f$ and all $y \in f([N])$,

$$\Pr[\alpha \leftarrow \mathcal{P}(f); x \leftarrow \mathcal{A}^f(\alpha, y); f(x) = y] \geq 9/10.$$

# What did we just do?

- We have a preprocessing algorithm $\mathcal{P}$ and an online algorithm $\mathcal{A}$. Preprocessing makes a $S$-bit data structure $\alpha = \mathcal{P}(f)$.
- $\mathcal{A}^f(\alpha, y)$ uses $\alpha$ to invert $y$. It treats $f$ as a *black box* and evaluates it $\leq T$ times.
- Hellman's algorithm uses $S = O((N \log N)/T)$.
- More generally, **the goal** is to design a preprocessing algorithm $(\mathcal{P}, \mathcal{A})$ such that for all $f$ and all $y \in f([N])$,

$$\Pr[\alpha \leftarrow \mathcal{P}(f); x \leftarrow \mathcal{A}^f(\alpha, y); f(x) = y] \geq 9/10.$$

- In this model, $\mathcal{P}$ and $\mathcal{A}$ have **unbounded** computational power,

# What did we just do?

- We have a preprocessing algorithm $\mathcal{P}$ and an online algorithm $\mathcal{A}$. Preprocessing makes a $S$-bit data structure $\alpha = \mathcal{P}(f)$.
- $\mathcal{A}^f(\alpha, y)$ uses $\alpha$ to invert $y$. It treats $f$ as a *black box* and evaluates it $\leq T$ times.
- Hellman's algorithm uses $S = O((N \log N)/T)$.
- More generally, **the goal** is to design a preprocessing algorithm $(\mathcal{P}, \mathcal{A})$ such that for all $f$ and all $y \in f([N])$,

$$\Pr[\alpha \leftarrow \mathcal{P}(f); x \leftarrow \mathcal{A}^f(\alpha, y); f(x) = y] \geq 9/10.$$

- In this model, $\mathcal{P}$ and $\mathcal{A}$ have **unbounded** computational power,
- But, $\alpha$ has bitlength at most $S$, and $\mathcal{A}$ can make at most $T$ evaluations of $f$.

# Applications

Scenarios

- The NSA wants to break a widely used cryptographic function, such as AES-128

# Applications

Scenarios

- The NSA wants to break a widely used cryptographic function, such as AES-128
- Hackers want to recover passwords from a stolen database of password hashes (Rainbow Tables)

# Applications

Scenarios

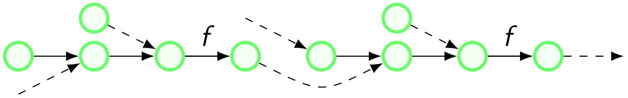
- The NSA wants to break a widely used cryptographic function, such as AES-128
- Hackers want to recover passwords from a stolen database of password hashes (Rainbow Tables)
- Theoretical computer scientists want better algorithms for 3-SUM [GGH+20],     , multiparty pointer jumping [CK19],   systematic substring search [CK19], ...

# Beyond permutations

- Preprocessing stores the endpoints of disjoint paths.

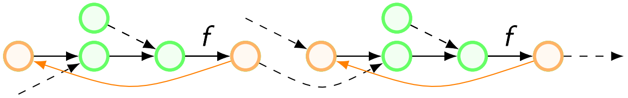
# Beyond permutations

- Preprocessing stores the endpoints of disjoint paths.



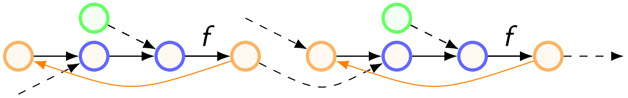
# Beyond permutations

- Preprocessing stores the endpoints of disjoint paths.



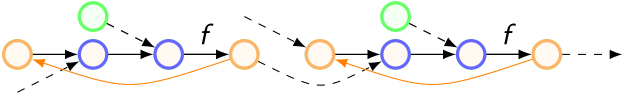
# Beyond permutations

- Preprocessing stores the endpoints of disjoint paths.



# Beyond permutations

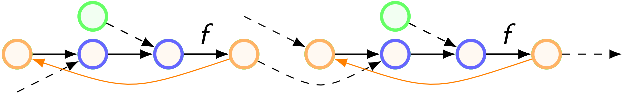
- Preprocessing stores the endpoints of disjoint paths.




- No longer possible to cover the entire range.

# Beyond permutations

- Preprocessing stores the endpoints of disjoint paths.

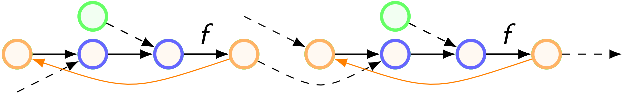



- No longer possible to cover the entire range.

- But, can still cover a small fraction with disjoint paths.

# Beyond permutations

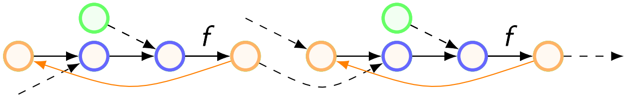
- Preprocessing stores the endpoints of disjoint paths.




