# Weight reduction in distributed protocols: new algorithms and analysis

Anatoliy Zinovyev
Boston University
https://cs-people.bu.edu/tolik

**Abstract**

We study the problem of minimizing the total weight of (potentially many) participants of a distributed protocol, a necessary step when the original values are large but the scheme to be deployed scales poorly with the weights. We assume that $\alpha$ fraction of the original weights can be corrupted and we must output new weights with at most $\beta$ adversarial fraction, for $\alpha < \beta$. This problem can be viewed from the prism of electing a small committee that does the heavy work, a powerful tool for making distributed protocols scalable. We solve the variant that requires giving parties potentially multiple seats in the committee and counting each seat towards the cost of the solution. Moreover, we focus on the "deterministic" version of the problem where the computed committee must be secure *for any* subset of parties that can be corrupted by the adversary; such a committee can be smaller than a randomly sampled one in some cases and is useful when security against adaptive corruptions is desired but parties in the sub-protocol speak multiple times.

Presented are new algorithms for the problem as well as analysis of prior work. We give two variants of the algorithm Swiper (PODC 2024) that significantly improve the running time without sacrificing the quality of the output, with the second one, in fact, improving the output. We prove, however, that all known algorithms, including our two variants of Swiper, have worst case approximation ratio $\Omega(n)$. To counter that, we give the first polynomial time algorithm with approximation factor $n/\log^2 n$ and also the first sub-exponential time exact algorithm, practical for some real-world inputs. Of theoretical interest is another polytime algorithm that we present, based on linear programming, whose output is no worse than an optimal solution to the problem with slightly different parameters.

We implemented and tested previous and new algorithms, comparing them on the stake distributions of popular proof-of-stake blockchains, and found that our second variant of Swiper computes solutions extremely close to the optimal, confirmed by our exact algorithm.

## 1 Introduction

Committee selection is an essential tool for making large-scale distributed protocols scalable. The idea is that, instead of all $n$ parties doing the work, the protocol is run

within a smaller committee, decided by the larger protocol; the participants outside the committee simply learn the result. To spread the work evenly or to achieve better security, multiple different committees can be created.

We are mainly interested in byzantine fault tolerant / cryptographic protocols where parties have some assigned weights $w_i$ and the adversary can corrupt parties up to some $\alpha$ fraction of the total weight in the system. For example, in a proof-of-stake blockchain parties own some amount of money in the system (stake) and the assumption is that the adversary controls, say, at most 20% of all the stake. We would like to compute a "small" committee where the adversary controls less than some larger $\beta$ fraction, say 1/3, of the total weight. Specifically, our goal is to assign each party a new non-negative integer weight $t_i$ such that we minimize the total new weight $\sum_{i=1}^{n} t_i$. This is useful when the protocol in question doesn't scale well with the weights such as in secret sharing, threshold signatures or secure multiparty computation. Additionally, we focus on "deterministic" committee selection that must output a committee secure against *any* set of parties that the adversary is allowed to corrupt. Formally, the new weights $t_1, ..., t_n$ must satisfy

$$\forall A \subseteq [n] \text{ with } \sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i, \ \sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i$$

(here $A$ denotes the set of parties controlled by the adversary). This property is useful when one requires security against an adversary that performs adaptive corruptions throughout the protocol execution with full knowledge of $(t_1, ..., t_n)$ – for example, when the members of the committee need to speak multiple times and thus reveal themselves after the first message. Moreover, committees of this flavor can be smaller than traditional "randomized" committees, as explained in section 2. As demonstrated in [TdS24], they also yield a convenient way to transform a wide range of unweighted protocols to their weighted version. We note that "deterministic" committee selection cannot be usefully instantiated in the unweighted setting since any committee would have to have $\Omega(n)$ members (otherwise, the adversary can corrupt the whole committee). In contrast, when parties have weights distributed non-uniformly, we can often compute a committee with the total new weight much smaller than $n$.

## 1.1 Prior work

The "deterministic" committee selection problem has a short history. To the best of our knowledge, the recent Swiper paper [TdS24] by Tonkikh and Freitas was the first and only to study it in isolation. Near-simultaneously, other works [BHS23, FMT24, DPTX25] attempted to solve it in the context of their larger protocols. All four papers consider different variants of the problem although all aim to minimize the total new weight $\sum_{i=1}^{n} t_i$.

Swiper defines a problem they call weight restriction that is only slightly different from the one we consider: given $w_1, ..., w_n$ as input, output $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that for all $A \subseteq [n]$ with $\sum_{i \in A} w_i < \alpha \sum_{i=1}^{n} w_i$, we have $\sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i$. The only distinction is that the inequality containing $\alpha$ is non-strict in our paper.

The authors of Swiper prove that the problem admits an efficiently computable solution of cost $\sum_{i=1}^{n} t_i \leq O(n)$ (assuming $\alpha$ and $\beta$ parameters are constant). Such a solution can be found by linearly scaling down the vector $(w_1, ..., w_n)$ and rounding each $w_i$ to an integer according to a simple rule. Swiper goes a step further and tries to compute a solution as small as possible by running a binary search on the scaling factor and testing the correctness of a vector $(t_1, ..., t_n)$ at each iteration. Its time complexity is $O(n^2 \log n)$.

We note that Swiper defines two more weight reduction problems. Their overview and the overview of the other works [BHS23, FMT24, DPTX25] that consider deterministic committee selection can be found in Appendix A.

## 1.2 Our contribution and paper outline

Section 2 gives background information on committee selection and how the problem we consider relates to other variants. In section 3 we formally define the problem and notation. Section 4 provides a detailed overview of the Swiper algorithm, useful for reading the following section.

Section 5 presents our new variants of Swiper called faster Swiper and super Swiper. The first variant improves the time complexity of Swiper from $O(n^2 \log n)$ to $O(n + R^2)$ where $R \leq O(n)$ is the total weight in the output. This improves the running time by a $\log n$ factor in the worst case, but given that the cost of the output $R$ is often much smaller than $n$, the improvement is much larger as demonstrated by our numerical evaluations. For low latency applications, our optimizations would be highly desired. Importantly, faster Swiper doesn't alter the quality of the output of Swiper. Our second variant super Swiper improves the output quality by replacing the binary search with a linear search. Although the idea is simple, the algorithm has non-trivial time complexity of $O(n + R^2 \log R)$. Numerical evaluations demonstrate a sizable reduction in the total new weight over Swiper and faster Swiper, since the two can get stuck in a local minimum; in fact, the cost of super Swiper's output is extremely close to optimal in our tests. At the same time, its running time is similar to that of faster Swiper. Both variants rely on two optimizations: 1) carefully bounding the running time in terms of the output cost $R$ and 2) reusing dynamic programming computations.

