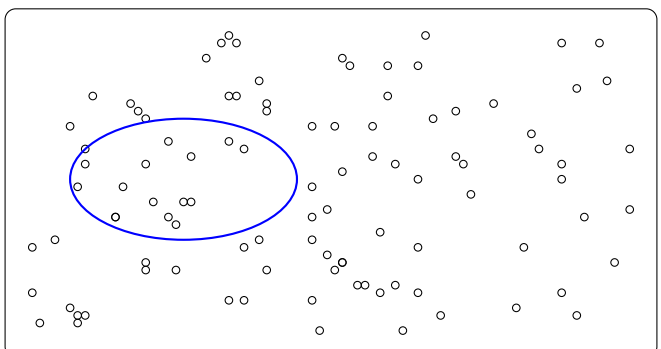
Weight reduction in distributed protocols: new algorithms and analysis

Tolik Zinovyev

Boston University

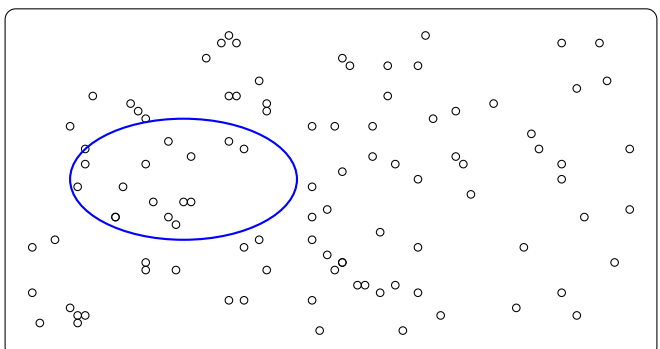
Committee selection

- Have n parties, want to select a small subset and delegate work to them (e.g., in consensus)
- The protocol should remain reliable / secure



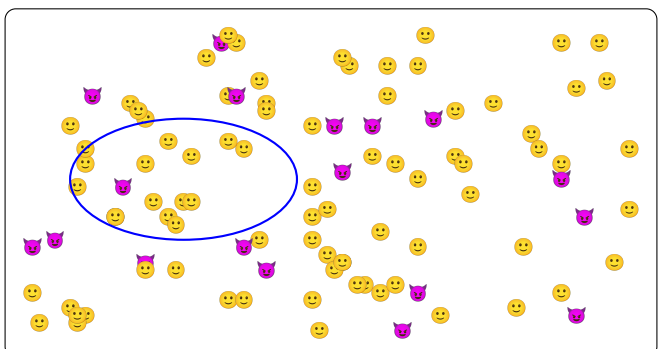
Committee selection

- Have n parties, want to select a small subset and delegate work to them (e.g., in consensus)
- The protocol should remain reliable / secure



Committee selection

- Have n parties, want to select a small subset and delegate work to them (e.g., in consensus)
- The protocol should remain reliable / secure



Committee selection (cont.)

Typical goal:

- 20% of parties are malicious, need to elect a committee with $< \frac{1}{3}$ parties malicious
- some security is lost: $\frac{1}{5} \to \frac{1}{3}$

$$t_i = \begin{cases} 1 \text{ if party } i \text{ is chosen} \\ 0 \text{ otherwise} \end{cases}$$

$A \subseteq [n]$ – adversary set, want

$$\text{for all } A \text{ with } |A| \leq \frac{n}{5} : \quad \Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i\right] \text{ is big}$$

Committee selection (cont.)

Typical goal:

- 20% of parties are malicious, need to elect a committee with $< \frac{1}{3}$ parties malicious
- some security is lost: $\frac{1}{5} \to \frac{1}{3}$

$$t_i = \begin{cases} 1 \text{ if party } i \text{ is chosen} \\ 0 \text{ otherwise} \end{cases}$$

$A \subseteq [n]$ – adversary set, want

$$\text{for all } A \text{ with } |A| \leq \frac{n}{5} : \quad \Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i\right] \text{ is big}$$

Committee selection (cont.)

Typical goal:

- 20% of parties are malicious, need to elect a committee with $< \frac{1}{3}$ parties malicious
- some security is lost: $\frac{1}{5} \to \frac{1}{3}$

$$t_i = \begin{cases} 1 \text{ if party } i \text{ is chosen} \\ 0 \text{ otherwise} \end{cases}$$

$A \subseteq [n]$ – adversary set, want

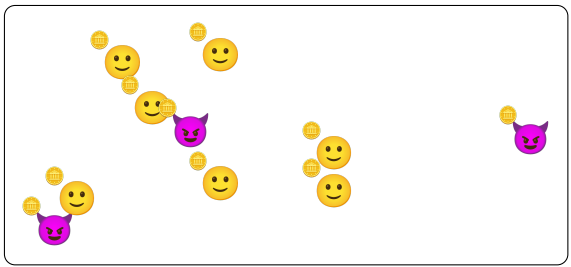
for all A with $|A| \leq \frac{n}{5}$:

$$\Pr\left[\sum_{i \in A} t_i < \frac{1}{3}\sum_{i=1}^{n} t_i\right] \text{ is big}$$

Committee selection (cont.)

Typical solution:

- select each party with probability p:
  $t_i \sim \texttt{Bernoulli}(p)$



- about 20% of selected parties will be malicious; use tail bounds to analyze bad event
- security error $2^{-\lambda}$ requires committee size $\approx 50\lambda$ (for $\frac{1}{5} \to \frac{1}{3}$)

$$t_i = \begin{cases} 1 \text{ if party } i \text{ is chosen} \\ 0 \text{ otherwise} \end{cases}$$

$A \subseteq [n]$ – adversary set, want

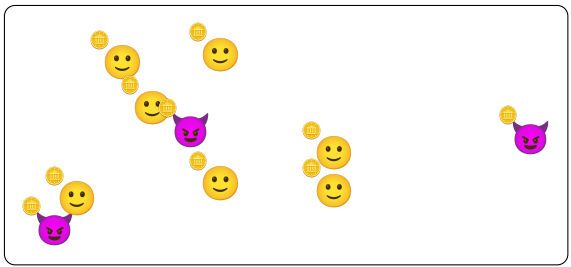
for all A with $|A| \leq \frac{n}{5}$:

$$\Pr\left[\sum_{i \in A} t_i < \frac{1}{3}\sum_{i=1}^{n} t_i\right] \text{ is big}$$

Committee selection (cont.)

Typical solution:

- select each party with probability p: $t_i \sim \texttt{Bernoulli}(p)$



- about 20% of selected parties will be malicious; use tail bounds to analyze bad event
- security error $2^{-\lambda}$ requires committee size $\approx 50\lambda$ (for $\frac{1}{5} \to \frac{1}{3}$)

$$t_i = \begin{cases} 1 \text{ if party } i \text{ is chosen} \\ 0 \text{ otherwise} \end{cases}$$

$A \subseteq [n]$ – adversary set, want

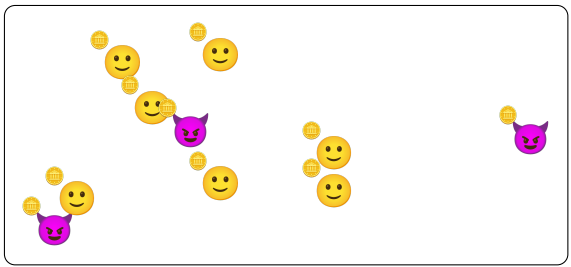
for all A with $|A| \leq \frac{n}{5}$:

$$\Pr\left[\sum_{i \in A} t_i < \frac{1}{3}\sum_{i=1}^{n} t_i\right] \text{ is big}$$

Committee selection (cont.)

Typical solution:

- select each party with probability p: $t_i \sim \texttt{Bernoulli}(p)$



- about 20% of selected parties will be malicious; use tail bounds to analyze bad event
- security error $2^{-\lambda}$ requires committee size $\approx 50\lambda$ (for $\frac{1}{5} \to \frac{1}{3}$)

$$t_i = \begin{cases} 1 \text{ if party } i \text{ is chosen} \\ 0 \text{ otherwise} \end{cases}$$

$A \subseteq [n]$ – adversary set, want

for all A with $|A| \leq \frac{n}{5}$:

$$\Pr\left[\sum_{i \in A} t_i < \frac{1}{3}\sum_{i=1}^{n} t_i\right] \text{ is big}$$

Introducing weights

- the n parties have weights $w_1, w_2, ..., w_n \in \mathbb{Z}_{\geq 0}$
- need to assign new weights $t_1, t_2, ..., t_n \in \mathbb{Z}_{\geq 0}$
  - say "party i gets t_i tickets"
- new problem statement: adversary has weight at most 20%; he must have $< \frac{1}{3}$ tickets

need to sample
$t_1, t_2, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that

for all A with

$$\sum_{i \in A} w_i \leq \frac{1}{5} \sum_{i=1}^{n} w_i \ :$$

$$\Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i\right] \text{ is big}$$

Introducing weights

- the n parties have weights $w_1, w_2, ..., w_n \in \mathbb{Z}_{\geq 0}$
- need to assign new weights $t_1, t_2, ..., t_n \in \mathbb{Z}_{\geq 0}$
  - say "party i gets t_i tickets"
- new problem statement: adversary has weight at most 20%; he must have $< \frac{1}{3}$ tickets

need to sample
$t_1, t_2, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that

for all A with

$$\sum_{i \in A} w_i \leq \frac{1}{5} \sum_{i=1}^{n} w_i \ :$$

$$\Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i\right] \text{ is big}$$

Introducing weights (cont.)

- what Algorand uses
- their solution: select each *unit* of weight with probability p
  - equivalently, $t_i \sim \texttt{Binomial}(w_i, p)$;
- often want security against adaptive corruptions
  - the adversary is allowed to corrupt parties throughout the protocol execution

- Algorand achieves it: does extra work to "hide" t_i

need to sample
$t_1, t_2, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that

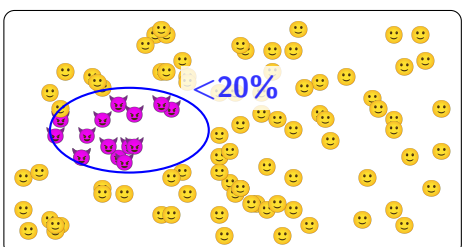
for all A with

$$\sum_{i \in A} w_i \leq \frac{1}{5} \sum_{i=1}^{n} w_i \ :$$

$$\Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i\right] \text{ is big}$$

Introducing weights (cont.)

- what Algorand uses
- their solution: select each *unit* of weight with probability p
  - equivalently, $t_i \sim \texttt{Binomial}(w_i, p)$;
- often want security against adaptive corruptions
  - the adversary is allowed to corrupt parties throughout the protocol execution



- Algorand achieves it: does extra work to "hide" t_i

need to sample
$t_1, t_2, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that

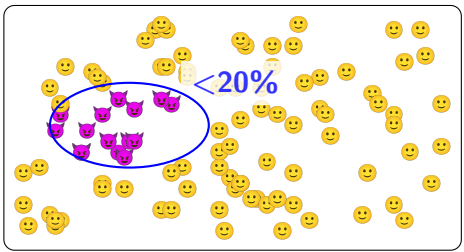
for all A with

$$\sum_{i \in A} w_i \leq \frac{1}{5} \sum_{i=1}^{n} w_i \ :$$

$$\Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i\right] \text{ is big}$$

Introducing weights (cont.)