- No longer possible to cover the entire range.
- But, can still cover a small fraction with disjoint paths.
- To boost the coverage, observe that inverting $g \circ f$ on $g(y)$ is enough to invert $f$ on $y$.

# Beyond permutations

- Preprocessing stores the endpoints of disjoint paths.




- No longer possible to cover the entire range.

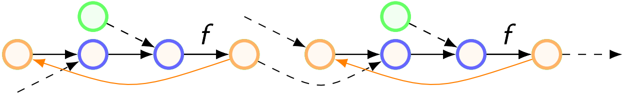
- But, can still cover a small fraction with disjoint paths.

- To boost the coverage, observe that inverting $g \circ f$ on $g(y)$ is enough to invert $f$ on $y$.

- So, can repeatedly apply the basic scheme to many compositions $g_i \circ f$, for suitably chosen "rerandomization" functions $g_i$.

# Beyond permutations

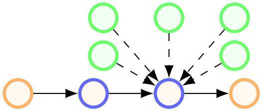
- Preprocessing stores the endpoints of disjoint paths.




- No longer possible to cover the entire range.
- But, can still cover a small fraction with disjoint paths.
- To boost the coverage, observe that inverting $g \circ f$ on $g(y)$ is enough to invert $f$ on $y$.
- So, can repeatedly apply the basic scheme to many compositions $g_i \circ f$, for suitably chosen “rerandomization” functions $g_i$.
- For *random* functions, Hellman showed (heuristically) this can be made to work. 

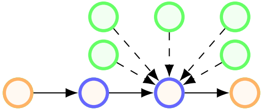
# Beyond permutations

- Hellman's argument fails for arbitrary functions.



# Beyond permutations

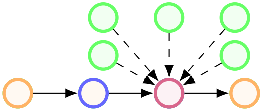
- Hellman's argument fails for arbitrary functions.



- There may be points with very many inverses ("junction points"), which cause paths to overlap quickly.

# Beyond permutations

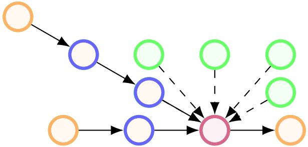
- Hellman's argument fails for arbitrary functions.



- There may be points with very many inverses ("junction points"), which cause paths to overlap quickly.

# Beyond permutations

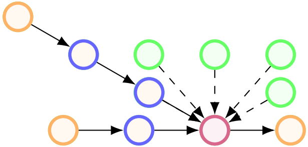
- Hellman's argument fails for arbitrary functions.



- There may be points with very many inverses ("junction points"), which cause paths to overlap quickly.

# Beyond permutations

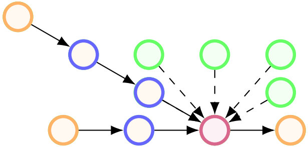
- Hellman's argument fails for arbitrary functions.



- There may be points with very many inverses ("junction points"), which cause paths to overlap quickly.
- Among other things, overlapping paths are redundant, causing massive inefficiency!
- Fiat and Naor deal with this by storing $\alpha = (\alpha', L)$, where $L$ contains junction points along with their inverses.

# Beyond permutations

- Hellman's argument fails for arbitrary functions.



- There may be points with very many inverses ("junction points"), which cause paths to overlap quickly.
- Among other things, overlapping paths are redundant, causing massive inefficiency!
- Fiat and Naor deal with this by storing $\alpha = (\alpha', L)$, where $L$ contains junction points along with their inverses.
- Intuitively, $\alpha'$ is the data structure for a *restriction* of $f$ that avoids the junction points in $L$.
- More precisely, the "rerandomization" functions are sampled using rejection sampling so that their range is $[N] - L$.

# The story so far

| Result | | Applies To | Tradeoff | Key Point |
|---|---|---|---|---|
| Hellman  | 1980 | permutations | $T \leq O((N \log N)/S)$ | $S = T \lesssim \sqrt{N}$ |

# The story so far

| Result | Applies To | Tradeoff | Key Point |
|---|---|---|---|
| Hellman 1980  | permutations | $T \leq O((N \log N)/S)$ | $S = T \lesssim \sqrt{N}$ |
| Yao 1990  | permutations | $T \geq \Omega((N \log N)/S)$ | $S = T \gtrsim \sqrt{N}$ |

# The story so far

| Result | Applies To | Tradeoff | Key Point |
|---|---|---|---|
| Hellman 1980  | permutations | $T \leq O((N \log N)/S)$ | $S = T \lesssim \sqrt{N}$ |
| Yao 1990  | permutations | $T \geq \Omega((N \log N)/S)$ | $S = T \gtrsim \sqrt{N}$ |
| Hellman 1980  | random $f$ | $T \leq \tilde{O}(N^2/S^2)$ | $S = T \lesssim N^{2/3}$ |