We, however, prove in section 6 that all previously known algorithms, as well as our two Swiper variants, suffer from $\Omega(n)$ approximation ratio in the worst case. Our lower bound applies to a class of linear scaling based solutions which contains all but one known algorithm (which is also based on linear scaling but requires a different argument). Despite super Swiper being a great "heuristic" scheme, we would still like to find an algorithm with better theoretical guarantees. We make some progress with this in subsequent sections.

Section 7 proves several facts about our weight reduction problem useful in the rest of the paper. Of particular importance is a new observation that assuming the weights $w_1, ..., w_n$ in the input are sorted, there exists an optimal solution $t_1, ..., t_n$ that is also sorted. In section 8 we show that this fact yields the first polynomial time algorithm with approximation factor $n/\left(c \log^2 n\right)$, for any $c > 0$, and the first exact (approximation factor 1) solution with time complexity $O\left(n + R^{3/2} e^{C\sqrt{R}}\right)$,

where $R$ is the cost of an optimal solution and $C \approx 2.57$. Given that $R$ is often much smaller than $n$, our exact algorithm turns out to be practical for some real-world weight distributions, as shown by numerical evaluations. We believe it makes sense to use it in production when the optimal solution is expected to be small, in combination with a fallback scheme if the exact algorithm takes too long. It can also be very useful for research purposes, for example, to test some hypothesis about the problem.

Section 9 describes a polynomial time algorithm based on the standard approach of representing an optimization problem as an integer linear program, relaxing it to allow fractional solutions, running an LP solver and rounding the solution to an integer vector in a problem specific manner. When denoting the cost of an optimal solution by $\mathsf{OPT}_{\leq\alpha,<\beta}$, our algorithm guarantees a solution of cost at most $\mathsf{OPT}_{\leq\alpha,<(1-\delta)\beta}$ for any arbitrarily small constant $\delta > 0$. We, however, prove in Appendix B that this can be $\Omega(n)$ times larger than the optimal solution. Nevertheless, we believe the algorithm presents theoretical interest useful for future research. Moreover, such a guarantee can be meaningful when the protocol relying on committee selection can work with a continuous range of the final adversarial fractions $\beta$. Take for example Approximate Lower Bound Arguments [CKRZ24] that allow one to create short threshold signatures. Its decentralized version uses committee selection and presents a configurable tradeoff between the size of the committee and the size of the final proof (e.g., a threshold signature) where the proof size would be some monotone function of $\beta$ in our notation. Their committee selection algorithm is agnostic to the weight distribution, but suppose we replace it with theoretically smallest committees for any given $\beta$ and plot the resulting tradeoff with the committee size on the horizontal axis and the proof size on the vertical axis. We would like to achieve this tradeoff, but since the optimal committee selection is likely infeasible, we are forced to use an approximation and get a worse curve. Using algorithms approximating the committee size for a fixed $\beta$, such as the one in subsection 8.2, would shift the tradeoff graph to the right while algorithms approximating $\beta$ for a fixed committee size, such as the one in section 9, would shift the graph up. In the absence of a good approximation on the committee size, the second transformation could even be superior.

Finally, section 10 contains numerical evaluations comparing old and new algorithms in terms of the total weight in the output and the running time. It confirms our claims about the improved running time of faster Swiper and the improved output quality of super Swiper. It also proves our sub-exponential time exact algorithm to be somewhat practical.

## 2 General overview of committee selection

The idea of selecting small committees can be seen as far back as [Bra85]. It has since been used in constructing many efficient protocols for consensus [KS09, KS11, KLST11, BSGH13, BCG24]. For example, the proof-of-stake blockchain protocol Algorand [GHM+17, CM19] modifies a classical setting $O(n^2)$ byzantine fault tolerant consensus protocol to run on committees created by selecting each unit of stake with some probability, which makes communication complexity per party indepen-

dent of the number of parties. Additionally, each step of the BFT sub-protocol uses a newly sampled anonymous committee that reveals itself only when the members of the committee speak, thus achieving security against an adaptive adversary. Such an adversary can corrupt participants at any time of the protocol based on the information he possesses thus far, up to some threshold of total corrupted stake.

Committee selection can also be seen in communication efficient threshold signatures [CK24, CKRZ24, FHASW24], leader election [KSSV06, KKK+10], secure multiparty computation [BGT13, BCP15, DKM+17, BCDH23, CHOR22], and distributed key-generation [CS04].

Schemes utilizing committees typically possess a gap between the absolutely necessary requirement on the fraction of adversarial power and the smaller fraction assumed by the protocol. For instance, solving consensus in an asynchronous environment requires adversarial weight to be strictly less than 1/3 of the total weight while Algorand assumes that at most 1/5 of the total stake is controlled by the adversary. Motivated by proof-of-stake blockchains, we are interested in constructing small committees in the *weighted* setting. Given original weights $w_1, w_2, ..., w_n \in \mathbb{N}$ for $n$ parties such that the adversary controls parties of total weight at most $\alpha \sum_{i=1}^{n} w_i$, we need to compute new weights $t_1, t_2, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that the adversary can control strictly less than $\beta \sum_{i=1}^{n} t_i$ of new weight.

There are at least two dimensions of interesting variants of this problem. One is the definition of the objective function. A party in Algorand sends out one vote per step of the protocol even if it has multiple units of stake on the committee. Thus, for Algorand, we are interested only in the number of distinct parties on the committee or, equivalently, the number of non-zero $t_i$. On the other hand, there are schemes that cannot efficiently aggregate weights, say Shamir secret sharing. For such protocols, it may be necessary to use "virtualization" – essentially, giving $w$ virtual identities to a party with weight $w \in \mathbb{Z}_{\geq 0}$ and running this many copies of the protocol. This can be prohibitively expensive when weights are large (e.g., in proof-of-stake blockchains weights can be on the order of $2^{64}$), and thus, for such protocols we are interested in constructing committees with $\sum_{i=1}^{n} t_i$ as small as possible – the goal we pursue in this paper. We note that there have been many works trying to make cryptographic protocols scale better with the weights [BW06, BHS23, GJM+24, DCX+23, CK24, CKRZ24, FHASW24]. Still, protocols such as those in [DPTX25] only improve the dependence on weights for the verifiers while the provers' running time is still linear in the weights, and protocols such as those in [GJM+23] only improve complexities to $O(W)$ down from $O(W \cdot \lambda)$, for a total weight $W$ and security parameter $\lambda$.