- what Algorand uses
- their solution: select each *unit* of weight with probability p
  - equivalently, $t_i \sim \texttt{Binomial}(w_i, p)$;
- often want security against adaptive corruptions
  - the adversary is allowed to corrupt parties throughout the protocol execution



- Algorand achieves it: does extra work to "hide" t_i

need to sample
$t_1, t_2, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that

for all A with

$$\sum_{i \in A} w_i \leq \frac{1}{5} \sum_{i=1}^{n} w_i \ :$$

$$\Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i\right] \text{ is big}$$

Improvements in weighted committee selection

- Take weight distribution $(w_1, ..., w_n)$ into account
- Fait Accompli [GKR23]:
  - selects biggest parties deterministically and everyone else randomly as in Algorand
  - big players selected almost certainly anyway
  - randomly sample from less weight $\Longrightarrow$ less variance
  - improves committee size vs. error tradeoff
- Swiper [TF23] + others [BHS23, FMT24, DPTX25]:
  - fully deterministic $\Rightarrow$

$$\text{for all } A \text{ with } \sum_{i \in A} w_i \leq \frac{1}{5} \sum_{i=1}^{n} w_i \ : \ \Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i \right] \text{ is big}$$

  - cons: committee size $\Omega(n)$ in worst case (e.g., $w_1 = w_2 = ... = w_n = 1$)
  - pros: security against adaptive corruptions guaranteed
  - pros: (?) deterministic committees are smaller when security parameter is large
- both show that realistic weight distributions allow small committees

Improvements in weighted committee selection

- Take weight distribution $(w_1, ..., w_n)$ into account
- Fait Accompli [GKR23]:
  - selects biggest parties deterministically and everyone else randomly as in Algorand
  - big players selected almost certainly anyway
  - randomly sample from less weight $\Longrightarrow$ less variance
  - improves committee size vs. error tradeoff
- Swiper [TF23] + others [BHS23, FMT24, DPTX25]:
  - fully deterministic $\Rightarrow$

$$\text{for all } A \text{ with } \sum_{i \in A} w_i \leq \frac{1}{5} \sum_{i=1}^{n} w_i \ : \ \Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i \right] \text{ is big}$$

  - cons: committee size $\Omega(n)$ in worst case (e.g., $w_1 = w_2 = ... = w_n = 1$)
  - pros: security against adaptive corruptions guaranteed
  - pros: (?) deterministic committees are smaller when security parameter is large
- both show that realistic weight distributions allow small committees

Improvements in weighted committee selection

- Take weight distribution $(w_1, ..., w_n)$ into account
- Fait Accompli [GKR23]:
  - selects biggest parties deterministically and everyone else randomly as in Algorand
  - big players selected almost certainly anyway
  - randomly sample from less weight $\Longrightarrow$ less variance
  - improves committee size vs. error tradeoff
- Swiper [TF23] + others [BHS23, FMT24, DPTX25]:
  - fully deterministic $\Rightarrow$

$$\text{for all } A \text{ with } \sum_{i \in A} w_i \leq \frac{1}{5} \sum_{i=1}^{n} w_i \ : \ \Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i \right] \text{ is big}$$

  - cons: committee size $\Omega(n)$ in worst case (e.g., $w_1 = w_2 = ... = w_n = 1$)
  - pros: security against adaptive corruptions guaranteed
  - pros: (?) deterministic committees are smaller when security parameter is large
- both show that realistic weight distributions allow small committees

Improvements in weighted committee selection

- Take weight distribution $(w_1, ..., w_n)$ into account
- Fait Accompli [GKR23]:
  - selects biggest parties deterministically and everyone else randomly as in Algorand
  - big players selected almost certainly anyway
  - randomly sample from less weight $\Longrightarrow$ less variance
  - improves committee size vs. error tradeoff
- Swiper [TF23] + others [BHS23, FMT24, DPTX25]:
  - fully deterministic $\Rightarrow$

$$\text{for all } A \text{ with } \sum_{i \in A} w_i \leq \frac{1}{5} \sum_{i=1}^{n} w_i \ : \ \Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i \right] \text{ is big}$$

  - cons: committee size $\Omega(n)$ in worst case (e.g., $w_1 = w_2 = ... = w_n = 1$)
  - pros: security against adaptive corruptions guaranteed
  - pros: (?) deterministic committees are smaller when security parameter is large
- both show that realistic weight distributions allow small committees

Improvements in weighted committee selection

- Take weight distribution $(w_1, ..., w_n)$ into account
- Fait Accompli [GKR23]:
  - selects biggest parties deterministically and everyone else randomly as in Algorand
  - big players selected almost certainly anyway
  - randomly sample from less weight $\Longrightarrow$ less variance
  - improves committee size vs. error tradeoff
- Swiper [TF23] + others [BHS23, FMT24, DPTX25]:
  - fully deterministic $\Rightarrow$

$$\text{for all } A \text{ with } \sum_{i \in A} w_i \leq \frac{1}{5} \sum_{i=1}^{n} w_i \ : \ \Pr\left[\sum_{i \in A} t_i < \frac{1}{3} \sum_{i=1}^{n} t_i \right] \text{ is big}$$

  - cons: committee size $\Omega(n)$ in worst case (e.g., $w_1 = w_2 = ... = w_n = 1$)
  - pros: security against adaptive corruptions guaranteed
  - pros: (?) deterministic committees are smaller when security parameter is large
- both show that realistic weight distributions allow small committees

Objective function

Minimize the size of the committee $(t_1, t_2, ..., t_n)$: how to quantify?

- option (1) : number of distinct parties on the committee: the number of $i \in [n]$ with $t_i > 0$
  - useful in e.g. Algorand
- option (2) : the total new weight: $\sum_{i=1}^{n} t_i$
  - useful when the protocol scales poorly with the weights (e.g., secret sharing)

Fait Accompli solves (1) and (2); Swiper and **this work** solves (2).

Objective function

Minimize the size of the committee $(t_1, t_2, ..., t_n)$: how to quantify?

- option (1) : number of distinct parties on the committee: the number of $i \in [n]$ with $t_i > 0$
  - useful in e.g. Algorand
- option (2) : the total new weight: $\sum_{i=1}^{n} t_i$
  - useful when the protocol scales poorly with the weights (e.g., secret sharing)

Fait Accompli solves (1) and (2); Swiper and **this work** solves (2).

Objective function

Minimize the size of the committee $(t_1, t_2, ..., t_n)$: how to quantify?

- option (1) : number of distinct parties on the committee:
  the number of $i \in [n]$ with $t_i > 0$
  - useful in e.g. Algorand
- option (2) : the total new weight: $\sum_{i=1}^{n} t_i$
  - useful when the protocol scales poorly with the weights (e.g., secret sharing)

Fait Accompli solves (1) and (2); Swiper and **this work** solves (2).

Objective function

Minimize the size of the committee $(t_1, t_2, ..., t_n)$: how to quantify?

- option (1) : number of distinct parties on the committee: the number of $i \in [n]$ with $t_i > 0$
  - useful in e.g. Algorand
- option (2) : the total new weight: $\sum_{i=1}^{n} t_i$
  - useful when the protocol scales poorly with the weights (e.g., secret sharing)

Fait Accompli solves (1) and (2); Swiper and **this work** solves (2).

Problem statement

Assume $0 < \alpha < \beta < 1$. Given $w_1, ..., w_n \in \mathbb{Z}_{\geq 0}$, find $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that

- minimize $\sum_{i=1}^{n} t_i$
- $\alpha \to \beta$, deterministic:

$$\text{for all } A \text{ with } \sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i \ : \sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i$$

This work extends Swiper (Tonkikh, Freitas '24).
Adopt terminology: party i gets t_i "tickets".

- pure optimization problem
- NP-hard? – unknown

Problem statement

Assume $0 < \alpha < \beta < 1$. Given $w_1, ..., w_n \in \mathbb{Z}_{\geq 0}$, find $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that

- minimize $\sum_{i=1}^{n} t_i$
- $\alpha \to \beta$, deterministic:

$$\text{for all } A \text{ with } \sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i \ : \sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i$$

This work extends Swiper (Tonkikh, Freitas '24).
Adopt terminology: party i gets t_i "tickets".

- pure optimization problem
- NP-hard? – unknown

Problem statement

Assume $0 < \alpha < \beta < 1$. Given $w_1, ..., w_n \in \mathbb{Z}_{\geq 0}$, find $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that

- minimize $\sum_{i=1}^{n} t_i$
- $\alpha \to \beta$, deterministic:

$$\text{for all } A \text{ with } \sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i \ : \sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i$$

This work extends Swiper (Tonkikh, Freitas '24).
Adopt terminology: party i gets t_i "tickets".

- pure optimization problem
- NP-hard? – unknown

Coming next...

- ☑ Problem statement
- ☐ **Swiper overview**
- ☐
- ☐
- ☐
- ☐
- ☐

Swiper [TF23] overview: verifying a solution

Input: $(w_1, t_1), ..., (w_n, t_n) \in \mathbb{Z}^2_{\geq 0}$

Output: True/False – test if the solution $(t_1, ..., t_n)$ is valid; need to check

$$\text{for all } A \text{ with } \sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i : \sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i.$$

Swiper: knapsack problem, dynamic programming solution in time $O(n \cdot T)$, where $T = \sum_{i=1}^{n} t_i$.

for $k \in [n]$ **do**

  for $t \in \{0, 1, ..., T\}$ **do**

    compute min weight the adversary has to corrupt from $[k]$ to get t tickets total:

$$\mathrm{dp}[t] = \min_{S} \sum_{i \in S} w_i \text{ subject to } S \subseteq [k] \text{ and } \sum_{i \in S} t_i = t$$

return $\neg \exists t$ s.t. $t \geq \beta \sum_{i=1}^{n} t_i \wedge \mathrm{dp}[t] \leq \alpha \sum_{i=1}^{n} w_i$;

Swiper [TF23] overview: verifying a solution

Input: $(w_1, t_1), ..., (w_n, t_n) \in \mathbb{Z}^2_{\geq 0}$
Output: True/False – test if the solution $(t_1, ..., t_n)$ is valid; need to check

$$\text{for all } A \text{ with } \sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i : \sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i.$$

Swiper: knapsack problem, dynamic programming solution in time $O\big(n \cdot T\big)$, where $T = \sum_{i=1}^{n} t_i$.

for $k \in [n]$ **do**
  for $t \in \{0, 1, ..., T\}$ **do**
    compute min weight the adversary has to corrupt from $[k]$ to get t tickets total:

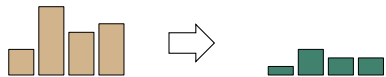
$$\mathtt{dp}[t] = \min_{S} \sum_{i \in S} w_i \text{ subject to } S \subseteq [k] \text{ and } \sum_{i \in S} t_i = t$$

return $\neg \exists t$ s.t. $t \geq \beta \sum_{i=1}^{n} t_i \wedge \mathtt{dp}[t] \leq \alpha \sum_{i=1}^{n} w_i$;

Swiper [TF23] overview

Linear scaling:

- assignment $(t_1, ..., t_n) = (w_1, ..., w_n)$ works, $(t_1, ..., t_n) = 0^n$ doesn't
- set in-between: $t_i = \lfloor s w_i + c \rfloor$ $(c = \mathrm{const})$
- run binary search to find locally minimal s that generates a valid ticket assignment
  - validity testing described in previous slide



Equivalently:

- define potential solutions $\vec{t_1}, \vec{t_2}, ...$ with $\mathrm{size}(\vec{t_j}) = j$ tickets
- find locally minimal j such that $\vec{t_j}$ is valid but $\vec{t_{j-1}}$ is not

Swiper paper proves that when $c = \alpha$, all $\vec{t_M}, \vec{t_{M+1}}, ...$ are valid, where

$$M = \left\lfloor \frac{\alpha(1-\alpha)}{\beta - \alpha} n + 1 \right\rfloor = O(n).$$

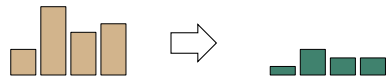
Running time analysis:

- each search iteration: validity testing in time $O\left(n \cdot \sum_{i=1}^{n} t_i\right) \leq O(n \cdot M) = O(n^2)$
- total: $O(n^2 \log n)$

Swiper [TF23] overview

Linear scaling:

- assignment $(t_1, ..., t_n) = (w_1, ..., w_n)$ works, $(t_1, ..., t_n) = 0^n$ doesn't
- set in-between: $t_i = \lfloor s w_i + c \rfloor$ $(c = \text{const})$
- run binary search to find locally minimal s that generates a valid ticket assignment
  - validity testing described in previous slide



Equivalently:

- define potential solutions $\vec{t_1}, \vec{t_2}, ...$ with $\text{size}(\vec{t_j}) = j$ tickets
- find locally minimal j such that $\vec{t_j}$ is valid but $\vec{t_{j-1}}$ is not