# The story so far

| Result | Applies To | Tradeoff | Key Point |
|---|---|---|---|
| Hellman 1980  | permutations | $T \leq O((N \log N)/S)$ | $S = T \lesssim \sqrt{N}$ |
| Yao 1990  | permutations | $T \geq \Omega((N \log N)/S)$ | $S = T \gtrsim \sqrt{N}$ |
| Hellman 1980  | random $f$ | $T \leq \tilde{O}(N^2/S^2)$ | $S = T \lesssim N^{2/3}$ |
| Fiat-Naor 1991   | all functions | $T \leq \tilde{O}(N^3/S^3)$ | $S = T \lesssim N^{3/4}$ |

# The story so far

| Result | Applies To | Tradeoff | Key Point |
|---|---|---|---|
| Hellman 1980  | permutations | $T \leq O((N \log N)/S)$ | $S = T \lesssim \sqrt{N}$ |
| Yao 1990  | permutations | $T \geq \Omega((N \log N)/S)$ | $S = T \gtrsim \sqrt{N}$ |
| Hellman 1980  | random $f$ | $T \leq \tilde{O}(N^2/S^2)$ | $S = T \lesssim N^{2/3}$ |
| Fiat-Naor 1991   | all functions | $T \leq \tilde{O}(N^3/S^3)$ | $S = T \lesssim N^{3/4}$ |

- Q: Can we improve Fiat-Naor? Can we improve Yao's lower bound?

# The story so far

| Result | Applies To | Tradeoff | Key Point |
|---|---|---|---|
| Hellman 1980  | permutations | $T \leq O((N \log N)/S)$ | $S = T \lesssim \sqrt{N}$ |
| Yao 1990  | permutations | $T \geq \Omega((N \log N)/S)$ | $S = T \gtrsim \sqrt{N}$ |
| Hellman 1980  | random $f$ | $T \leq \tilde{O}(N^2/S^2)$ | $S = T \lesssim N^{2/3}$ |
| Fiat-Naor 1991   | all functions | $T \leq \tilde{O}(N^3/S^3)$ | $S = T \lesssim N^{3/4}$ |

- Q: Can we improve Fiat-Naor? Can we improve Yao's lower bound?
- A: Sort of and sort of!

# Result #1: A Simple Improvement to Fiat-Naor

- Recall that for $f$ arbitrary, Hellman's algorithm is tripped up by points with many inverses ("junction points").

# Result #1: A Simple Improvement to Fiat-Naor

- Recall that for $f$ arbitrary, Hellman's algorithm is tripped up by points with many inverses ("junction points").
- Fiat and Naor's idea is to (implicitly) work with a *restriction* of $f$ that leaves out the junction points.

# Result #1: A Simple Improvement to Fiat-Naor

- Recall that for $f$ arbitrary, Hellman's algorithm is tripped up by points with many inverses ("junction points").
- Fiat and Naor's idea is to (implicitly) work with a *restriction* of $f$ that leaves out the junction points.
- Preprocessing conveys the restriction to online via a list $L$ of junction points (along with their inverses).

# Result #1: A Simple Improvement to Fiat-Naor

- Recall that for $f$ arbitrary, Hellman's algorithm is tripped up by points with many inverses ("junction points").
- Fiat and Naor's idea is to (implicitly) work with a *restriction* of $f$ that leaves out the junction points.
- Preprocessing conveys the restriction to online via a list $L$ of junction points (along with their inverses).
- We observe that the tradeoff $T \lesssim N^3/S^3$ comes from

$$T \lesssim \frac{1}{|L|} \cdot \frac{N^3}{S^2}.$$

# Result #1: A Simple Improvement to Fiat-Naor

- Recall that for $f$ arbitrary, Hellman's algorithm is tripped up by points with many inverses ("junction points").
- Fiat and Naor's idea is to (implicitly) work with a *restriction* of $f$ that leaves out the junction points.
- Preprocessing conveys the restriction to online via a list $L$ of junction points (along with their inverses).
- We observe that the tradeoff $T \lesssim N^3/S^3$ comes from

$$T \lesssim \frac{1}{|L|} \cdot \frac{N^3}{S^2}.$$

- So we'd love for $L$ to be longer, but it needs to fit in $\alpha \in \{0,1\}^S$.

# Result #1: A Simple Improvement to Fiat-Naor

- Recall that for $f$ arbitrary, Hellman's algorithm is tripped up by points with many inverses ("junction points").
- Fiat and Naor's idea is to (implicitly) work with a *restriction* of $f$ that leaves out the junction points.
- Preprocessing conveys the restriction to online via a list $L$ of junction points (along with their inverses).
- We observe that the tradeoff $T \lesssim N^3/S^3$ comes from

$$T \lesssim \frac{1}{|L|} \cdot \frac{N^3}{S^2}.$$

- So we'd love for $L$ to be longer, but it needs to fit in $\alpha \in \{0,1\}^S$.
- Or does it?

# Result #1: A Simple Improvement to Fiat-Naor

- Fiat and Naor's list $L$ actually consists of images $f(x_i)$ of random points $x_i \sim [N]$.