Another configuration knob for the problem of committee selection is, what kind of security do we require? Traditionally, especially in a setting where the adversary is static, meaning he chooses parties to corrupt before the start of the protocol, a random committee is chosen such that the (unknown) corrupted set of parties owns only a small fraction of the elected weight. More formally, the security requirement

of this "randomized" variant is that

$$\begin{aligned} &\forall (w_1, ..., w_n),\ \forall A \subseteq [n] \text{ with } \sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i, \\ &\Pr\left[ \sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i : (t_1, ..., t_n) \leftarrow \text{ComputeCommittee}(w_1, ..., w_n) \right] \text{ is large} \end{aligned} \tag{1}$$

(we assume that $\alpha$ and $\beta$ are constants). Instead, this work considers a stronger security notion where the computed committee must be secure *for any* subset of the parties that can be corrupted. A more formal statement of this "deterministic" version of the problem is as follows.

$$\begin{aligned} &\forall (w_1, ..., w_n), \quad \text{if } (t_1, ..., t_n) = \text{ComputeCommittee}(w_1, ..., w_n) \text{ then} \\ &\forall A \subseteq [n] \text{ with } \sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i,\ \sum_{i \in A} t_i < \beta \sum_{i=1}^{n} t_i \end{aligned} \tag{2}$$

(where ComputeCommittee is now deterministic). When the weaker security notion is sufficient, it is great news since selecting each unit of weight with some equal probability is a simple and efficient scheme yielding an expected constant size committee; this is generally impossible for a deterministic algorithm (since $n$ parties with equal weights must elect a committee of size $\Omega(n)$, lest the entire committee be corrupted). On the other hand, deterministic committee selection makes it easy to maintain security against an adversary who is allowed to corrupt parties adaptively throughout the protocol execution since any corrupted set will have less than $\beta$ fraction of new weight in the committee. While simple quorum systems can get away with using random committees with VRF as a source of secret randomness, other problems are less trivial and require intricate protocols and arguments of security [CM19, GHK+21, CDGK22, CKR+23, ACD+25]. The security against adaptive corruptions has been recognized important by the community, and in fact, is the reason why [DPTX25, DPTX24] has been deployed in the Aptos blockchain in combination with a deterministic committee selection scheme (from private communication with the authors). Moreover, there is evidence that a deterministic scheme can produce smaller committees than a randomized one. Take, for example, the standard randomized procedure of selecting each unit of weight with some equal probability. As the requirement on how large the probability in Equation 1 must be increases, the expected number of selected units of weight approaches infinity and the expected number of elected parties approaches $n$. Thus, for a sufficiently large mandated success probability, the chosen committee would be worse than that of a deterministic algorithm. Another example is the randomized schemes presented in [GKR23] called Fait Accompli that include larger parties in the committee deterministically to either reduce the number of parties in the committee or their total new weight. From their formulas and plots, one can see that as the required probability of failure decreases, the committee must grow. Indeed, it seems inherent that when we require $\sum_{i \in A} t_i$ in Equation 1 to be increasingly concentrated, its expectation must approach infinity; it would be a good exercise to prove that as the probability of

failure goes to 0, whenever at least one $t_i$ is non-deterministic, $E\left[\sum_{i=1}^{n} t_i\right]$ must approach infinity and $E\left[|\{i : t_i > 0\}|\right]$ must approach or exceed the number of parties in an optimal deterministic committee.

In our work, we focus on the "deterministic" flavor of committee selection with the objective to minimize the total weight $\sum_{i=1}^{n} t_i$, and believe this problem is important for the reasons outlined above. Our paper can be viewed as a continuation of the Fait Accompli work [GKR23] by Gaži, Kiayias and Russell and the Swiper work [TdS24] by Tonkikh and Freitas that frame committee selection as an optimization problem with parties' weights in the input. Fait Accompli shows randomized schemes that optimize either $\sum_{i=1}^{n} t_i$ or $|\{i : t_i > 0\}|$, while Swiper describes algorithms for the same problem we consider, and in fact, we directly extend some of their results. Despite both works numerically showing impressive results on real-world, far from uniform, weight distributions, they say nothing about the quality of their algorithms' output compared to the best possible solution. The golden standard in approximation algorithms is to provide one that yields solutions of the cost within a certain multiplicative factor of the cost of an optimal solution [WS11], or otherwise bounded by some other "interesting" function of the input (as in [HKZ24] for ECSS). We make some progress towards this.

Finally, we note that there is no reason to require algorithms satisfying the stronger security notion in Equation 2 to be deterministic. The use of randomness in optimization often helps [WS11], but all algorithms in this paper are in fact deterministic.

# 3 Problem definition

Let $n \in \mathbb{N}$ be the number of parties, $w_1, ..., w_n \in \mathbb{N}$ be their (original) weights, and let $\alpha$ and $\beta$ be constants independent of $n$ with $0 < \alpha < \beta < 1$. Borrowing notation from [TdS24], for the new weights $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$, we say party $i$ gets $t_i$ tickets and call $(t_1, ..., t_n)$ a ticket assignment. We call $(t_1, ..., t_n)$ valid if and only if $\sum_{i \in I} t_i < \beta \sum_{i=1}^{n} t_i$ whenever $I \subseteq [n]$ and $\sum_{i \in I} w_i \leq \alpha \sum_{i=1}^{n} w_i$. Our goal is to find a valid ticket assignment that minimizes the total number of tickets $\sum_{i=1}^{n} t_i$. We call this the $(\leq \alpha, < \beta)$-weight reduction problem and denote $\mathsf{OPT}_{\leq \alpha, < \beta}((w_1, ..., w_n))$ to be the smallest total number of tickets $\sum_{i=1}^{n} t_i$ in a valid solution $(t_1, ..., t_n)$. We can also formulate the problem using the following integer linear program.