Swiper paper proves that when $c = \alpha$, all $\vec{t_M}, \vec{t_{M+1}}, ...$ are valid, where

$$M = \left\lfloor \frac{\alpha(1-\alpha)}{\beta - \alpha} n + 1 \right\rfloor = O(n).$$

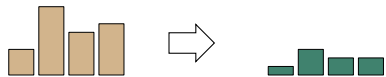
Running time analysis:

- each search iteration: validity testing in time $O\left(n \cdot \sum_{i=1}^{n} t_i\right) \leq O\left(n \cdot M\right) = O\left(n^2\right)$
- total: $O\left(n^2 \log n\right)$

Swiper [TF23] overview

Linear scaling:

- assignment $(t_1, ..., t_n) = (w_1, ..., w_n)$ works, $(t_1, ..., t_n) = 0^n$ doesn't
- set in-between: $t_i = \lfloor s w_i + c \rfloor$ $(c = \text{const})$
- run binary search to find locally minimal s that generates a valid ticket assignment
  - validity testing described in previous slide



Equivalently:

- define potential solutions $\vec{t_1}, \vec{t_2}, ...$ with $\text{size}(\vec{t_j}) = j$ tickets
- find locally minimal j such that $\vec{t_j}$ is valid but $\vec{t_{j-1}}$ is not

Swiper paper proves that when $c = \alpha$, all $\vec{t_M}, \vec{t_{M+1}}, ...$ are valid, where

$$M = \left\lfloor \frac{\alpha(1-\alpha)}{\beta - \alpha} n + 1 \right\rfloor = O(n).$$

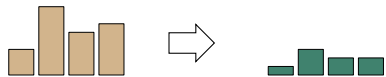
Running time analysis:

- each search iteration: validity testing in time $O\left(n \cdot \sum_{i=1}^{n} t_i\right) \leq O\left(n \cdot M\right) = O\left(n^2\right)$
- total: $O\left(n^2 \log n\right)$

Swiper [TF23] overview

Linear scaling:

- assignment $(t_1, ..., t_n) = (w_1, ..., w_n)$ works, $(t_1, ..., t_n) = 0^n$ doesn't
- set in-between: $t_i = \lfloor s w_i + c \rfloor$ $(c = \text{const})$
- run binary search to find locally minimal s that generates a valid ticket assignment
  - validity testing described in previous slide



Equivalently:

- define potential solutions $\vec{t_1}, \vec{t_2}, ...$ with $\text{size}(\vec{t_j}) = j$ tickets
- find locally minimal j such that $\vec{t_j}$ is valid but $\vec{t_{j-1}}$ is not

Swiper paper proves that when $c = \alpha$, all $\vec{t_M}, \vec{t_{M+1}}, ...$ are valid, where

$$M = \left\lfloor \frac{\alpha(1-\alpha)}{\beta - \alpha} n + 1 \right\rfloor = O(n).$$

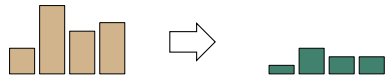
Running time analysis:

- each search iteration: validity testing in time $O\left(n \cdot \sum_{i=1}^{n} t_i\right) \leq O\left(n \cdot M\right) = O\left(n^2\right)$
- total: $O\left(n^2 \log n\right)$

Coming next...

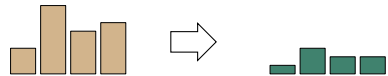
- ☑ Problem statement
- ☑ Swiper overview
- ☐ **Contribution 1: improving Swiper**
- ☐
- ☐
- ☐
- ☐

Improving Swiper: *super Swiper*



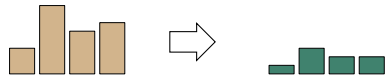
- Replaces binary search with linear search
  - $j \leftarrow 1$;
    while $\overrightarrow{t_j}$ is not valid **do**
    $\quad j \leftarrow j + 1$;
    return $\overrightarrow{t_j}$;
  - binary search can get stuck in a "local minimum"
  - linear search improves output
- reduces running time from $O(n^2 \log n)$ to $O(n + R^2 \log R)$, where $R \leq O(n)$ is the number of tickets $\sum_{i=1}^{n} t_i$ in the output
  - not worse than before
  - normally, $R \ll n \implies n + R^2 \log R \ll n^2 \log n$

Improving Swiper: *super Swiper*



- Replaces binary search with linear search
  - $j \leftarrow 1$;
    while $\overrightarrow{t_j}$ is not valid **do**
    $j \leftarrow j + 1$;
    return $\overrightarrow{t_j}$;
  - binary search can get stuck in a "local minimum"
  - linear search improves output
- reduces running time from $O(n^2 \log n)$ to $O(n + R^2 \log R)$, where $R \leq O(n)$ is the number of tickets $\sum_{i=1}^{n} t_i$ in the output
  - not worse than before
  - normally, $R \ll n \implies n + R^2 \log R \ll n^2 \log n$

Improving Swiper: *super Swiper*



- Replaces binary search with linear search
  - $j \leftarrow 1$;
    while $\overrightarrow{t_j}$ is not valid **do**
    $\quad j \leftarrow j + 1$;
    return $\overrightarrow{t_j}$;

    Challenge 1: compute $\overrightarrow{t_j}$
    Challenge 2: test $\overrightarrow{t_j}$
  - binary search can get stuck in a "local minimum"
  - linear search improves output
- reduces running time from $O(n^2 \log n)$ to $O(n + R^2 \log R)$, where $R \leq O(n)$ is the number of tickets $\sum_{i=1}^{n} t_i$ in the output
  - not worse than before
  - normally, $R \ll n \implies n + R^2 \log R \ll n^2 \log n$

Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$

- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

weights $\vec{w}$: 9 5 4

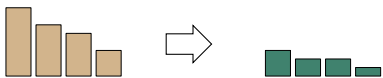
tickets $\vec{t}$: 0 0 0

next s: $1/9$

Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

weights $\vec{w}$: 9 5 4

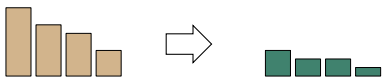
tickets $\vec{t}$: 0 0 0

next s: 1/9

Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

weights $\vec{w}$: 9 5 4

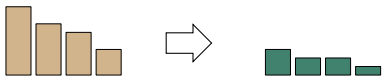
tickets $\vec{t}$: 0 0 0

next s: 1/9

Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

weights $\vec{w}$: 9 5 4

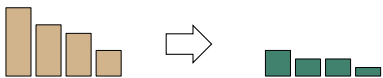
tickets $\vec{t}$: 0 0 0

next s: 1/9

Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

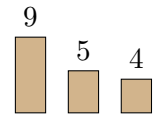
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

weights $\vec{w}$:



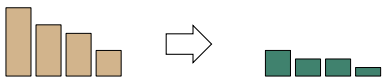
tickets $\vec{t}$: $\underline{0}$ $\underline{0}$ $\underline{0}$

next s: $1/9$

Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

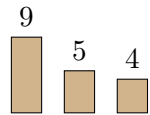
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



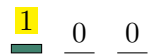
- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

weights $\vec{w}$:



tickets $\vec{t}$:

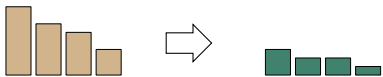


next s:

Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

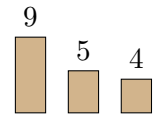
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



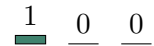
- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

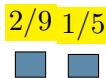
weights $\vec{w}$:



tickets $\vec{t}$:



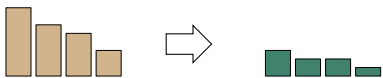
next s:



Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

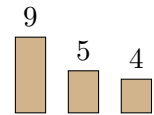
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



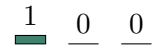
- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

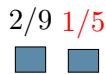
weights $\vec{w}$:



tickets $\vec{t}$:



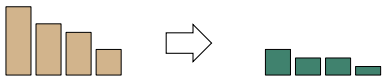
next s:



Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

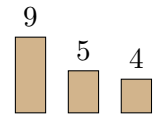
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

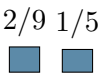
$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

weights $\vec{w}$:



tickets $\vec{t}$:

next s:

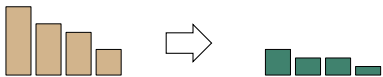


Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

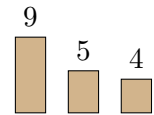
$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



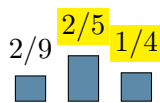
- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

weights $\vec{w}$:



tickets $\vec{t}$:

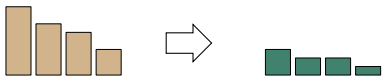
next s:



Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

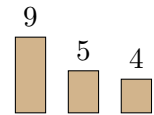
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

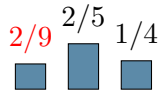
$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

weights $\vec{w}$:



tickets $\vec{t}$:

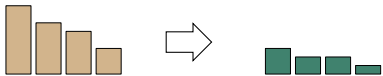
next s:



Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

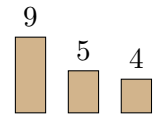
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



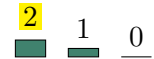
- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

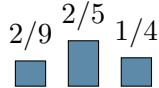
weights $\vec{w}$:



tickets $\vec{t}$:



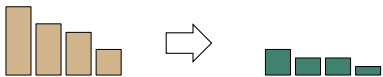
next s:



Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

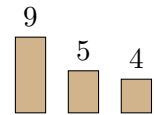
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



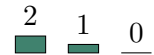
- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

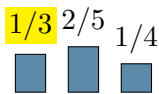
weights $\vec{w}$:



tickets $\vec{t}$:



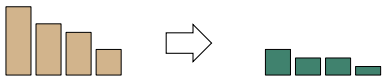
next s:



Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

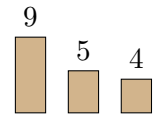
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



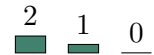
- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

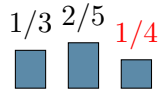
weights $\vec{w}$:



tickets $\vec{t}$:



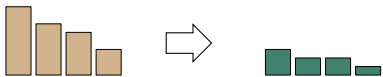
next s:



Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

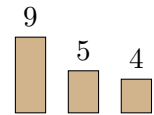
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



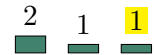
- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

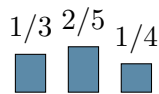
weights $\vec{w}$:



tickets $\vec{t}$:



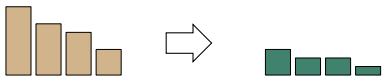
next s:



Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

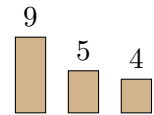
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



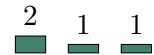
- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

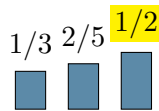
weights $\vec{w}$:



tickets $\vec{t}$:



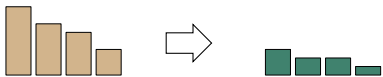
next s:



Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

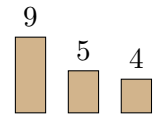
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



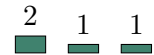
- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

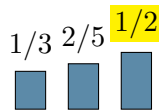
weights $\vec{w}$:



tickets $\vec{t}$:



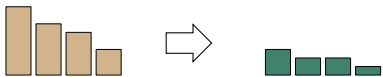
next s:



Computing $\vec{t_j}$

$$\vec{t} = \lfloor s\vec{w} + c \rfloor$$

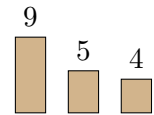
- Challenge: how to compute $\vec{t_1}, ..., \vec{t_R}$?
- assume $w_1 \geq w_2 \geq ... \geq w_n$, produce sorted $\vec{t_j}$



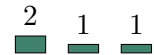
- observation: $\overrightarrow{t_{j+1}} = \vec{t_j} + (..., 0, 0, 1, 0, 0, ...)$ – calculate index to increment
- "slowly" increase s and see which indices increase
  - for index i, store next s that increments t_i
  - each iteation: find minimum s, increment t_i
  - using binary heap: j-th query takes time $O(\log j)$
- computing $\vec{t_1}, ..., \vec{t_R}$ takes time $O(R \log R)$

$c = 0$: $\vec{t} = \lfloor s\vec{w} \rfloor$

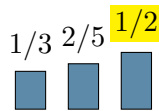
weights $\vec{w}$:



tickets $\vec{t}$:



next s:



Improving Swiper: *super Swiper* (cont.)