# Result #1: A Simple Improvement to Fiat-Naor

- Fiat and Naor's list $L$ actually consists of images $f(x_i)$ of random points $x_i \sim [N]$.
- Our idea: Instead of reading $L$ from $\alpha$, $\mathcal{A}$ recovers $L$ by evaluating $f$ on the random points $x_i$.

# Result #1: A Simple Improvement to Fiat-Naor

- Fiat and Naor's list $L$ actually consists of images $f(x_i)$ of random points $x_i \sim [N]$.
- Our idea: Instead of reading $L$ from $\alpha$, $\mathcal{A}$ recovers $L$ by evaluating $f$ on the random points $x_i$.
- This allows $|L| \simeq T$, so we can get $T \lesssim N^3/(S^2 T)$, or

$$T \lesssim N^{3/2}/S.$$

# Result #1: A Simple Improvement to Fiat-Naor

- Fiat and Naor's list $L$ actually consists of images $f(x_i)$ of random points $x_i \sim [N]$.
- Our idea: Instead of reading $L$ from $\alpha$, $\mathcal{A}$ recovers $L$ by evaluating $f$ on the random points $x_i$.
- This allows $|L| \simeq T$, so we can get $T \lesssim N^3/(S^2 T)$, or

$$T \lesssim N^{3/2}/S.$$

- But I've cheated here...

# Result #1: A Simple Improvement to Fiat-Naor

- Fiat and Naor's list $L$ actually consists of images $f(x_i)$ of random points $x_i \sim [N]$.
- Our idea: Instead of reading $L$ from $\alpha$, $\mathcal{A}$ recovers $L$ by evaluating $f$ on the random points $x_i$.
- This allows $|L| \simeq T$, so we can get $T \lesssim N^3/(S^2 T)$, or

$$T \lesssim N^{3/2}/S.$$

- But I've cheated here...
- How do $\mathcal{A}$ and $\mathcal{P}$ agree on the same list of random values $x_i$?

# Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality. Here, that means shared random $x_i$.

# Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality. Here, that means shared random $x_i$.
- More precisely, shared randomness costs an additive $O(\log N)$ in $S$ and a factor 2 increase in failure probability.

# Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality. Here, that means shared random $x_i$.
- More precisely, shared randomness costs an additive $O(\log N)$ in $S$ and a factor 2 increase in failure probability.
- The proof adapts *Newman's lemma* [New91]  from communication complexity.

# Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality. Here, that means shared random $x_i$.
- More precisely, shared randomness costs an additive $O(\log N)$ in $S$ and a factor 2 increase in failure probability.
- The proof adapts *Newman's lemma* [New91]  from communication complexity.
- The idea: rather than storing the random bits in full, store a random *index* into a fixed list of strings.

# Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality. Here, that means shared random $x_i$.
- More precisely, shared randomness costs an additive $O(\log N)$ in $S$ and a factor 2 increase in failure probability.
- The proof adapts *Newman's lemma* [New91]  from communication complexity.
- The idea: rather than storing the random bits in full, store a random *index* into a fixed list of strings.
- If the number $\Sigma = O(N)$ of fixed strings is large enough, this will be almost as good as fully random.

# Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality. Here, that means shared random $x_i$.
- More precisely, shared randomness costs an additive $O(\log N)$ in $S$ and a factor 2 increase in failure probability.
- The proof adapts *Newman's lemma* [New91]  from communication complexity.
- The idea: rather than storing the random bits in full, store a random *index* into a fixed list of strings.
- If the number $\Sigma = O(N)$ of fixed strings is large enough, this will be almost as good as fully random.
- This approach leans fairly heavily on non-uniformity. But in practice, can just instantiate a random oracle.

# In Context

| Result | | Applies To | Tradeoff | Key Point |
|---|---|---|---|---|
| Hellman  | 1980 | permutations | $T \leq O((N \log N)/S)$ | $S = T \lesssim \sqrt{N}$ |
| Yao 1990  | | permutations | $T \geq \Omega((N \log N)/S)$ | $S = T \gtrsim \sqrt{N}$ |
| Hellman  | 1980 | random $f$ | $T \leq \tilde{O}(N^2/S^2)$ | $S = T \lesssim N^{2/3}$ |
| Fiat-Naor   | 1991 | all functions | $T \leq \tilde{O}(N^3/S^3)$ | $S = T \lesssim N^{3/4}$ |

# In Context

| Result | | Applies To | Tradeoff | Key Point |
|---|---|---|---|---|
| Hellman  | 1980 | permutations | $T \leq O((N \log N)/S)$ | $S = T \lesssim \sqrt{N}$ |
| Yao 1990  | | permutations | $T \geq \Omega((N \log N)/S)$ | $S = T \gtrsim \sqrt{N}$ |
| Hellman  | 1980 | random $f$ | $T \leq \tilde{O}(N^2/S^2)$ | $S = T \lesssim N^{2/3}$ |
| Fiat-Naor   | 1991 | all functions | $T \leq \tilde{O}(N^3/S^3)$ | $S = T \lesssim N^{3/4}$ |
| This work | | all functions | $T \leq \tilde{O}(N^{3/2}/S)$ | $S = T \lesssim N^{3/4}$ |