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

We note that the formulation in [TdS24] uses a strict inequality where $\alpha$ is involved. For simplicity, especially when dealing with linear programming, we use a non-strict inequality. However, we still keep a strict inequality where $\beta$ is involved

since we are typically interested in solutions where the adversary controls strictly less than $1/2$ or $1/3$ of the total number of tickets and formulating the problem using a strict inequality simplifies asymptotic analysis where $n$ approaches infinity. We call the variant in [TdS24] the $(<\alpha, <\beta)$-weight reduction problem with optimum $\mathsf{OPT}_{<\alpha,<\beta}((w_1, ..., w_n))$, and also similarly define $(\leq\alpha, \leq\beta)$- and $(<\alpha, \leq\beta)$-weight reduction problems that have an additional requirement $\sum_{i=1}^{n} t_i \geq 1$.

We also note that we do not allow zero weights $w_i$ in the input, but it does not change the problem since it is always optimal to set $t_i \coloneqq 0$ for such $i$.

# 4 Detailed overview of Swiper

We now give a more detailed overview of the Swiper algorithm from [TdS24, TF23] since we will be modifying it later in section 5.

It works by linearly scaling down the weights and rounding to an integer according to a rule. Specifically, given weights $w_1, ..., w_n$ as input, Swiper outputs $t_1, ..., t_n$ such that for all $i$, either $t_i = \lfloor sw_i + c \rfloor$ or $t_i = \lfloor sw_i + c \rfloor - 1$, where $c$ is a constant to be defined later. Swiper first runs a binary search on $s$ to find a local minimal value for which the ticket assignment with all $t_i = \lfloor sw_i + c \rfloor$ is valid. Then the algorithm considers those $i$ for which $(sw_i + c)$ is an integer, or equivalently those $i$ for which $t_i$ decreases when $s$ is decreased by any value; call this borderline set of indices $B$. Finally, Swiper runs a binary search on $k \in \mathbb{N}$ to find a local minimal value in $\left[|B|\right]$ for which the ticket assignment remains valid when all but $k$ arbitrary indices from $B$ lose one ticket.

From a slightly different perspective, Swiper defines a set of potential solutions $\overrightarrow{t_1}, \overrightarrow{t_2}, ...$ with $\sum_{i=1}^{n} \left(\overrightarrow{t_j}\right)_i = j$ and finds a local minimal $j$ for which $\overrightarrow{t_j}$ is a correct solution but $\overrightarrow{t_{j-1}}$ is not.

The authors observe that starting some index $M \in O(n)$, all $\overrightarrow{t_M}, \overrightarrow{t_{M+1}}, ...$ are valid. In particular, when $c = \alpha$, we have

$$M = \left\lfloor \frac{\alpha(1-\alpha)}{\beta - \alpha} n + 1 \right\rfloor \tag{3}$$

(for the $(\leq\alpha, <\beta)$ weight reduction variant of the problem). By making $M$ the right threshold of the search, Swiper always outputs a solution with at most this many tickets, though a much better solution is usually found for real-world weight distributions.

Each iteration of a binary search needs to test whether some ticket assignment $(t_1, ..., t_n)$ is valid. This is done precisely by solving the knapsack problem where weights are $w_i$ and profits are $t_i$. Since Swiper only tests ticket assignments of size at most $M = O(n)$, the knapsack problem can be solved in time $O\left(n \cdot \sum_{i=1}^{n} t_i\right) = O\left(n^2\right)$ using dynamic programming that for each $0 \leq t \leq \sum_{i=1}^{n} t_i$ computes the smallest possible weight that achieves $t$ tickets. The total time complexity of Swiper is $O(n^2 \log n)$.

# 5 Improving Swiper

In this section we present two variants of Swiper, both of which improve the running time of the algorithm, while the second also improves the quality of the output.

## 5.1 Faster Swiper

We first show a new algorithm named faster Swiper that improves the asymptotic running time of the original Swiper algorithm without semantically changing the output. Like the original Swiper, faster Swiper finds a ticket assignment $\vec{t}$ such that

- $\vec{t}$ is valid, equals a linear scaling of the weights according to Swiper rules (see section 4), and $\sum_i \left(\vec{t}\right)_i$ satisfies the $O(n)$ bound of the original Swiper;
- there exists a ticket assignment $\vec{t'}$ with one fewer ticket such that it equals a linear scaling of the weights according to Swiper rules but is not valid.

The stated time complexity of Swiper in [TdS24] is $O(n^2 \log n)$ where $n$ is the number of weights in the input. Our algorithm faster Swiper has time complexity $O(n + R^2)$ where $R$ is the cost of the output (i.e., if the output is $\vec{t}$, $R = \sum_i \left(\vec{t}\right)_i$). Note that $R$ is always $O(n)$ but at the same time, $R$ can be much smaller than $n$ in practice, and thus, looking at the asymptotic complexity in terms of two variables $n$ and $R$ can be useful.

It will be convenient to have access to the input weights vector sorted in the descending order. Let us assume such sorted input $(w_1, w_2, ..., w_n)$. We first explain how to achieve running time $O(R^2 \log R)$ and then improve it further to $O(R^2)$. In subsection 5.3 we will explain how to remove the assumption.

### 5.1.1 $O(R^2 \log R)$ time complexity

For all $i \geq 0$, let $\vec{t_i}$ be the vector with $i$ tickets in total constructed according to Swiper rules such that for any $i$, $\vec{t_i}$ and $\vec{t_{i+1}}$ have the same values for all indices except for one; when multiple indices could be incremented in the transition from $\vec{t_i}$ to $\vec{t_{i+1}}$, choose the lowest index. The (uniquely defined) sequence $\left\{\vec{t_i}\right\}$ has a property that if $\left(\vec{t_i}\right)_j > 0$ then $\left(\vec{t_i}\right)_k > 0$ for all $k \leq j$. This in turn implies that if the cost of $\vec{t_i}$ is $c$, only the first $c$ indices of $\vec{t_i}$ can be non-zero. Thus, if an algorithm only ever considers ticket assignments of cost at most $O(R)$, it only needs to query the first $O(R)$ elements of the (sorted) weight vector.

We make use of this fact. To get time complexity $O(R^2 \log R)$, we run binary search on $i$ to find a value for which $\vec{t_i}$ is a valid ticket assignment but $\vec{t_{i-1}}$ is not, roughly as follows.