$j \leftarrow 1$;
while $\vec{t_j}$ is not valid **do**
  $j \leftarrow j + 1$;
return $\vec{t_j}$;

- Challenge 1: compute $\vec{t_j}$ – time $O(R \log R)$
  - $O(n + R \log R)$ if $\vec{w}$ is unordered
- Challenge 2: test $\vec{t_j}$ – ?

Testing $\vec{t_j}$

- Reuse dynamic programming computations for testing ticket assignments
- generalized data structure DP:

method	mutable?	parameters	output
.apply(w, t)	mutable	party's weight w_i, # tickets t_i	none
.get()	immutable	none	integer

- guarantee: want to test $\vec{t}$
  - call DP.apply$\left(w_i, (\vec{t})_i\right)$ once for all $i \in [n]$ with $(\vec{t})_i \neq 0$ in any order
  - then DP.get() returns max # tickets the adversary gets in $\vec{t}$
- example: want to test $\vec{t} = (4, 3, 3, 0)$ for $\vec{w} = (90, 60, 50, 10)$

Testing $\vec{t_j}$

- Reuse dynamic programming computations for testing ticket assignments
- generalized data structure `DP`:

method	mutable?	parameters	output
`.apply`(w, t)	mutable	party's weight w_i, # tickets t_i	none
`.get`$()$	immutable	none	integer

- guarantee: want to test $\vec{t}$
  - call `DP.apply`$\left(w_i, (\vec{t})_i\right)$ once for all $i \in [n]$ with $(\vec{t})_i \neq 0$ in any order
  - then `DP.get`$()$ returns max # tickets the adversary gets in $\vec{t}$
- example: want to test $\vec{t} = (4, 3, 3, 0)$ for $\vec{w} = (90, 60, 50, 10)$

Testing $\vec{t_j}$

- Reuse dynamic programming computations for testing ticket assignments
- generalized data structure `DP`:

method	mutable?	parameters	output
`.apply`(w, t)	mutable	party's weight w_i, # tickets t_i	none
`.get`$()$	immutable	none	integer

- guarantee: want to test $\vec{t}$
  - call `DP.apply`$\left(w_i, (\vec{t})_i\right)$ once for all $i \in [n]$ with $(\vec{t})_i \neq 0$ in any order
  - then `DP.get`$()$ returns max # tickets the adversary gets in $\vec{t}$
- example: want to test $\vec{t} = (4, 3, 3, 0)$ for $\vec{w} = (90, 60, 50, 10)$

Testing $\vec{t_j}$

- Reuse dynamic programming computations for testing ticket assignments
- generalized data structure DP:

method	mutable?	parameters	output
.apply(w, t)	mutable	party's weight w_i, # tickets t_i	none
.get()	immutable	none	integer

- guarantee: want to test $\vec{t}$
  - call DP.apply$\left(w_i, (\vec{t})_i\right)$ once for all $i \in [n]$ with $(\vec{t})_i \neq 0$ in any order
  - then DP.get() returns max # tickets the adversary gets in $\vec{t}$
- example: want to test $\vec{t} = (4, 3, 3, 0)$ for $\vec{w} = (90, 60, 50, 10)$

Testing $\vec{t_j}$

- Reuse dynamic programming computations for testing ticket assignments
- generalized data structure `DP`:

method	mutable?	parameters	output
`.apply`(w, t)	mutable	party's weight w_i, # tickets t_i	none
`.get`$()$	immutable	none	integer

- guarantee: want to test $\vec{t}$
  - call `DP.apply`$\left(w_i, (\vec{t})_i\right)$ once for all $i \in [n]$ with $(\vec{t})_i \neq 0$ in any order
  - then `DP.get`$()$ returns max # tickets the adversary gets in $\vec{t}$
- example: want to test $\vec{t} = (4, 3, 3, 0)$ for $\vec{w} = (90, 60, 50, 10)$

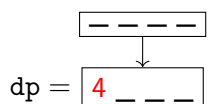
  `dp` $=$ $\boxed{\text{– – – –}}$

Testing $\vec{t_j}$

- Reuse dynamic programming computations for testing ticket assignments
- generalized data structure `DP`:

method	mutable?	parameters	output
`.apply`(w, t)	mutable	party's weight w_i, # tickets t_i	none
`.get`$()$	immutable	none	integer

- guarantee: want to test $\vec{t}$
  - call `DP.apply`$\left(w_i, (\vec{t})_i\right)$ once for all $i \in [n]$ with $(\vec{t})_i \neq 0$ in any order
  - then `DP.get`$()$ returns max # tickets the adversary gets in $\vec{t}$
- example: want to test $\vec{t} = (4, 3, 3, 0)$ for $\vec{w} = (90, 60, 50, 10)$



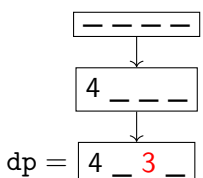
`dp.apply`$(90, 4)$

Testing $\vec{t_j}$

- Reuse dynamic programming computations for testing ticket assignments
- generalized data structure `DP`:

method	mutable?	parameters	output
`.apply`(w, t)	mutable	party's weight w_i, # tickets t_i	none
`.get`$()$	immutable	none	integer

- guarantee: want to test $\vec{t}$
  - call `DP.apply`$\left(w_i, (\vec{t})_i\right)$ once for all $i \in [n]$ with $(\vec{t})_i \neq 0$ in any order
  - then `DP.get`$()$ returns max # tickets the adversary gets in $\vec{t}$
- example: want to test $\vec{t} = (4, 3, 3, 0)$ for $\vec{w} = (90, 60, 50, 10)$



`dp.apply`$(90, 4)$

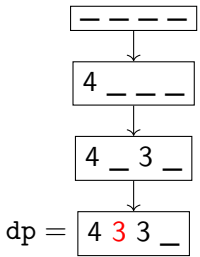
`dp.apply`$(50, 3)$

Testing $\vec{t_j}$

- Reuse dynamic programming computations for testing ticket assignments
- generalized data structure DP:

method	mutable?	parameters	output
.apply(w, t)	mutable	party's weight w_i, # tickets t_i	none
.get()	immutable	none	integer

- guarantee: want to test $\vec{t}$
  - call DP.apply$\left(w_i, (\vec{t})_i\right)$ once for all $i \in [n]$ with $(\vec{t})_i \neq 0$ in any order
  - then DP.get() returns max # tickets the adversary gets in $\vec{t}$
- example: want to test $\vec{t} = (4, 3, 3, 0)$ for $\vec{w} = (90, 60, 50, 10)$



dp.apply$(90, 4)$

dp.apply$(50, 3)$

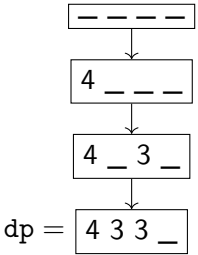
dp.apply$(60, 3)$

Testing $\vec{t_j}$

- Reuse dynamic programming computations for testing ticket assignments
- generalized data structure DP:

method	mutable?	parameters	output
.apply(w, t)	mutable	party's weight w_i, # tickets t_i	none
.get()	immutable	none	integer

- guarantee: want to test $\vec{t}$
  - call DP.apply$\left(w_i, (\vec{t})_i\right)$ once for all $i \in [n]$ with $(\vec{t})_i \neq 0$ in any order
  - then DP.get() returns max # tickets the adversary gets in $\vec{t}$
- example: want to test $\vec{t} = (4, 3, 3, 0)$ for $\vec{w} = (90, 60, 50, 10)$



dp.apply$(90, 4)$

dp.apply$(50, 3)$

dp.apply$(60, 3)$

dp.get()

Testing $\vec{t_j}$ (cont.)

- To implement DP:

  for $k \in [n]$ **do**

  for $t \in \{0, 1, ..., T\}$ **do**

  apply (compute min weight the adversary has to corrupt from $[k]$ to get t tickets total:

  $$\mathrm{dp}[t] = \min_{S} \sum_{i \in S} w_i \text{ subject to } S \subseteq [k] \text{ and } \sum_{i \in S} t_i = t$$

  get* (**return** $\neg\exists t$ s.t. $t \geq \beta \sum_{i=1}^{n} t_i \wedge \mathrm{dp}[t] \leq \alpha \sum_{i=1}^{n} w_i$;

- time complexity
  - if $\sum_{i=1}^{n} (\vec{t})_i = T$, then DP.apply$\left(w_i, (\vec{t})_i\right)$ takes time $O(T)$
  - DP.get() takes time $O(1)$

Testing $\vec{t_j}$ (cont.)

- To implement DP:

  for $k \in [n]$ **do**

  for $t \in \{0, 1, ..., T\}$ **do**

  compute min weight the adversary has to corrupt from $[k]$ to get t tickets total:

  $$\mathrm{dp}[t] = \min_{S} \sum_{i \in S} w_i \text{ subject to } S \subseteq [k] \text{ and } \sum_{i \in S} t_i = t$$

  apply

  get* (**return** $\neg \exists t$ s.t. $t \geq \beta \sum_{i=1}^{n} t_i \wedge \mathrm{dp}[t] \leq \alpha \sum_{i=1}^{n} w_i$;

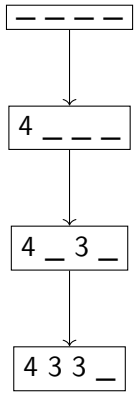
- time complexity
  - if $\sum_{i=1}^{n} (\vec{t})_i = T$, then $\texttt{DP.apply}\big(w_i, (\vec{t})_i\big)$ takes time $O(T)$
  - $\texttt{DP.get}()$ takes time $O(1)$

Testing $\vec{t_j}$ (cont.)

More interesting example: want to test $\vec{t} = (4, 3, 3, 0)$ and $\vec{t'} = (4, 4, 3, 0)$

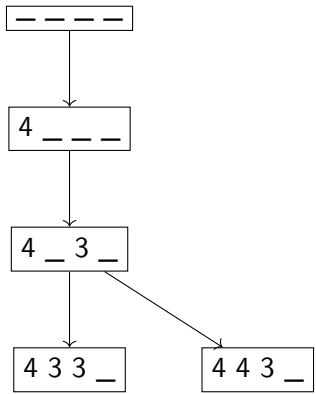
Testing $\vec{t_j}$ (cont.)

More interesting example: want to test $\vec{t} = (4, 3, 3, 0)$ and $\vec{t'} = (4, 4, 3, 0)$



Testing $\vec{t_j}$ (cont.)

More interesting example: want to test $\vec{t} = (4, 3, 3, 0)$ and $\vec{t'} = (4, 4, 3, 0)$



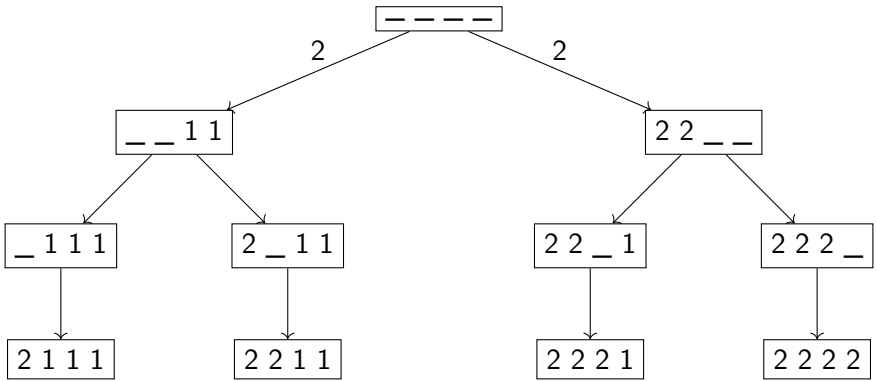
Testing $\vec{t_j}$ (cont.)