# In Context

| Result | | Applies To | Tradeoff | Key Point |
|---|---|---|---|---|
| Hellman  | 1980 | permutations | $T \leq O((N \log N)/S)$ | $S = T \lesssim \sqrt{N}$ |
| Yao 1990  | | permutations | $T \geq \Omega((N \log N)/S)$ | $S = T \gtrsim \sqrt{N}$ |
| Hellman  | 1980 | random $f$ | $T \leq \tilde{O}(N^2/S^2)$ | $S = T \lesssim N^{2/3}$ |
| Fiat-Naor   | 1991 | all functions | $T \leq \tilde{O}(N^3/S^3)$ | $S = T \lesssim N^{3/4}$ |
| This work | | all functions | $T \leq \tilde{O}(N^{3/2}/S)$<br>$T \lesssim N^3/(S^2 T)$ | $S = T \lesssim N^{3/4}$ |

# Non-Adaptive Algorithms

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.

# Non-Adaptive Algorithms

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.
- Corrigan-Gibbs and Kogan   [CK19]: any small improvement $\implies$ new lower bounds in circuit complexity.

# Non-Adaptive Algorithms

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.
- Corrigan-Gibbs and Kogan   [CK19]: any small improvement $\implies$ new lower bounds in circuit complexity.
- Even improving Yao's bound just for *non-adaptive* algorithms would do it!

# Non-Adaptive Algorithms

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.
- Corrigan-Gibbs and Kogan   [CK19]: any small improvement $\implies$ new lower bounds in circuit complexity.
- Even improving Yao's bound just for *non-adaptive* algorithms would do it!
- $\mathcal{A}$ is non-adaptive if its evaluation points $x_1, \ldots, x_T$ are chosen up front, before any evaluations of $f$ are seen.

# Non-Adaptive Algorithms

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.
- Corrigan-Gibbs and Kogan   [CK19]: any small improvement $\implies$ new lower bounds in circuit complexity.
- Even improving Yao's bound just for *non-adaptive* algorithms would do it!
- $\mathcal{A}$ is non-adaptive if its evaluation points $x_1, \ldots, x_T$ are chosen up front, before any evaluations of $f$ are seen.
- Non-adaptive algorithms seem very weak. Hellman's algorithm is *very* adaptive.

# Non-Adaptive Algorithms

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.
- Corrigan-Gibbs and Kogan   [CK19]: any small improvement $\implies$ new lower bounds in circuit complexity.
- Even improving Yao's bound just for *non-adaptive* algorithms would do it!
- $\mathcal{A}$ is non-adaptive if its evaluation points $x_1, \ldots, x_T$ are chosen up front, before any evaluations of $f$ are seen.
- Non-adaptive algorithms seem very weak. Hellman's algorithm is *very* adaptive.
- Corrigan-Gibbs and Kogan speculated that there is no non-adaptive algorithm with

$$S = o(N \log N) \text{ and } T = o(N).$$

# Result #2: A Non-Adaptive Algorithm

- Corrigan-Gibbs and Kogan speculated that there is no non-adaptive algorithm with

$$S = o(N \log N) \text{ and } T = o(N).$$

- We show that there IS such an algorithm, that (barely!) outperforms the trivial inverter.

# Result #2: A Non-Adaptive Algorithm

- Corrigan-Gibbs and Kogan speculated that there is no non-adaptive algorithm with

$$S = o(N \log N) \text{ and } T = o(N).$$

- We show that there IS such an algorithm, that (barely!) outperforms the trivial inverter.
- We achieve, for example,

$$S = O(N \log \log N), T = O(N/\log^c N)$$

for any constant $c$.

# Result #2: A Non-Adaptive Algorithm

- Corrigan-Gibbs and Kogan speculated that there is no non-adaptive algorithm with

$$S = o(N \log N) \text{ and } T = o(N).$$

- We show that there IS such an algorithm, that (barely!) outperforms the trivial inverter.
- We achieve, for example,

$$S = O(N \log \log N), T = O(N/ \log^c N)$$

  for any constant $c$.

- Thus non-adaptive algorithms are not the barren, lifeless desert previously expected...

## The Algorithm

- Try $T = 0$, for starters.

# The Algorithm

- Try $T = 0$, for starters.
- Why not $S = \log N$, so that $\alpha \in [N]$?

# The Algorithm

- Try $T = 0$, for starters.
- Why not $S = \log N$, so that $\alpha \in [N]$?
- How many $y$'s can we invert? (What's trivial here?)

# The Algorithm

- Try $T = 0$, for starters.
- Why not $S = \log N$, so that $\alpha \in [N]$?
- How many $y$'s can we invert? (What's trivial here?)
- $\mathcal{A}(\alpha, y)$ is just some (possibly random) function...

# The Algorithm

- Try $T = 0$, for starters.
- Why not $S = \log N$, so that $\alpha \in [N]$?
- How many $y$'s can we invert? (What's trivial here?)
- $\mathcal{A}(\alpha, y)$ is just some (possibly random) function...
- Try $\mathcal{A}(\alpha, y) = \alpha + g(y) \pmod{N}$ where $g : [N] \to [N]$ is random.