$l \leftarrow 0$; $r \leftarrow 1$;
**while** $\vec{t_r}$ is not valid **do**
    $l \leftarrow r$; $r \leftarrow 2 \cdot r$;
**while** $r - l > 1$ **do**
    $m \leftarrow (l + r)/2$;
    **if** $\vec{t_m}$ is valid **then**

```
        r ← m;
    else
        l ← m;
return t_r;
```

This requires $O(\log R)$ iterations of binary search, unlike in the original Swiper that does binary search on the scaling factor $s$ potentially requiring $\Omega(\log n)$ iteration. Each iteration needs to solve a knapsack problem using dynamic programming to determine whether some $\vec{t_i}$ is valid. The time required to do this is at most the total number of tickets times the number of non-zero entries in $\vec{t_i}$, since the zero entries cannot help the adversary. Both are at most $i$. Hence, the first loop will run in time $O(R^2)$ and the second loop in time $O(R^2 \log R)$.

We are left to describe how to efficiently calculate $\vec{t_0}, \vec{t_1}, ..., \vec{t_{2R}}$. Recall that these vectors are defined using the formula $\vec{t} = s\vec{w} + c$ where $s$ is a variable scaling factor, with the exception that some indices might be further decreased by one (see section 4). We use a mutable data structure called gen with a single method gen.next() such that if the $i$-th invocation of gen.next() returns index, then $(\vec{t_i})_{\text{index}} = (\vec{t_{i-1}})_{\text{index}} + 1$ (and all other indices stay the same). Such a data structure can be constructed using a binary heap as a priority queue defining the order of the produced indices by gen.next(): for an index index, the heap contains the smallest scaling factor $s$ that would increment this index. The heap initially contains a value $s$ only for the first index – the index of the largest weight. When gen.next() outputs some index, this index is reinserted in the heap; if index has never been produced before, index+1 is additionally inserted. Therefore, during the $k$-th call to gen.next(), the heap contains at most $k$ elements and thus, the call takes time $O(\log k)$. Our Swiper variant needs to query gen at most $2R$ times, and hence, will add $O(R \log R)$ to the time complexity.

### 5.1.2 $O(R^2)$ time complexity

To improve the running time of faster Swiper further, we notice that iterations of the binary search can reuse some of the dynamic programming (DP) computations. The structure of the binary search remains the same as in the $O(R^2 \log R)$ version, and we only need to optimize its main phase (the second while loop) since the initial phase already runs in time $O(R^2)$.

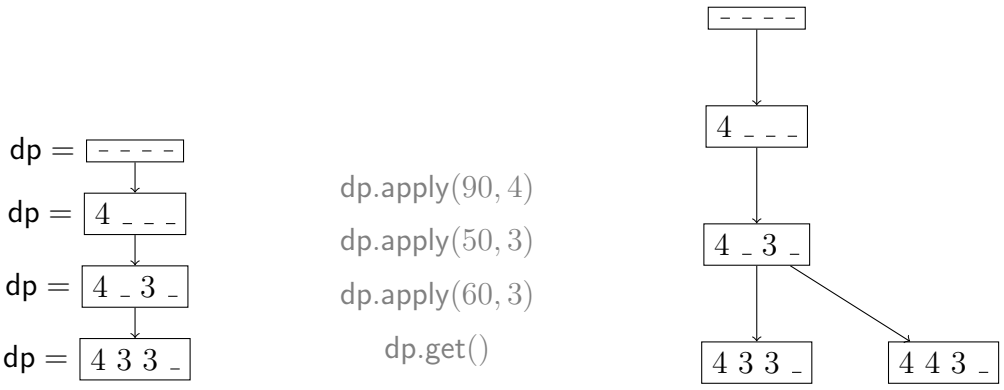
We need a data structure called DP containing (partial) DP computations for the knapsack problem. Specifically, it will have methods DP.apply$(w, t)$ and DP.get()

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

Figure 1: Methods on DP data structure

such that after running DP.apply on a sequence $(w_1, t_1), ..., (w_k, t_k)$, DP.get() reports the maximum number of tickets that the adversary can get $\sum_{j \in S} t_j$ provided $S \subseteq [k]$ and $\sum_{j \in S} w_j \leq \alpha W$ ($W = \sum_{i=1}^{n} w_i$ being the total number of tickets in the input).

Thus, in order to test a ticket assignment $\vec{t}$, one needs to call DP.apply$\big(w_i, (\vec{t})_i\big)$ once for all $i \in [n]$ with $(\vec{t})_i \neq 0$ *in any order*, followed by a call to DP.get(). Additionally, one can reduce the number of applies when testing multiple ticket assignments by reusing some of the intermediate states of DP. See Figure 2 for illustration.




(a) Testing a ticket assignment $(4, 3, 3, 0)$ with weights $(90, 60, 50, 10)$

(b) Testing two ticket assignments

Figure 2: Using DP

It can be seen that the $k$-th invocation DP.apply$(w_k, t_k)$ will take time $O\big(\sum_{i=1}^{k} t_i\big)$, and we can assume that DP.get() takes time $O(1)$ since DP.apply() on each invocation can memorize the maximum number of tickets the adversary gets. For faster Swiper, $O\big(\sum_{i=1}^{k} t_i\big) \leq O(R)$ always since the binary search only tests ticket assignments $\vec{t_i}$ of cost at most $2R$,.

We will reuse dynamic programming computation to have each iteration of the binary search (second loop) run in time $O\big(R(r-l)\big)$. Since $(r-l)$ halves each iteration, the entire algorithm will be $O(R^2)$.

The loop will have the following invariants holding at the beginning of the loop.

- tickets_l: array containing the non-zero prefix of $\vec{t_l}$;
- deltas: array containing the indices from gen.next() to generate $\overrightarrow{t_{l+1}}, ..., \vec{t_r}$;
- dp_head: DP data structure with applied $\left(w_i, \left(\vec{t_l}\right)_i\right)$ for all $i \notin$ deltas.

Essentially, dp_head will have indices applied whose values are not going to change anymore. Then testing the validity of $\vec{t_m}$ requires computing it in time $O(R)$ and applying the missing indices, contained in deltas, to a copy of dp_head in time $O(R(r-l))$. After that, the loop iteration needs to update the variables to maintain the invariants: tickets_l in time $O(R)$, deltas in constant time and dp_head in time $O(R(r-l))$ by applying a subset of indices from deltas.