More interesting example: want to test $(2,1,1,1)$, $(2,2,1,1)$, $(2,2,2,1)$, $(2,2,2,2)$

$4 \cdot (\log 4 + 1) = 12$ applies. Generalize: can test Swiper's $\vec{t_1}, \vec{t_2}, ..., \overrightarrow{t_{O(R)}}$ with $O(R \log R)$ applies; each apply takes time $O(R)$.

Testing $\vec{t_j}$ (cont.)

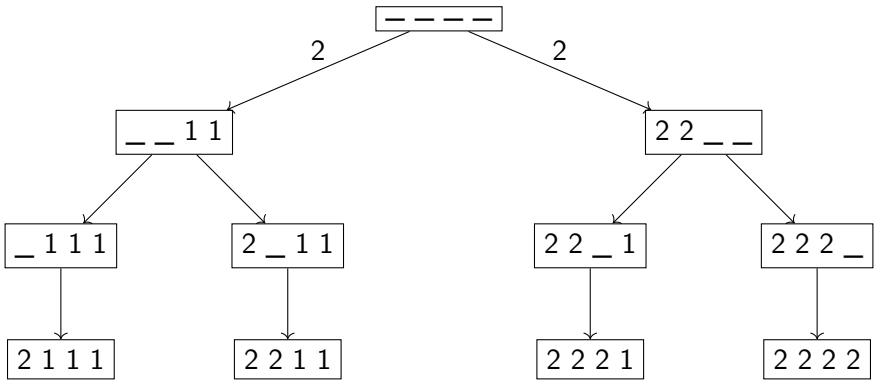
More interesting example: want to test $(2,1,1,1)$, $(2,2,1,1)$, $(2,2,2,1)$, $(2,2,2,2)$



$4 \cdot (\log 4 + 1) = 12$ applies. Generalize: can test Swiper's $\vec{t_1}, \vec{t_2}, ..., \vec{t_{O(R)}}$ with $O(R \log R)$ applies; each apply takes time $O(R)$.

Testing $\vec{t_j}$ (cont.)

More interesting example: want to test $(2,1,1,1)$, $(2,2,1,1)$, $(2,2,2,1)$, $(2,2,2,2)$



$4 \cdot (\log 4 + 1) = 12$ applies. Generalize: can test Swiper's $\vec{t_1}, \vec{t_2}, ..., \vec{t_{O(R)}}$ with $O(R \log R)$ applies; each apply takes time $O(R)$.

Improving Swiper: *super Swiper* (cont.)

$j \leftarrow 1$;
while $\overrightarrow{t_j}$ is not valid **do**
  $j \leftarrow j + 1$;
return $\overrightarrow{t_j}$;

- Challenge 1: compute $\overrightarrow{t_j}$ – time $O(n + R \log R)$
- Challenge 2: test $\overrightarrow{t_j}$ – time $O(R^2 \log R)$

Total: time $O(n + R^2 \log R)$

Improving Swiper: *super Swiper* (cont.)

- linear search outputs fewer tickets
- linear search faster than binary search

	$\beta = 1/3$			$\beta = 1/2$			
	$\alpha = 0.2$	$\alpha = 0.25$	$\alpha = 0.3$	$\alpha = 0.3$	$\alpha = 0.35$	$\alpha = 0.4$	$\alpha = 0.45$
Swiper	22	85	346	23	29	95	279
	1.61ms	2.07ms	1.96ms	1.40ms	1.88ms	1.80ms	1.96ms
super Swiper	22	58	277	23	29	95	241
	6.98µs	14.1µs	121µs	7.21µs	7.74µs	33.2µs	99.9µs

Table: Evaluation on the Aptos stake distribution

Improving Swiper: *super Swiper* (cont.)

- linear search outputs fewer tickets
- linear search faster than binary search

	$\beta = 1/3$			$\beta = 1/2$			
	$\alpha = 0.2$	$\alpha = 0.25$	$\alpha = 0.3$	$\alpha = 0.3$	$\alpha = 0.35$	$\alpha = 0.4$	$\alpha = 0.45$
Swiper	22	85	346	23	29	95	279
	1.61ms	2.07ms	1.96ms	1.40ms	1.88ms	1.80ms	1.96ms
super Swiper	22	58	277	23	29	95	241
	6.98µs	14.1µs	121µs	7.21µs	7.74µs	33.2µs	99.9µs

Table: Evaluation on the Aptos stake distribution

Improving Swiper: *super Swiper* (cont.)

- linear search outputs fewer tickets
- linear search faster than binary search

	$\beta = 1/3$			$\beta = 1/2$			
	$\alpha = 0.2$	$\alpha = 0.25$	$\alpha = 0.3$	$\alpha = 0.3$	$\alpha = 0.35$	$\alpha = 0.4$	$\alpha = 0.45$
Swiper	235	745	22393	203	383	1203	17591
	1.10s	1.22s	1.78s	1.03s	1.14s	1.19s	1.65s
super Swiper	232	745	22384	201	383	1171	17581
	81.5µs	485µs	129ms	88.0µs	215µs	1.54ms	140ms

Table: Evaluation on the Algorand stake distribution

Coming next...

- ☑ Problem statement
- ☑ Swiper overview
- ☑ Contribution 1: *super Swiper* – linear search in time $O\left(n + R^2 \log R\right)$
- ☐ **Contribution 2: lower bounds**
- ☐
- ☐
- ☐

Lower bounds

- *super Swiper* works great in practice, what about in theory?
- want to know worst case approximation factor
  - how big is output compared to optimal solution?
- our result: *super Swiper*, Swiper [TF23] + others [BHS23, FMT24, DPTX25] all have approximation factor $\Omega(n)$
  - constructed an example where output is $\Omega(n)$ but $\mathrm{OPT} = O(1)$

Lower bounds

- *super Swiper* works great in practice, what about in theory?
- want to know worst case approximation factor
  - how big is output compared to optimal solution?
- our result: *super Swiper*, Swiper [TF23] + others [BHS23, FMT24, DPTX25] all have approximation factor $\Omega(n)$
  - constructed an example where output is $\Omega(n)$ but $\mathrm{OPT} = O(1)$

Lower bounds

- *super Swiper* works great in practice, what about in theory?
- want to know worst case approximation factor
  - how big is output compared to optimal solution?
- our result: *super Swiper*, Swiper [TF23] + others [BHS23, FMT24, DPTX25] all have approximation factor $\Omega(n)$
  - constructed an example where output is $\Omega(n)$ but $\mathrm{OPT} = O(1)$

Coming next...

- ☑ Problem statement
- ☑ Swiper overview
- ☑ Contribution 1: *super Swiper* – linear search in time $O\left(n + R^2 \log R\right)$
- ☑ Contribution 2: $\Omega(n)$ approximation factor lower bound
- ☐ **Contribution 3: algorithms with better guarantees**
- ☐
- ☐

Algorithms with better theoretical guarantees

- Would like algorithms with good approximation factor; made some progress
- useful fact: assuming $w_1 \geq w_2 \geq ... \geq w_n$,
  $\exists$ an optimal solution with $t_1 \geq t_2 \geq ... \geq t_n$
  - take an optimal solution $(t_1, ..., t_i, ...t_j, ..., t_n)$ with $t_i < t_j$
  - swap t_i and t_j
- bruteforcing sorted ticket assignment gives exact solution (approximation factor 1), takes time

$$O\left(n + R^{3/2} e^{C\sqrt{R}}\right)$$

  - $R \leq O(n)$ – size of optimal solution
  - $C = \pi\sqrt{6}/3 \approx 2.57$
  - practical when $R < 100$
- corollary: polytime algorithm with approximation factor $O\left(n/\log^2 n\right)$

Algorithms with better theoretical guarantees

- Would like algorithms with good approximation factor; made some progress
- useful fact: assuming $w_1 \geq w_2 \geq ... \geq w_n$,
  $\exists$ an optimal solution with $t_1 \geq t_2 \geq ... \geq t_n$
  - take an optimal solution $(t_1, ..., t_i, ...t_j, ..., t_n)$ with $t_i < t_j$
  - swap t_i and t_j
- bruteforcing sorted ticket assignment gives exact solution (approximation factor 1), takes time

$$O\left(n + R^{3/2} e^{C\sqrt{R}}\right)$$

  - $R \leq O(n)$ – size of optimal solution
  - $C = \pi\sqrt{6}/3 \approx 2.57$
  - practical when $R < 100$
- corollary: polytime algorithm with approximation factor $O\left(n/\log^2 n\right)$

Algorithms with better theoretical guarantees

- Would like algorithms with good approximation factor; made some progress
- useful fact: assuming $w_1 \geq w_2 \geq ... \geq w_n$,
  $\exists$ an optimal solution with $t_1 \geq t_2 \geq ... \geq t_n$
  - take an optimal solution $(t_1, ..., t_i, ...t_j, ..., t_n)$ with $t_i < t_j$
  - swap t_i and t_j
- bruteforcing sorted ticket assignment gives exact solution (approximation factor 1), takes time

$$O\left(n + R^{3/2} e^{C\sqrt{R}}\right)$$

  - $R \leq O(n)$ – size of optimal solution
  - $C = \pi\sqrt{6}/3 \approx 2.57$
  - practical when $R < 100$
- corollary: polytime algorithm with approximation factor $O\left(n/\log^2 n\right)$

Algorithms with better theoretical guarantees

- Would like algorithms with good approximation factor; made some progress
- useful fact: assuming $w_1 \geq w_2 \geq ... \geq w_n$,
  $\exists$ an optimal solution with $t_1 \geq t_2 \geq ... \geq t_n$
  - take an optimal solution $(t_1, ..., t_i, ...t_j, ..., t_n)$ with $t_i < t_j$
  - swap t_i and t_j
- bruteforcing sorted ticket assignment gives exact solution (approximation factor 1), takes time

$$O\left(n + R^{3/2} e^{C\sqrt{R}}\right)$$

  - $R \leq O(n)$ – size of optimal solution
  - $C = \pi\sqrt{6}/3 \approx 2.57$
  - practical when $R < 100$
- corollary: polytime algorithm with approximation factor $O\left(n/\log^2 n\right)$

Coming next...

- ☑ Problem statement
- ☑ Swiper overview
- ☑ Contribution 1: *super Swiper* – linear search in time $O\left(n + R^2 \log R\right)$
- ☑ Contribution 2: $\Omega(n)$ approximation factor lower bound
- ☑ Contribution 3: sub-expontential exact algorithm and polytime approx. algorithm
- ☐ **Contribution 4: LP based algorithm**
- ☐

Algorithm based on linear programming

- Weight reduction can be formulated as an integer linear program

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} t_i \\ \text{subject to} \quad & \sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i \quad \forall A \subseteq [n] \text{ s.t. } \sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i \\ & t_i \in \mathbb{Z}_{\geq 0} \qquad\qquad \forall i \in [n] \end{aligned}$$

- relax to (fractional) LP, run ellipsoid method, round output to an integer solution
  - polynomial time algorithm

Algorithm based on linear programming

- Weight reduction can be formulated as an integer linear program

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} t_i \\ \text{subject to} \quad & \sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i \quad \forall A \subseteq [n] \text{ s.t. } \sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i \\ & t_i \in \mathbb{Z}_{\geq 0} \qquad\qquad \forall i \in [n] \end{aligned}$$

- relax to (fractional) LP, run ellipsoid method, round output to an integer solution
  - polynomial time algorithm

Algorithm based on linear programming (cont.)

- Let $\mathrm{OPT}_{\alpha,\beta}$ denote the smallest number of tickets
- solution of size $\mathrm{OPT}_{\alpha,(1-\delta)\beta}$ in polynomial time for any constant $\delta > 0$
  - example: security $\alpha = \frac{1}{5} \to \beta = \frac{1}{3}$, size optimal for $\alpha = \frac{1}{5} \to (1-\delta)\beta = \frac{3}{10}$
- $\mathrm{OPT}_{\alpha,(1-\delta)\beta}/\mathrm{OPT}_{\alpha,\beta}$ can be $\Omega(n)$;

Algorithm based on linear programming (cont.)