# The Algorithm

- Try $T = 0$, for starters.
- Why not $S = \log N$, so that $\alpha \in [N]$?
- How many $y$'s can we invert? (What's trivial here?)
- $\mathcal{A}(\alpha, y)$ is just some (possibly random) function...
- Try $\mathcal{A}(\alpha, y) = \alpha + g(y) \pmod{N}$ where $g : [N] \to [N]$ is random.
- For simplicity, assume $f$ is a permutation. Then

$$\begin{aligned} y \text{ inverted} &\Leftrightarrow \alpha + g(y) = f^{-1}(y) \\ &\Leftrightarrow \alpha = \alpha_y := f^{-1}(y) - g(y) \end{aligned}$$

# The Algorithm

- Try $T = 0$, for starters.
- Why not $S = \log N$, so that $\alpha \in [N]$?
- How many $y$'s can we invert? (What's trivial here?)
- $\mathcal{A}(\alpha, y)$ is just some (possibly random) function...
- Try $\mathcal{A}(\alpha, y) = \alpha + g(y) \pmod N$ where $g : [N] \to [N]$ is random.
- For simplicity, assume $f$ is a permutation. Then

$$\begin{aligned} y \text{ inverted} &\Leftrightarrow \alpha + g(y) = f^{-1}(y) \\ &\Leftrightarrow \alpha = \alpha_y := f^{-1}(y) - g(y) \end{aligned}$$

- $\implies$ preprocessing returns $\alpha$ maximizing

$$|\{y \in [N] : \alpha = \alpha_y\}|.$$

# The Algorithm

- Recall $\alpha \in [N]$ inverts all $y \in [N]$ with

$$\alpha = \alpha_y := f^{-1}(y) - g(y).$$

# The Algorithm

- Recall $\alpha \in [N]$ inverts all $y \in [N]$ with

$$\alpha = \alpha_y := f^{-1}(y) - g(y).$$

- Key point: $g$ is random $\implies$ $\alpha_y$'s are uniform and independent.

# The Algorithm

- Recall $\alpha \in [N]$ inverts all $y \in [N]$ with
$$\alpha = \alpha_y := f^{-1}(y) - g(y).$$

- Key point: $g$ is random $\implies$ $\alpha_y$'s are uniform and independent.
- Balls and bins! With high probability, $\Omega(\log N / \log \log N)$ balls in the bin with most balls.

# The Algorithm

- Recall $\alpha \in [N]$ inverts all $y \in [N]$ with
$$\alpha = \alpha_y := f^{-1}(y) - g(y).$$
- Key point: $g$ is random $\implies \alpha_y$'s are uniform and independent.
- Balls and bins! With high probability, $\Omega(\log N / \log\log N)$ balls in the bin with most balls.
- $\implies \max_\alpha |\{y \in [N] : \alpha_y = \alpha\}| \geq \Omega(\log N / \log\log N)$.

# The Algorithm

- What if we allow ourselves to store another element $\alpha' \in [N]$?

# The Algorithm

- What if we allow ourselves to store another element $\alpha' \in [N]$?
- First remove the "balls" $\{y \in [N] : \alpha_y = \alpha\}$ from consideration.

# The Algorithm

- What if we allow ourselves to store another element $\alpha' \in [N]$?
- First remove the "balls" $\{y \in [N] : \alpha_y = \alpha\}$ from consideration.
- Toss the remaining balls into the bins again, using a fresh random function $g'$, and store new bin $\alpha'$ with most balls.

# The Algorithm

- What if we allow ourselves to store another element $\alpha' \in [N]$?
- First remove the "balls" $\{y \in [N] : \alpha_y = \alpha\}$ from consideration.
- Toss the remaining balls into the bins again, using a fresh random function $g'$, and store new bin $\alpha'$ with most balls.
- Online computes the two candidates $x = g(y) + \alpha$, $x' = g'(y) + \alpha'$.

# The Algorithm

- What if we allow ourselves to store another element $\alpha' \in [N]$?
- First remove the "balls" $\{y \in [N] : \alpha_y = \alpha\}$ from consideration.
- Toss the remaining balls into the bins again, using a fresh random function $g'$, and store new bin $\alpha'$ with most balls.
- Online computes the two candidates $x = g(y) + \alpha$, $x' = g'(y) + \alpha'$.
- It evaluates $f$ on each to check if $f(x) = y$ or $f(x') = y$, and outputs whichever passes the check (if any). Non-adaptive!

# The Algorithm

- What if we allow ourselves to store another element $\alpha' \in [N]$?
- First remove the "balls" $\{y \in [N] : \alpha_y = \alpha\}$ from consideration.
- Toss the remaining balls into the bins again, using a fresh random function $g'$, and store new bin $\alpha'$ with most balls.
- Online computes the two candidates $x = g(y) + \alpha$, $x' = g'(y) + \alpha'$.
- It evaluates $f$ on each to check if $f(x) = y$ or $f(x') = y$, and outputs whichever passes the check (if any). Non-adaptive!
- For the full algorithm, instead of repeating twice, we repeat $r = O(N/(\log N/\log\log N))$ times.