## 5.2 Super Swiper

We will now present a variant of Swiper that we call super Swiper which replaces the binary search on the number of tickets in the solution with a linear search.

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

The linear search sometimes significantly improves the output in practice (see section 10) since the binary search can get stuck in a "local minimum". Although the idea is trivial, our contribution is an algorithm that runs the linear search in time $O(R^2 \log R)$ instead of the easy $O(R^3)$, assuming the input weights are sorted.

The idea is to reuse DP computations as in subsection 5.1. We check the validity of $\overrightarrow{t_j}$ in batches where batches are split further into smaller batches recursively. The batch number $k \geq 0$ will be $\left(\overrightarrow{t_{j_k+1}}, ..., \overrightarrow{t_{j_{k+1}}}\right)$ where $j_0 = 0$ and $j_{k+1} = j_k + 2^k$. The batch $k$ will be processed in time $O\left(k \cdot 2^{2k}\right)$, and thus the whole algorithm will run in time $O(R^2 \log R)$.

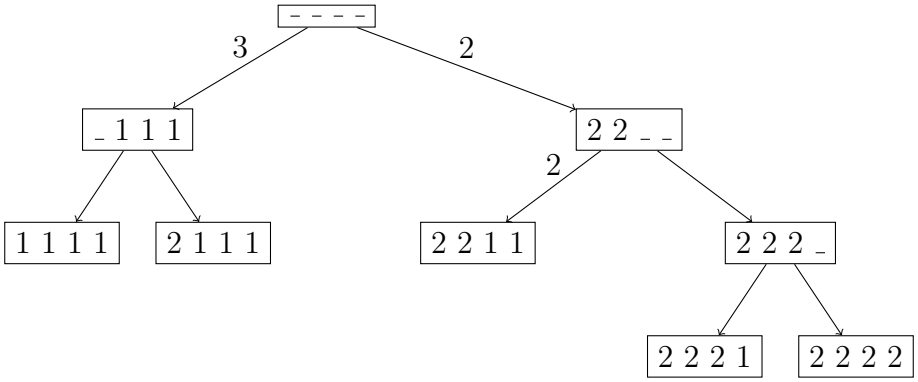



Figure 3: Recursive linear search example

The batch algorithm organizes all validity tests in a binary tree as shown in Figure 3; we now describe it. For batch $k$, collect indices from gen.next() that generate $\overrightarrow{t_{j_k+2}}, ..., \overrightarrow{t_{j_{k+1}}}$ into an array deltas. Create a DP data structure (described in subsubsection 5.1.2) dp_head and call dp_head.apply$\left(w_i, \left(\overrightarrow{t_{j_k+1}}\right)_i\right)$ for all $i \notin$ deltas; this does at most nnz $\leq T$ applies each taking time $O(T)$ where nnz $= \left|\left\{i \in [n] : \left(\overrightarrow{t_{j_k+1}}\right)_i > 0\right\}\right|$ and $T = \sum_{i=1}^{n} \left(\overrightarrow{t_{j_k+1}}\right)_i = j_k + 1 = O(2^k)$, for a total time $O\left(2^{2k}\right)$. Then invoke ProcessBatchRecursive($\overrightarrow{t_{j_k+1}}$, deltas, dp_head) defined below.

// Input:
// $\overrightarrow{t_j}$: a ticket assignment
// deltas: an array of indices that generate $\overrightarrow{t_{j+1}}, \overrightarrow{t_{j+2}}, ..., \overrightarrow{t_{j+|\text{deltas}|}}$
// 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 $j \leq j' \leq j + |\text{deltas}|$,
// or $\perp$ if no such solution exists
**procedure** ProcessBatchRecursive($\overrightarrow{t_j}$, deltas, dp_head)
  **if** |deltas| $= 0$ **then**
    check dp_head.get() and **return** $\overrightarrow{t_j}$ if it is a valid solution or $\perp$ otherwise;
  left_branch_size $\leftarrow \left\lfloor \frac{|deltas|+1}{2} \right\rfloor$;
  deltas_left $\leftarrow$ the first (left_branch_size $- 1$) elements of deltas;

dp_left ← dp_head;
call dp_left.apply$\left(w_i, \left(\vec{t_j}\right)_i\right)$ for $i \in$ deltas $\setminus$ deltas_left;
call ProcessBatchRecursive($\vec{t_j}$, deltas_left, dp_left) and **return** if a solution is found;
deltas_right ← the last ($|$deltas$|$ − left_branch_size) elements of deltas;
dp_right ← dp_head;
call dp_right.apply$\left(w_i, \left(\overrightarrow{t_{j+\text{left_branch_size}}}\right)_i\right)$ for $i \in$ deltas $\setminus$ deltas_right;
**return** ProcessBatchRecursive($\overrightarrow{t_{j+\text{left_branch_size}}}$, deltas_right, dp_right);

It is easy to verify the correctness of the algorithm. We only need to show that the outermost invocation of ProcessBatchRecursive in $k$-th batch takes total time $O\left(k \cdot 2^{2k}\right)$. For simplicity, we will only count the number of calls to DP.apply(), each taking time $O(2^k)$, since they dominate the time complexity.

Define a recursive function $N$ as follows.

$$\begin{aligned} N(0) &= 0 \\ N(2t+1) &= 2t + 2 + 2T(t) \\ N(2t) &= 2t + 1 + N(t) + N(t-1) \end{aligned} \tag{4}$$

It can be seen that ProcessBatchRecursive($\vec{t_j}$, deltas, dp_head) does at most $N(|\text{deltas}|)$ applies, and thus, the outermost invocation does at most $N(j_{k+1} - j_k - 1) = N(2^k - 1)$ applies which is $k \cdot 2^k$ by the following lemma.