- Let $\mathrm{OPT}_{\alpha,\beta}$ denote the smallest number of tickets
- solution of size $\mathrm{OPT}_{\alpha,(1-\delta)\beta}$ in polynomial time for any constant $\delta > 0$
  - example: security $\alpha = \frac{1}{5} \to \beta = \frac{1}{3}$, size optimal for $\alpha = \frac{1}{5} \to (1-\delta)\beta = \frac{3}{10}$
- $\mathrm{OPT}_{\alpha,(1-\delta)\beta}/\mathrm{OPT}_{\alpha,\beta}$ can be $\Omega(n)$;

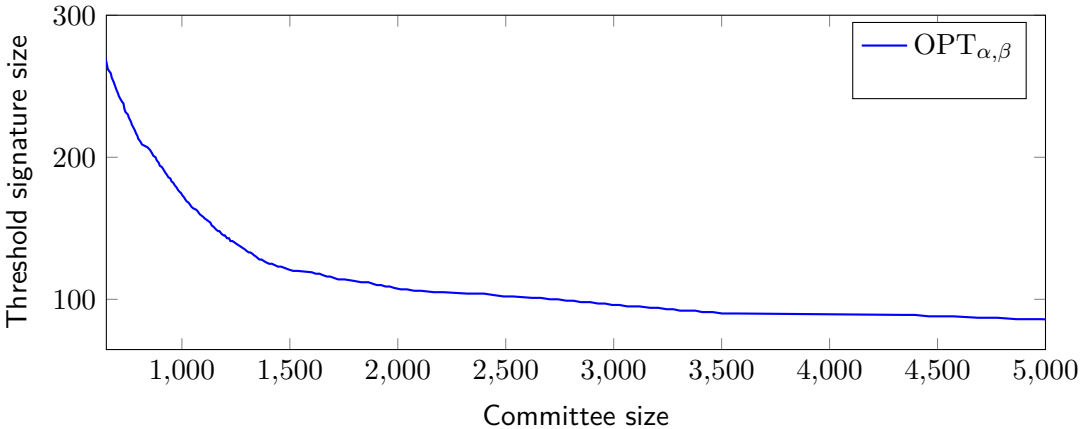
Algorithm based on linear programming (cont.)

- Let $\mathrm{OPT}_{\alpha,\beta}$ denote the smallest number of tickets
- solution of size $\mathrm{OPT}_{\alpha,(1-\delta)\beta}$ in polynomial time for any constant $\delta > 0$
  - example: security $\alpha = \frac{1}{5} \rightarrow \beta = \frac{1}{3}$, size optimal for $\alpha = \frac{1}{5} \rightarrow (1-\delta)\beta = \frac{3}{10}$
- $\mathrm{OPT}_{\alpha,(1-\delta)\beta}/\mathrm{OPT}_{\alpha,\beta}$ can be $\Omega(n)$;

Algorithm based on linear programming (cont.)

$\mathrm{OPT}_{\alpha,(1-\delta)\beta}$ can be useful anyway:

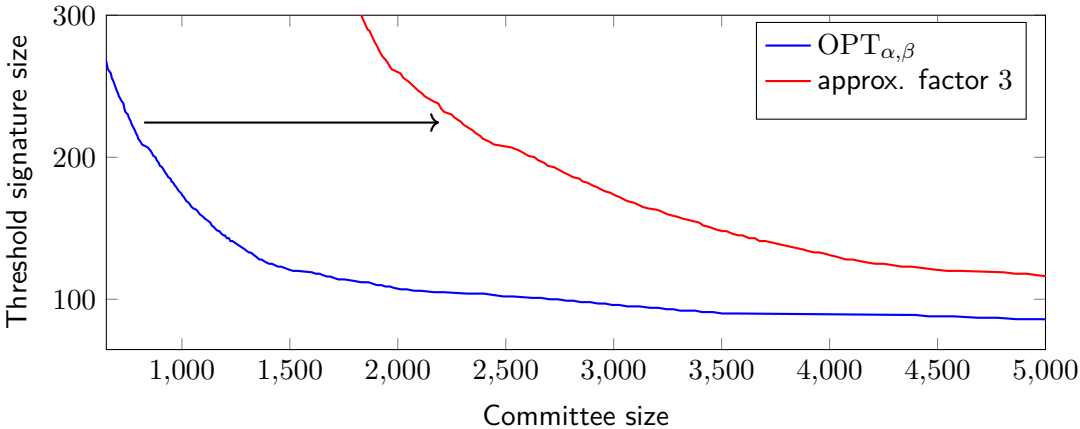
- ex.: Approximate Lower Bound Arguments [CKRZ24] yields decentralized threshold signatures



Algorithm based on linear programming (cont.)

$\mathrm{OPT}_{\alpha,(1-\delta)\beta}$ can be useful anyway:

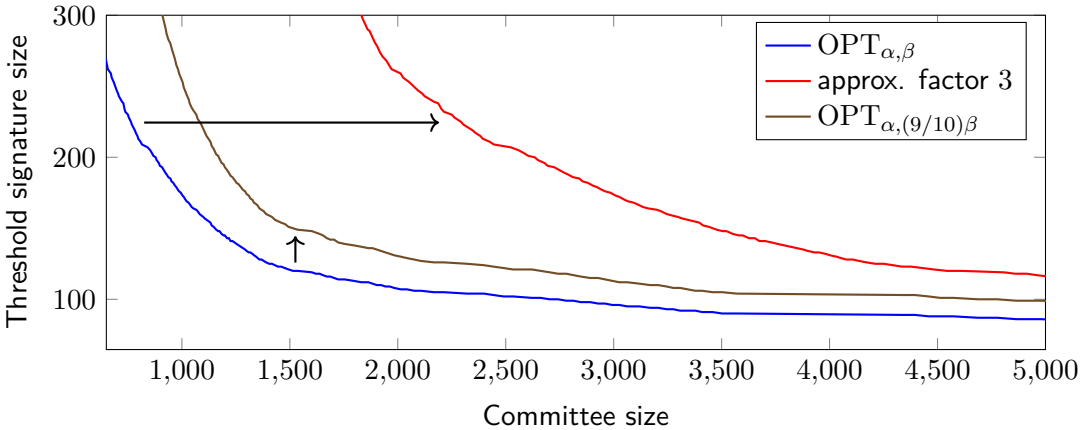
- ex.: Approximate Lower Bound Arguments [CKRZ24] yields decentralized threshold signatures



Algorithm based on linear programming (cont.)

$\mathrm{OPT}_{\alpha,(1-\delta)\beta}$ can be useful anyway:

- ex.: Approximate Lower Bound Arguments [CKRZ24] yields decentralized threshold signatures



Algorithm based on linear programming (cont.)

- LP algorithm is impractical
- time complexity $\Omega\left(n^{10}\right)$ for reasonable parameters

Coming next...

- ☑ Problem statement
- ☑ Swiper overview
- ☑ Contribution 1: *super Swiper* – linear search in time $O\left(n + R^2 \log R\right)$
- ☑ Contribution 2: $\Omega(n)$ approximation factor lower bound
- ☑ Contribution 3: sub-expontential exact algorithm and polytime approx. algorithm
- ☑ Contribution 4: LP based algorithm
- ☐ **Numerical evaluation**

Numerical evaluation

- sub-exponential time exact algorithm can be practical!
- *super Swiper* is almost optimal in practice

	$\beta = 1/3$			$\beta = 1/2$			
	$\alpha = 0.2$	$\alpha = 0.25$	$\alpha = 0.3$	$\alpha = 0.3$	$\alpha = 0.35$	$\alpha = 0.4$	$\alpha = 0.45$
Swiper	22	85	346	23	29	95	279
	1.61ms	2.07ms	1.96ms	1.40ms	1.88ms	1.80ms	1.96ms
super Swiper	22	58	277	23	29	95	241
	6.98µs	14.1µs	121µs	7.21µs	7.74µs	33.2µs	99.9µs
exact	22	55		23	29	91	
	133µs	127ms		236µs	1.08ms	45.9s	

Table: Evaluation on the Aptos stake distribution

Numerical evaluation

- sub-exponential time exact algorithm can be practical!
- *super Swiper* is almost optimal in practice

	$\beta = 1/3$			$\beta = 1/2$			
	$\alpha = 0.2$	$\alpha = 0.25$	$\alpha = 0.3$	$\alpha = 0.3$	$\alpha = 0.35$	$\alpha = 0.4$	$\alpha = 0.45$
Swiper	22	85	346	23	29	95	279
	1.61ms	2.07ms	1.96ms	1.40ms	1.88ms	1.80ms	1.96ms
super Swiper	22	58	277	23	29	95	241
	6.98µs	14.1µs	121µs	7.21µs	7.74µs	33.2µs	99.9µs
exact	22	55		23	29	91	
	133µs	127ms		236µs	1.08ms	45.9s	

Table: Evaluation on the Aptos stake distribution

Numerical evaluation

- sub-exponential time exact algorithm can be practical!
- *super Swiper* is almost optimal in practice

	$\beta = 1/3$			$\beta = 1/2$			
	$\alpha = 0.2$	$\alpha = 0.25$	$\alpha = 0.3$	$\alpha = 0.3$	$\alpha = 0.35$	$\alpha = 0.4$	$\alpha = 0.45$
Swiper	22	85	346	23	29	95	279
	1.61ms	2.07ms	1.96ms	1.40ms	1.88ms	1.80ms	1.96ms
super Swiper	22	58	277	23	29	95	241
	6.98µs	14.1µs	121µs	7.21µs	7.74µs	33.2µs	99.9µs
exact	22	55		23	29	91	
	133µs	127ms		236µs	1.08ms	45.9s	

Table: Evaluation on the Aptos stake distribution

Conclusion

- Good practical improvements
  - *super Swiper*: linear search in time $O\big(n + R^2 \log R\big)$
  - exact algorithm in time $O(n) + 2^{O(\sqrt{R})}$ – sometimes practical
- more theoretical work to be done
  - NP-hard?
  - better approximation factor in polytime
- to appear in DISC 2025
- thanks go to:
  - Leo Reyzin for help with paper & presentation
  - Nathan Klein for ideas & rounding class

Weight reduction problem:
given $w_1, ..., w_n \in \mathbb{Z}_{\geq 0}$, find $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that
- minimize $\sum_{i=1}^{n} t_i$
- for all A with $\sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i$: $\sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i$

Conclusion

- Good practical improvements
  - *super Swiper*: linear search in time $O\big(n + R^2 \log R\big)$
  - exact algorithm in time $O(n) + 2^{O(\sqrt{R})}$ – sometimes practical
- more theoretical work to be done
  - NP-hard?
  - better approximation factor in polytime
- to appear in DISC 2025
- thanks go to:
  - Leo Reyzin for help with paper & presentation
  - Nathan Klein for ideas & rounding class

Weight reduction problem:
given $w_1, ..., w_n \in \mathbb{Z}_{\geq 0}$, find $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that
- minimize $\sum_{i=1}^{n} t_i$
- for all A with $\sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i$: $\sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i$

Conclusion

- Good practical improvements
  - *super Swiper*: linear search in time $O\big(n + R^2 \log R\big)$
  - exact algorithm in time $O(n) + 2^{O(\sqrt{R})}$ – sometimes practical
- more theoretical work to be done
  - NP-hard?
  - better approximation factor in polytime
- to appear in DISC 2025
- thanks go to:
  - Leo Reyzin for help with paper & presentation
  - Nathan Klein for ideas & rounding class

Weight reduction problem:
given $w_1, ..., w_n \in \mathbb{Z}_{\geq 0}$, find $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that
- minimize $\sum_{i=1}^{n} t_i$
- for all A with $\sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i$: $\sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i$

Conclusion

- Good practical improvements
  - *super Swiper*: linear search in time $O\big(n + R^2 \log R\big)$
  - exact algorithm in time $O(n) + 2^{O(\sqrt{R})}$ – sometimes practical
- more theoretical work to be done
  - NP-hard?
  - better approximation factor in polytime
- to appear in DISC 2025
- thanks go to:
  - Leo Reyzin for help with paper & presentation
  - Nathan Klein for ideas & rounding class

Weight reduction problem:
given $w_1, ..., w_n \in \mathbb{Z}_{\geq 0}$, find $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that
- minimize $\sum_{i=1}^{n} t_i$
- for all A with $\sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i$: $\sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i$

Additional slides

Super Swiper

- test $\vec{t_1}, \vec{t_2}, \vec{t_3}, \ldots$ in batches
- batch $k \geq 0$ tests $\vec{t_j}$ for $j \in \left(j_k, j_{k+1}\right]$
- $j_0 = 0$, $j_{k+1} = j_k + 2^k$ (batches grow)
- batch k takes time $O\left(k \cdot 2^{2k}\right)$
- total time dominated by last batch: $O(R^2 \log R)$

Super Swiper

- test $\vec{t_1}, \vec{t_2}, \vec{t_3}, \ldots$ in batches
- batch $k \geq 0$ tests $\vec{t_j}$ for $j \in \left(j_k, j_{k+1}\right]$
- $j_0 = 0$, $j_{k+1} = j_k + 2^k$ (batches grow)
- batch k takes time $O\left(k \cdot 2^{2k}\right)$
- total time dominated by last batch: $O(R^2 \log R)$

Super Swiper(cont.)