# The Algorithm

- What if we allow ourselves to store another element $\alpha' \in [N]$?
- First remove the "balls" $\{y \in [N] : \alpha_y = \alpha\}$ from consideration.
- Toss the remaining balls into the bins again, using a fresh random function $g'$, and store new bin $\alpha'$ with most balls.
- Online computes the two candidates $x = g(y) + \alpha$, $x' = g'(y) + \alpha'$.
- It evaluates $f$ on each to check if $f(x) = y$ or $f(x') = y$, and outputs whichever passes the check (if any). Non-adaptive!
- For the full algorithm, instead of repeating twice, we repeat $r = O(N/(\log N/\log\log N))$ times.
- $S = O(r \log N) = O(N \log\log N)$, $T = O(r) = O(N \log\log N/\log N)$.

# The Algorithm

- Generalizing to arbitrary functions is easy: just assign a unique inverse $x_y$ to each $y \in f([N])$ and define $\alpha_y = x_y - g(y)$.

# The Algorithm

- Generalizing to arbitrary functions is easy: just assign a unique inverse $x_y$ to each $y \in f([N])$ and define $\alpha_y = x_y - g(y)$.
- By partitioning the range and letting each $\alpha$ specify multiple candidates, we can get $S = O(N \log(N/T))$.

# The Algorithm

- Generalizing to arbitrary functions is easy: just assign a unique inverse $x_y$ to each $y \in f([N])$ and define $\alpha_y = x_y - g(y)$.
- By partitioning the range and letting each $\alpha$ specify multiple candidates, we can get $S = O(N \log(N/T))$.
- Is this tradeoff the "right answer"?

# The Algorithm

- Generalizing to arbitrary functions is easy: just assign a unique inverse $x_y$ to each $y \in f([N])$ and define $\alpha_y = x_y - g(y)$.
- By partitioning the range and letting each $\alpha$ specify multiple candidates, we can get $S = O(N \log(N/T))$.
- Is this tradeoff the "right answer"?
- Our algorithm is a *guess-and-check algorithm*—it's non-adaptive, and it returns one of the points $x_1, \ldots, x_T$ that it queries.

# The Algorithm

- Generalizing to arbitrary functions is easy: just assign a unique inverse $x_y$ to each $y \in f([N])$ and define $\alpha_y = x_y - g(y)$.
- By partitioning the range and letting each $\alpha$ specify multiple candidates, we can get $S = O(N \log(N/T))$.
- Is this tradeoff the "right answer"?
- Our algorithm is a *guess-and-check algorithm*—it's non-adaptive, and it returns one of the points $x_1, \ldots, x_T$ that it queries.
- That is, it can be thought of returning $T$ candidate inverses $x_1, \ldots, x_T$, *without querying $f$ at all*.

# The Algorithm

- Generalizing to arbitrary functions is easy: just assign a unique inverse $x_y$ to each $y \in f([N])$ and define $\alpha_y = x_y - g(y)$.
- By partitioning the range and letting each $\alpha$ specify multiple candidates, we can get $S = O(N \log(N/T))$.
- Is this tradeoff the "right answer"?
- Our algorithm is a *guess-and-check algorithm*—it's non-adaptive, and it returns one of the points $x_1, \ldots, x_T$ that it queries.
- That is, it can be thought of returning $T$ candidate inverses $x_1, \ldots, x_T$, *without querying $f$ at all*.
- We show that all *guess-and-check* algorithms satisfy the matching lower bound $S = \Omega(N \log(N/T))$.

# Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters $S, T$. We use a compression argument.

# Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters $S$, $T$. We use a compression argument.
- Theorem: Can encode any permutation $f$ using $S + N \log T$ bits.

# Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters $S, T$. We use a compression argument.
- Theorem: Can encode any permutation $f$ using $S + N \log T$ bits.

$$\begin{aligned} &\implies S + N \log T = \Omega(N \log N) \\ &\implies S = \Omega(N \log(N/T)). \end{aligned}$$

# Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters $S, T$. We use a compression argument.
- Theorem: Can encode any permutation $f$ using $S + N \log T$ bits.

$$\begin{aligned} &\implies S + N \log T = \Omega(N \log N) \\ &\implies S = \Omega(N \log(N/T)). \end{aligned}$$

- Proof:
  - Encoder computes $\alpha \leftarrow \mathcal{P}(f)$.

# Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters $S$, $T$. We use a compression argument.
- Theorem: Can encode any permutation $f$ using $S + N \log T$ bits.

$$\implies S + N \log T = \Omega(N \log N)$$
$$\implies S = \Omega(N \log(N/T)).$$

- Proof:
  - Encoder computes $\alpha \leftarrow \mathcal{P}(f)$.
  - For each $y \in [N]$, encoder runs $\mathcal{A}(\alpha, y)$ and receives $x_1, \ldots, x_T$. It writes down the $i_y \in [T]$ that satisfies $f(x_{i_y}) = y$.

# Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters $S, T$. We use a compression argument.
- Theorem: Can encode any permutation $f$ using $S + N \log T$ bits.

$$\implies S + N \log T = \Omega(N \log N)$$
$$\implies S = \Omega(N \log(N/T)).$$

- Proof:
  - Encoder computes $\alpha \leftarrow \mathcal{P}(f)$.
  - For each $y \in [N]$, encoder runs $\mathcal{A}(\alpha, y)$ and receives $x_1, \ldots, x_T$. It writes down the $i_y \in [T]$ that satisfies $f(x_{i_y}) = y$.
  - Encoding is $(\alpha, i_1, \ldots, i_N)$.

# Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters $S, T$. We use a compression argument.
- Theorem: Can encode any permutation $f$ using $S + N \log T$ bits.

$$\implies S + N \log T = \Omega(N \log N)$$
$$\implies S = \Omega(N \log(N/T)).$$

- Proof:
  - Encoder computes $\alpha \leftarrow \mathcal{P}(f)$.
  - For each $y \in [N]$, encoder runs $\mathcal{A}(\alpha, y)$ and receives $x_1, \ldots, x_T$. It writes down the $i_y \in [T]$ that satisfies $f(x_{i_y}) = y$.
  - Encoding is $(\alpha, i_1, \ldots, i_N)$.
  - For each $y$, decoder again runs $\mathcal{A}(\alpha, y)$ and receives $x_1, \ldots, x_T$. It sets $f^{-1}(y) = x_{i_y}$.

# Function Inversion Revisited: Our Results

- Improving Fiat-Naor: a

$$T \lesssim N^{3/2}/S$$

algorithm for inverting arbitrary functions.

# Function Inversion Revisited: Our Results

- Improving Fiat-Naor: a

$$T \lesssim N^{3/2}/S$$

algorithm for inverting arbitrary functions.

- The first nontrivial non-adaptive function inversion algorithm.

# Function Inversion Revisited: Our Results

- Improving Fiat-Naor: a

$$T \lesssim N^{3/2}/S$$

  algorithm for inverting arbitrary functions.

- The first nontrivial non-adaptive function inversion algorithm.
  - Best possible among *guess-and-check* algorithms.

# Function Inversion Revisited: Our Results

- Improving Fiat-Naor: a

$$T \lesssim N^{3/2}/S$$

  algorithm for inverting arbitrary functions.

- The first nontrivial non-adaptive function inversion algorithm.
  - Best possible among *guess-and-check* algorithms.

- Shared randomness is essentially free

# Function Inversion Revisited: Our Results

- Improving Fiat-Naor: a

$$T \lesssim N^{3/2}/S$$

  algorithm for inverting arbitrary functions.
- The first nontrivial non-adaptive function inversion algorithm.
  - Best possible among *guess-and-check* algorithms.
- Shared randomness is essentially free
- Not in this talk:
  - Search-to-decision reductions

# Function Inversion Revisited: Our Results

- Improving Fiat-Naor: a

$$T \lesssim N^{3/2}/S$$

  algorithm for inverting arbitrary functions.
- The first nontrivial non-adaptive function inversion algorithm.
  - Best possible among *guess-and-check* algorithms.
- Shared randomness is essentially free
- Not in this talk:
  - Search-to-decision reductions
  - A larger codomain does not make the problem harder (us and [CK19])  

# Open Problems

- Only one real open problem–

# Open Problems

- Only one real open problem–**close the gap!**

# Open Problems

- Only one real open problem–**close the gap!**
- Improve Yao's lower bound against (general) non-adaptive algorithms?

# Open Problems

- Only one real open problem–**close the gap!**
- Improve Yao's lower bound against (general) non-adaptive algorithms?
- $S^2T = N^2$ algorithm for worst-case function inversion?

# Open Problems

- Only one real open problem–**close the gap!**
- Improve Yao's lower bound against (general) non-adaptive algorithms?
- $S^2 T = N^2$ algorithm for worst-case function inversion?
- Improved algorithm for injective functions?

# Thank you!

I'm happy to take additional questions offline. You can ping me at `sp2473@cornell.edu`.

# References I


Henry Corrigan-Gibbs and Dmitry Kogan.
The function-inversion problem: Barriers and opportunities.
In *TCC*, 2019.

Alexander Golovnev, Siyao Guo, Thibaut Horel, Sunoo Park, and Vinod Vaikuntanathan.
Data structures meet cryptography: 3SUM with preprocessing.
In *STOC*, 2020.

Alexander Golovnev, Siyao Guo, Spencer Peters, and Noah Stephens-Davidowitz.
Revisiting time-space tradeoffs for function inversion.
Available at `https://eccc.weizmann.ac.il/report/2022/145/`, 2022.

# References II


Martin Hellman.
A cryptanalytic time-memory trade-off.
*IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

Ilan Newman.
Private vs. common random bits in communication complexity.
*Information processing letters*, 39(2):67–71, 1991.