**Lemma 1.** *For all $d \in \mathbb{Z}_{\geq 0}$, $N(2^d - 1) = d \cdot 2^d$.*

*Proof.* The proof is by mathematic induction. If $d = 0$, then $N(2^d - 1) = N(0) = 0 = d \cdot 2^d$. Now assume $d \geq 0$ and $N(2^d - 1) = d \cdot 2^d$. By Lemma 2, $N(2^{d+1} - 1) = N(2^d - 1) + 2^d \cdot (d + 2) = d \cdot 2^d + 2^d \cdot (d + 2) = (2d + 2) \cdot 2^d = (d + 1) \cdot 2^{d+1}$. □

**Lemma 2.** *For all $d \in \mathbb{N}$, for all $m \in [2^{d-1}, 2^d - 1]$, $N(m) - N(m-1) = d + 1$.*

*Proof.* The proof is by mathematic induction on $d$. Since $N(1) - N(0) = 2$, the base case $d = 1$ is proven. Now assume $d \geq 1$ and that $\forall m \in [2^{d-1}, 2^d - 1]$, $N(m) - N(m-1) = d + 1$. Take any $m \in [2^d, 2^{d+1} - 1]$, we need to show that $N(m) - N(m-1) = d + 2$.
Case 1: $m$ is odd. Then $N(m) - N(m-1) = m + 1 + 2N\left(\frac{m-1}{2}\right) - m - N\left(\frac{m-1}{2}\right) - N\left(\frac{m-1}{2} - 1\right) = 1 + N\left(\frac{m-1}{2}\right) - N\left(\frac{m-1}{2} - 1\right) = 1 + (d + 1) = d + 2$.
Case 2: $m$ is even. Then $N(m) - N(m-1) = m + 1 + N\left(\frac{m}{2}\right) + N\left(\frac{m}{2} - 1\right) - m - 2N\left(\frac{m-2}{2}\right) = 1 + N\left(\frac{m}{2}\right) - N\left(\frac{m}{2} - 1\right) = 1 + (d + 1) = d + 2$.
The inductive step is proven. □

We note that super Swiper cannot be made faster in the general case by optimizing the search tree. See Appendix C.

## 5.3 Arbitrarily ordered weights

We previously assumed that the weights vector in the input is sorted. We will now explain how to remove this assumption for an additive cost of $O(n + R \log R)$.

We need a mutable data structure that we call sorted_weights that has a single method sorted_weights.next() returning the next largest weight from the input. We require that initializing sorted_weights takes time $O(n)$ and that the first $m$ calls to sorted_weights.next() take cumulative time $O(m \log m)$. This is sufficient since our algorithms would query sorted_weights.next() only $O(R)$ times.

Such a data structure can be constructed using a binary max-heap initialized in time $O(n)$ to contain all the weights and an auxiliary max-heap initially containing the reference to the main heap's root. sorted_weights.next() returns the root of the auxiliary heap and inserts in the auxiliary heap the references to the two children of the returned element in the main heap. [Fre93]

### 5.4 A note on reducing time complexity further

Swiper [TF23] explores using a quasilinear time approximate knapsack solver in lieu of the quadratic time exact algorithm. Numerical evaluations by the authors show that this reduces the quality of the output only by a little. While we do not explore this modification, it could be interesting to employ it to potentially reduce the running time of super Swiper to $O(n) + \tilde{O}(R)$.

## 6 Analysis of linear scaling based algorithms

We are interested in analyzing the approximation factor of known algorithms solving the $(\leq \alpha, < \beta)$-weight reduction problem, meaning how many times larger can the computed solution be compared to an optimal solution. We notice that all previously known algorithms use linear scaling with rounding to compute ticket assignments $(t_1, ..., t_n)$ and prove that all have worst case approximation factor $\Omega(n)$. We start with the following theorem that applies to a class of linear scaling based solutions.

**Theorem 1.** *Let $\alpha, \beta \in \mathbb{R}$ and $r, s \in \mathbb{N}$ such that $0 < \alpha < \beta < 1$ and $\frac{r}{s+1} < \beta \leq r/s \leq 1$. Also let $0 \leq c < 1$, $m \in \mathbb{Z}_{\geq 0}$, and suppose an algorithm solving the $(\leq \alpha, < \beta)$-weight reduction problem on input $(w_1, ..., w_n)$ always outputs $(t_1, ..., t_n)$ with the property that there exists $\delta > 0$ such that for all $i$,*

$$\lfloor \delta w_i + c \rfloor - m \leq t_i \leq \lfloor \delta w_i + c \rfloor.$$

*Then for infinitely many $n$ there exists a weight vector $\vec{w} = (w_1, ..., w_n)$ such that*

$$OPT_{\leq \alpha, < \beta}(\vec{w}) = s + 1$$

*but the algorithm outputs a solution with the number of tickets*

$$> (1 - c)\frac{\alpha n - r - 1}{r/s - \alpha} - ms.$$

*The worst case approximation factor of the algorithm is thus*

$$> \frac{(1 - c)\alpha}{(s + 1)(r/s - \alpha)} n - O(1).$$

*Proof.* Define $\vec{w} = (w_1, ..., w_n)$ where $w_1 = w_2 = ... = w_{n-s} = 1$ and $w_{n-s+1} = ... = w_n = w$ where

$$w = \left\lfloor \frac{\alpha n - \alpha s}{r - \alpha s} \right\rfloor = \frac{\alpha n - \alpha s}{r - \alpha s} - \varepsilon \tag{5}$$

for some $0 \leq \varepsilon < 1$. The total weight is $\sum_{i=1}^{n} w_i = n - s + sw$ and the adversary is allowed to corrupt the following weight.

$$W_{\mathsf{A}} = \lfloor \alpha W \rfloor = \lfloor \alpha n - \alpha s + \alpha s w \rfloor = \left\lfloor \alpha n - \alpha s + \frac{\alpha^2 sn - \alpha^2 s^2}{r - \alpha s} - \alpha s \varepsilon \right\rfloor =$$

$$\left\lfloor \frac{\alpha r n - \alpha r s}{r - \alpha s} - \alpha s \varepsilon \right\rfloor = \frac{\alpha r n - \alpha r s}{r - \alpha s} - \alpha s \varepsilon - \varepsilon'$$

for some $0 \leq \varepsilon' < 1$.