Batch algorithm

- goal: test $\overrightarrow{t_j}$ for $j \in (l, r]$, $l = 2^k - 1$, $r = 2^{k+1} - 1$, in time $O(k \cdot 2^{2k})$
- create array `deltas`: indices to increment (in $\overrightarrow{t_l}$) that generate $\overrightarrow{t_{l+1}}, \overrightarrow{t_{l+2}}, ..., \overrightarrow{t_r}$
  - $|\texttt{deltas}| = 2^k$
- create `dp_head` : DP and call $\texttt{dp\_head.apply}(w_i, (\overrightarrow{t_l})_i)$ for $i \notin \texttt{deltas}$
  - takes time $O(2^{2k})$
- observation: could test $\overrightarrow{t_{l+1}}, ..., \overrightarrow{t_r}$ and save $\approx 50\%$ of CPU cycles
  - for each $\overrightarrow{t_j}$ only apply indices $i \in \texttt{deltas}$
- invoke $\textsc{ProcessBatchRecursive}(\overrightarrow{t_l}, \texttt{deltas}, \texttt{dp\_head})$
  - takes time $O(k \cdot 2^{2k})$

Super Swiper(cont.)

Batch algorithm

- goal: test $\vec{t_j}$ for $j \in (l, r]$, $l = 2^k - 1$, $r = 2^{k+1} - 1$, in time $O(k \cdot 2^{2k})$
- create array `deltas`: indices to increment (in $\vec{t_l}$) that generate $\overrightarrow{t_{l+1}}, \overrightarrow{t_{l+2}}, ..., \vec{t_r}$
  - $|\texttt{deltas}| = 2^k$
- create `dp_head` : DP and call $\texttt{dp\_head.apply}(w_i, (\vec{t_l})_i)$ for $i \notin \texttt{deltas}$
  - takes time $O(2^{2k})$
- observation: could test $\overrightarrow{t_{l+1}}, ..., \vec{t_r}$ and save $\approx 50\%$ of CPU cycles
  - for each $\vec{t_j}$ only apply indices $i \in \texttt{deltas}$
- invoke $\textsc{ProcessBatchRecursive}(\vec{t_l}, \texttt{deltas}, \texttt{dp\_head})$
  - takes time $O(k \cdot 2^{2k})$

Super Swiper(cont.)

Batch algorithm

- goal: test $\vec{t_j}$ for $j \in (l, r]$, $l = 2^k - 1$, $r = 2^{k+1} - 1$, in time $O(k \cdot 2^{2k})$
- create array `deltas`: indices to increment (in $\vec{t_l}$) that generate $\overrightarrow{t_{l+1}}, \overrightarrow{t_{l+2}}, ..., \vec{t_r}$
  - $|\texttt{deltas}| = 2^k$
- create `dp_head` : `DP` and call $\texttt{dp\_head.apply}(w_i, (\vec{t_l})_i)$ for $i \notin \texttt{deltas}$
  - takes time $O(2^{2k})$
- observation: could test $\overrightarrow{t_{l+1}}, ..., \vec{t_r}$ and save $\approx 50\%$ of CPU cycles
  - for each $\vec{t_j}$ only apply indices $i \in \texttt{deltas}$
- invoke $\textsc{ProcessBatchRecursive}(\vec{t_l}, \texttt{deltas}, \texttt{dp\_head})$
  - takes time $O(k \cdot 2^{2k})$

Super Swiper(cont.)

Batch algorithm

- goal: test $\vec{t_j}$ for $j \in (l, r]$, $l = 2^k - 1$, $r = 2^{k+1} - 1$, in time $O(k \cdot 2^{2k})$
- create array `deltas`: indices to increment (in $\vec{t_l}$) that generate $\overrightarrow{t_{l+1}}, \overrightarrow{t_{l+2}}, ..., \vec{t_r}$
  - $|\texttt{deltas}| = 2^k$
- create `dp_head` : `DP` and call $\texttt{dp\_head.apply}(w_i, (\vec{t_l})_i)$ for $i \notin \texttt{deltas}$
  - takes time $O(2^{2k})$
- observation: could test $\overrightarrow{t_{l+1}}, ..., \vec{t_r}$ and save $\approx 50\%$ of CPU cycles
  - for each $\vec{t_j}$ only apply indices $i \in \texttt{deltas}$
- invoke $\textsc{ProcessBatchRecursive}(\vec{t_l}, \texttt{deltas}, \texttt{dp\_head})$
  - takes time $O(k \cdot 2^{2k})$

Super Swiper(cont.)

Batch algorithm

- goal: test $\vec{t_j}$ for $j \in (l, r]$, $l = 2^k - 1$, $r = 2^{k+1} - 1$, in time $O(k \cdot 2^{2k})$
- create array `deltas`: indices to increment (in $\vec{t_l}$) that generate $\overrightarrow{t_{l+1}}, \overrightarrow{t_{l+2}}, ..., \vec{t_r}$
  - $|\texttt{deltas}| = 2^k$
- create `dp_head` : `DP` and call $\texttt{dp\_head.apply}(w_i, (\vec{t_l})_i)$ for $i \notin \texttt{deltas}$
  - takes time $O(2^{2k})$
- observation: could test $\overrightarrow{t_{l+1}}, ..., \vec{t_r}$ and save $\approx 50\%$ of CPU cycles
  - for each $\vec{t_j}$ only apply indices $i \in \texttt{deltas}$
- invoke $\textsc{ProcessBatchRecursive}(\vec{t_l}, \texttt{deltas}, \texttt{dp\_head})$
  - takes time $O(k \cdot 2^{2k})$

Super Swiper(cont.)

Divide and conquer

Base case. If $|\texttt{deltas}| = 1$:

- call $\texttt{dp\_head.apply}\left(w_i, (\overrightarrow{t_{j+1}})_i\right)$ where $i \in \texttt{deltas}$
- check $\texttt{dp\_head.get}()$

ProcessBatchRecursive($\overrightarrow{t_j}$, deltas, dp_head)

- Input:
  - $\overrightarrow{t_j}$: a ticket assignment
  - deltas: an array of indices that generate $\overrightarrow{t_{j+1}}, \overrightarrow{t_{j+2}}, ..., \overrightarrow{t_{j+|\texttt{deltas}|}}$
  - dp_head: DP data structure where $\left(w_i, \left(\overrightarrow{t_j}\right)_i\right)$ are applied exactly for $i \notin \texttt{deltas}$
- Output:
  - the smallest valid ticket assignment $\overrightarrow{t_{j'}}$ where $j < j' \leq j + |\texttt{deltas}|$, or $\perp$ if no such solution exists

Super Swiper(cont.)

Divide and conquer

Base case. If $|\texttt{deltas}| = 1$:

- call $\texttt{dp\_head.apply}\left(w_i, (\overrightarrow{t_{j+1}})_i\right)$ where $i \in \texttt{deltas}$
- check $\texttt{dp\_head.get}()$

ProcessBatchRecursive($\overrightarrow{t_j}$, deltas, dp_head)

- Input:
  - $\overrightarrow{t_j}$: a ticket assignment
  - deltas: an array of indices that generate $\overrightarrow{t_{j+1}}, \overrightarrow{t_{j+2}}, ..., \overrightarrow{t_{j+|\texttt{deltas}|}}$
  - dp_head: DP data structure where $\left(w_i, \left(\overrightarrow{t_j}\right)_i\right)$ are applied exactly for $i \notin$ deltas
- Output:
  - the smallest valid ticket assignment $\overrightarrow{t_{j'}}$ where $j < j' \leq j + |\texttt{deltas}|$, or $\perp$ if no such solution exists

Super Swiper(cont.)

Recursive case. If $|\texttt{deltas}| > 1$:

- split deltas into two halves
  - deltas_l, deltas_r
- to test left half:
  - dp_l $\leftarrow$ dp_head
  - call dp_l.apply$\left(w_i, \left(\overrightarrow{t_j}\right)_i\right)$ for $i \in$ deltas_r $\setminus$ deltas_l
  - call ProcessBatchRecursive$(\overrightarrow{t_j}$, deltas_l, dp_l$)$
- to test right half:
  - $m \leftarrow j + |\texttt{deltas\_l}|$
  - dp_r $\leftarrow$ dp_head
  - call dp_r.apply$\left(w_i, \left(\overrightarrow{t_m}\right)_i\right)$ for $i \in$ deltas_l $\setminus$ deltas_r
  - call ProcessBatchRecursive$(\overrightarrow{t_m}$, deltas_r, dp_r$)$

ProcessBatchRecursive$(\overrightarrow{t_j}$, deltas, dp_head$)$

- Input:
  - $\overrightarrow{t_j}$: a ticket assignment
  - deltas: an array of indices that generate $\overrightarrow{t_{j+1}}, \overrightarrow{t_{j+2}}, ..., \overrightarrow{t_{j+|\texttt{deltas}|}}$
  - dp_head: DP data structure where $\left(w_i, \left(\overrightarrow{t_j}\right)_i\right)$ are applied exactly for $i \notin$ deltas
- Output:
  - the smallest valid ticket assignment $\overrightarrow{t_{j'}}$ where $j < j' \leq j + |\texttt{deltas}|$, or $\perp$ if no such solution exists

Super Swiper(cont.)

Recursive case. If $|\texttt{deltas}| > 1$:

- split `deltas` into two halves
  - `deltas_l`, `deltas_r`
- to test left half:
  - `dp_l` $\leftarrow$ `dp_head`
  - call `dp_l.apply`$\left(w_i, \left(\vec{t_j}\right)_i\right)$ for $i \in$ `deltas_r` $\setminus$ `deltas_l`
  - call ProcessBatchRecursive$(\vec{t_j}, \texttt{deltas_l}, \texttt{dp_l})$
- to test right half:
  - $m \leftarrow j + |\texttt{deltas\_l}|$
  - `dp_r` $\leftarrow$ `dp_head`
  - call `dp_r.apply`$\left(w_i, \left(\vec{t_m}\right)_i\right)$ for $i \in$ `deltas_l` $\setminus$ `deltas_r`
  - call ProcessBatchRecursive$(\vec{t_m}, \texttt{deltas_r}, \texttt{dp_r})$

ProcessBatchRecursive($\vec{t_j}$, `deltas`, `dp_head`)

- Input:
  - $\vec{t_j}$: a ticket assignment
  - `deltas`: an array of indices that generate $\overrightarrow{t_{j+1}}, \overrightarrow{t_{j+2}}, ..., \overrightarrow{t_{j+|\texttt{deltas}|}}$
  - `dp_head`: DP data structure where $\left(w_i, \left(\vec{t_j}\right)_i\right)$ are applied exactly for $i \notin$ `deltas`
- Output:
  - the smallest valid ticket assignment $\vec{t_{j'}}$ where $j < j' \leq j + |\texttt{deltas}|$, or $\perp$ if no such solution exists

Super Swiper(cont.)

Recursive case. If $|\texttt{deltas}| > 1$:

- split `deltas` into two halves
  - `deltas_l`, `deltas_r`
- to test left half:
  - $\texttt{dp\_l} \leftarrow \texttt{dp\_head}$
  - call $\texttt{dp\_l.apply}\left(w_i, \left(\vec{t_j}\right)_i\right)$ for $i \in \texttt{deltas\_r} \setminus \texttt{deltas\_l}$
  - call ProcessBatchRecursive($\vec{t_j}$, `deltas_l`, `dp_l`)
- to test right half:
  - $m \leftarrow j + |\texttt{deltas\_l}|$
  - $\texttt{dp\_r} \leftarrow \texttt{dp\_head}$
  - call $\texttt{dp\_r.apply}\left(w_i, \left(\vec{t_m}\right)_i\right)$ for $i \in \texttt{deltas\_l} \setminus \texttt{deltas\_r}$
  - call ProcessBatchRecursive($\vec{t_m}$, `deltas_r`, `dp_r`)

ProcessBatchRecursive($\vec{t_j}$, `deltas`, `dp_head`)

- Input:
  - $\vec{t_j}$: a ticket assignment
  - `deltas`: an array of indices that generate $\vec{t_{j+1}}, \vec{t_{j+2}}, \ldots, \vec{t_{j+|\texttt{deltas}|}}$
  - `dp_head`: DP data structure where $\left(w_i, \left(\vec{t_j}\right)_i\right)$ are applied exactly for $i \notin \texttt{deltas}$
- Output:
  - the smallest valid ticket assignment $\vec{t_{j'}}$ where $j < j' \leq j + |\texttt{deltas}|$, or $\perp$ if no such solution exists

Super Swiper(cont.)

Recursive case. If $|\texttt{deltas}| > 1$:

- split `deltas` into two halves
  - `deltas_l`, `deltas_r`
- to test left half:
  - $\texttt{dp\_l} \leftarrow \texttt{dp\_head}$
  - call $\texttt{dp\_l.apply}\left(w_i, \left(\vec{t_j}\right)_i\right)$ for $i \in \texttt{deltas\_r} \setminus \texttt{deltas\_l}$
  - call ProcessBatchRecursive($\vec{t_j}$, `deltas_l`, `dp_l`)
- to test right half:
  - $m \leftarrow j + |\texttt{deltas\_l}|$
  - $\texttt{dp\_r} \leftarrow \texttt{dp\_head}$
  - call $\texttt{dp\_r.apply}\left(w_i, \left(\vec{t_m}\right)_i\right)$ for $i \in \texttt{deltas\_l} \setminus \texttt{deltas\_r}$
  - call ProcessBatchRecursive($\vec{t_m}$, `deltas_r`, `dp_r`)

ProcessBatchRecursive($\vec{t_j}$, `deltas`, `dp_head`)

- Input:
  - $\vec{t_j}$: a ticket assignment
  - `deltas`: an array of indices that generate $\vec{t_{j+1}}, \vec{t_{j+2}}, \ldots, \vec{t_{j+|\texttt{deltas}|}}$
  - `dp_head`: DP data structure where $\left(w_i, \left(\vec{t_j}\right)_i\right)$ are applied exactly for $i \notin \texttt{deltas}$
- Output:
  - the smallest valid ticket assignment $\vec{t_{j'}}$ where $j < j' \leq j + |\texttt{deltas}|$, or $\perp$ if no such solution exists

Super Swiper(cont.)

Back to batch algorithm

- goal: test $\vec{t_j}$ for $j \in (l, r]$, $l = 2^k - 1$, $r = 2^{k+1} - 1$

Running time analysis:

- DP.apply always called on $\vec{t}$ with $\text{size}(\vec{t}) < 2^{k+1}$; takes time $O(2^k)$
- $m = |\texttt{deltas}|$: ProcessBatchRecursive$(\cdot, \texttt{deltas}, \cdot)$ calls DP.apply $\leq m$ times (at current stack level)
- $T(m)$ – total # calls to DP.apply
  - $T(1) = 1$
  - $T(m) \leq 2T(m/2) + m$
- solving: $T(2^k) = O(k \cdot 2^k)$; total time $O(k \cdot 2^{2k})$

ProcessBatchRecursive($\vec{t_j}$, deltas, dp_head)

- Input:
  - $\vec{t_j}$: a ticket assignment
  - deltas: an array of indices that generate $\vec{t_{j+1}}, \vec{t_{j+2}}, ..., \vec{t_{j+|\texttt{deltas}|}}$
  - dp_head: DP data structure where $(w_i, (\vec{t_j})_i)$ are applied exactly for $i \notin$ deltas
- Output:
  - the smallest valid ticket assignment $\vec{t_{j'}}$ where $j < j' \leq j + |\texttt{deltas}|$, or $\perp$ if no such solution exists

Super Swiper(cont.)

Back to batch algorithm

- goal: test $\vec{t_j}$ for $j \in (l, r]$, $l = 2^k - 1$, $r = 2^{k+1} - 1$

Running time analysis:

- `DP.apply` always called on $\vec{t}$ with $\text{size}(\vec{t}) < 2^{k+1}$; takes time $O(2^k)$
- $m = |\texttt{deltas}|$: ProcessBatchRecursive($\cdot$, `deltas`, $\cdot$) calls `DP.apply` $\leq m$ times (at current stack level)
- $T(m)$ – total # calls to `DP.apply`
  - $T(1) = 1$
  - $T(m) \leq 2T(m/2) + m$
- solving: $T(2^k) = O(k \cdot 2^k)$; total time $O(k \cdot 2^{2k})$

ProcessBatchRecursive($\vec{t_j}$, `deltas`, `dp_head`)

- Input:
  - $\vec{t_j}$: a ticket assignment
  - `deltas`: an array of indices that generate $\vec{t_{j+1}}, \vec{t_{j+2}}, \ldots, \vec{t_{j+|\texttt{deltas}|}}$
  - `dp_head`: DP data structure where $(w_i, (\vec{t_j})_i)$ are applied exactly for $i \notin$ `deltas`
- Output:
  - the smallest valid ticket assignment $\vec{t_{j'}}$ where $j < j' \leq j + |\texttt{deltas}|$, or $\perp$ if no such solution exists

Super Swiper(cont.)

Back to batch algorithm

- goal: test $\vec{t_j}$ for $j \in (l, r]$, $l = 2^k - 1$, $r = 2^{k+1} - 1$

Running time analysis:

- `DP.apply` always called on $\vec{t}$ with $\text{size}(\vec{t}) < 2^{k+1}$; takes time $O(2^k)$
- $m = |\texttt{deltas}|$: ProcessBatchRecursive($\cdot$, `deltas`, $\cdot$) calls `DP.apply` $\leq m$ times (at current stack level)
- $T(m)$ – total # calls to `DP.apply`
  - $T(1) = 1$
  - $T(m) \leq 2T(m/2) + m$
- solving: $T(2^k) = O(k \cdot 2^k)$; total time $O(k \cdot 2^{2k})$

ProcessBatchRecursive($\vec{t_j}$, `deltas`, `dp_head`)

- Input:
  - $\vec{t_j}$: a ticket assignment
  - `deltas`: an array of indices that generate $\vec{t_{j+1}}, \vec{t_{j+2}}, ..., \vec{t_{j+|\texttt{deltas}|}}$
  - `dp_head`: DP data structure where $(w_i, (\vec{t_j})_i)$ are applied exactly for $i \notin$ `deltas`
- Output:
  - the smallest valid ticket assignment $\vec{t_{j'}}$ where $j < j' \leq j + |\texttt{deltas}|$, or $\perp$ if no such solution exists

Super Swiper(cont.)

Back to batch algorithm

- goal: test $\vec{t_j}$ for $j \in (l, r]$, $l = 2^k - 1$, $r = 2^{k+1} - 1$

Running time analysis:

- `DP.apply` always called on $\vec{t}$ with $\text{size}(\vec{t}) < 2^{k+1}$; takes time $O(2^k)$
- $m = |\texttt{deltas}|$: ProcessBatchRecursive($\cdot$, `deltas`, $\cdot$) calls `DP.apply` $\leq m$ times (at current stack level)
- $T(m)$ – total # calls to `DP.apply`
  - $T(1) = 1$
  - $T(m) \leq 2T(m/2) + m$
- solving: $T(2^k) = O(k \cdot 2^k)$; total time $O(k \cdot 2^{2k})$

ProcessBatchRecursive($\vec{t_j}$, `deltas`, `dp_head`)

- Input:
  - $\vec{t_j}$: a ticket assignment
  - `deltas`: an array of indices that generate $\vec{t_{j+1}}, \vec{t_{j+2}}, ..., \vec{t_{j+|\texttt{deltas}|}}$
  - `dp_head`: DP data structure where $(w_i, (\vec{t_j})_i)$ are applied exactly for $i \notin$ `deltas`
- Output:
  - the smallest valid ticket assignment $\vec{t_{j'}}$ where $j < j' \leq j + |\texttt{deltas}|$, or $\perp$ if no such solution exists

Super Swiper(cont.)

Back to batch algorithm

- goal: test $\vec{t_j}$ for $j \in (l, r]$, $l = 2^k - 1$, $r = 2^{k+1} - 1$

Running time analysis:

- `DP.apply` always called on $\vec{t}$ with $\text{size}(\vec{t}) < 2^{k+1}$; takes time $O(2^k)$
- $m = |\texttt{deltas}|$: ProcessBatchRecursive($\cdot$, `deltas`, $\cdot$) calls `DP.apply` $\leq m$ times (at current stack level)
- $T(m)$ – total # calls to `DP.apply`
  - $T(1) = 1$
  - $T(m) \leq 2T(m/2) + m$
- solving: $T(2^k) = O(k \cdot 2^k)$; total time $O(k \cdot 2^{2k})$

ProcessBatchRecursive($\vec{t_j}$, `deltas`, `dp_head`)

- Input:
  - $\vec{t_j}$: a ticket assignment
  - `deltas`: an array of indices that generate $\vec{t_{j+1}}, \vec{t_{j+2}}, ..., \vec{t_{j+|\texttt{deltas}|}}$
  - `dp_head`: DP data structure where $(w_i, (\vec{t_j})_i)$ are applied exactly for $i \notin$ `deltas`
- Output:
  - the smallest valid ticket assignment $\vec{t_{j'}}$ where $j < j' \leq j + |\texttt{deltas}|$, or $\perp$ if no such solution exists

Fabrice Benhamouda, Shai Halevi, and Lev Stambler.
Weighted secret sharing from wiretap channels.
In Kai-Min Chung, editor, *ITC 2023*, volume 267 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl, June 2023.
doi:10.4230/LIPIcs.ITC.2023.8.

Pyrros Chaidos, Aggelos Kiayias, Leonid Reyzin, and Anatoliy Zinovyev.
Approximate lower bound arguments.
In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part IV*, volume 14654 of *LNCS*, pages 55–84. Springer, Cham, May 2024.
doi:10.1007/978-3-031-58737-5_3.

Sourav Das, Benny Pinkas, Alin Tomescu, and Zhuolun Xiang.
Distributed randomness using weighted VUFs.
In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part VII*, volume 15607 of *LNCS*, pages 314–344. Springer, Cham, May 2025.
doi:10.1007/978-3-031-91098-2_12.

Hanwen Feng, Tiancheng Mai, and Qiang Tang.

Scalable and adaptively secure any-trust distributed key generation and all-hands checkpointing.
In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 2636–2650. ACM Press, October 2024.
doi:10.1145/3658644.3690253.


Peter Gazi, Aggelos Kiayias, and Alexander Russell.
Fait accompli committee selection: Improving the size-security tradeoff of stake-based committees.
In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 845–858. ACM Press, November 2023.
doi:10.1145/3576915.3623194.


Andrei Tonkikh and Luciano Freitas.
Swiper: a new paradigm for efficient weighted distributed protocols.
Cryptology ePrint Archive, Report 2023/1164, 2023.
URL: https://eprint.iacr.org/2023/1164.