Then $W_{\mathsf{A}} - rw = (r - \alpha s)\varepsilon - \varepsilon'$. We would like to identify an infinite sequence of $n$ such that $W_{\mathsf{A}} = rw$. Since $r - \alpha s > 0$, we have $W_{\mathsf{A}} - rw > -1$. If $r - \alpha s \leq 1$, then also $W_{\mathsf{A}} - rw < 1$ and we are done. Otherwise, only consider $n = s + \lceil k \cdot \frac{r - \alpha s}{\alpha} \rceil$ for $k \in \mathbb{N}$. Then $n = s + k \cdot \frac{r - \alpha s}{\alpha} + \varepsilon''$ for some $0 \leq \varepsilon'' < 1$ and $\alpha(n - s) = k(r - \alpha s) + \alpha \varepsilon''$. From Equation 5,

$$\varepsilon = \frac{\alpha n - \alpha s}{r - \alpha s} - w = \frac{k(r - \alpha s) + \alpha \varepsilon''}{r - \alpha s} - w = k - w + \frac{\alpha \varepsilon''}{r - \alpha s}.$$

Since $k - w \in \mathbb{Z}$, $0 \leq \varepsilon < 1$ and $0 \leq \frac{\alpha \varepsilon''}{r - \alpha} < 1$, we have

$$\varepsilon = \frac{\alpha \varepsilon''}{r - \alpha s} < \frac{1}{r - \alpha s}.$$

Hence, $W_{\mathsf{A}} - rw < 1$ and $W_{\mathsf{A}} = rw$.

Consider a ticket assignment $\vec{t} = (t_1, ..., t_n)$ where $t_1 = ... = t_{n-s-1} = 0$ and $t_{n-s} = ... = t_n = 1$. Since the adversary can only corrupt $r$ parties that have a ticket, the total number of tickets is $s + 1$ and $r < \beta(s + 1)$ by an assumption in the theorem statement, this is a valid solution of cost $s + 1$. Clearly, no better solution is possible since the adversary can always corrupt $r$ tickets (assuming they exist) to get adversarial ratio $\geq r/s$. Hence, $\mathsf{OPT}_{\leq \alpha, < \beta}(\vec{w}) = s + 1$.

We will now prove a lower bound on the cost of algorithm's output. Take any valid ticket assignment $\vec{t} = (t_1, ..., t_n)$ and notice that at least one of $t_1$, ..., $t_{n-s}$ must be non-zero; otherwise the adversary can corrupt $r$ parties with the largest number of tickets to get adversarial ratio $\geq r/s$. Since the algorithm chooses $\delta > 0$ and for all $i$ outputs $t_i \leq \lfloor \delta w_i + c \rfloor$, we must have $\delta \geq 1 - c$. Hence, the algorithm outputs a ticket assignment of cost at least

$$s\big(\lfloor (1 - c)w + c \rfloor - m\big) > s\big((1 - c)w + c - 1 - m\big) >$$

$$s\left(\big(1 - c\big)\frac{\alpha n - \alpha s - 1}{r - \alpha s} - (1 - c) - m\right) = s\left(\big(1 - c\big)\frac{\alpha n - r - 1}{r - \alpha s} - m\right) =$$

$$\big(1 - c\big)\frac{\alpha n - r - 1}{r/s - \alpha} - ms.$$

□

When $\beta$ is rational, simply set $r$ and $s$ such that $\beta = r/s$ to apply the lower bound. Otherwise, we need to find $r, s$ such that $\frac{r}{s+1} < \beta \leq \frac{r}{s} \leq 1$ or, equivalently, $1 \leq \frac{s}{r} \leq \frac{1}{\beta} < \frac{s+1}{r}$. Setting $s = \lfloor 1/\beta \rfloor$ and $r = 1$ satisfies the inequality.

We now discuss particular weight reduction algorithms. By a simple modification of an argument in [TF23], we know that Swiper that solves the $(\leq \alpha, < \beta)$-weight reduction problem always outputs a solution with at most $\frac{\alpha(1-\alpha)}{\beta-\alpha}n+1$ tickets. Thus, this is also an upper bound on its approximation factor. Since Swiper outputs solutions $(t_1, ..., t_n)$ such that for each $i$, $t_i = \lfloor \delta w_i + \alpha \rfloor$ or $t_i = \lfloor \delta w_i + \alpha \rfloor - 1$, by Theorem 1, the worst case approximation factor of Swiper is $\frac{\alpha(1-\alpha)}{(s+1)(r/s-\alpha)}n - O(1)$ (for appropriate $r$ and $s$). Notice, that for some values of $\beta$, this is very close to the upper bound; for instance, when $\beta = 1/2$, it is a factor 3 away from the upper bound. The same result applies to faster Swiper and super Swiper.

We now discuss the gcdWR algorithm from [FMT24] (also see an overview in Appendix A). It is quite similar to Swiper in that it outputs a ticket assignment $(t_1, ..., t_n)$ of the form $t_i = \lfloor \delta w_i + c \rfloor$; here, $\delta = 1/\gcd$ and $c = 1/2$. Theorem 1 gives approximation factor lower bound of $\frac{\alpha}{2(s+1)(r/s-\alpha)}n - O(1)$. We conclude that the worst case approximation factor of gcdWR is $\Theta(n)$.

Finally, we discuss the algorithms in [DPTX24, v1] and [BHS23]. Even though they solve other variants of the weight reduction problem, we can still see what approximation factor they would give if solutions of the same form were to be used for the $(\leq \alpha, < \beta)$-weight reduction problem. [DPTX24, v1] outputs solutions of the form $t_i = \lfloor \delta w_i + 1/2 \rfloor$. By Theorem 1, its worst case approximation factor exceeds $\frac{\alpha}{2(s+1)(r/s-\alpha)}n - O(1)$ (for appropriate $r$ and $s$).

[BHS23] outputs solutions of the form $t_i = \lceil \delta w_i \rceil$. Since each $w_i$ is positive, each $t_i$ also is and thus the algorithm always outputs a solution of cost at least $n$. We are left to find an example where the optimal solution is small. If there exists a weight $w_i$ large enough that the adversary cannot corrupt it, then the optimal solution has one ticket in total. Hence, the worst case approximation factor is at least $n$.

# 7 Useful facts

In this section we present several facts that will be useful in the rest of the paper. We start with a (new) characterization of optimal solutions.

**Theorem 2.** *Assume $w_1 \leq w_2 \leq ... \leq w_n$. There exists an optimal solution $(t_1, ..., t_n)$ such that $t_1 \leq t_2 \leq ... \leq t_n$.*

*Proof.* Let $(t_1, ..., t_n)$ be an optimal solution such that $t_i > t_j$ for some $i < j$. We will show that $(t'_1, ..., t'_n) = (t_1, ..., t_{i-1}, t_j, t_{i+1}, ..., t_{j-1}, t_i, t_{j+1}, ...t_n)$ is also an optimal solution (i.e., $t_i$ and $t_j$ are swapped). This has the same cost, we just need to show that it is a valid ticket assignment.

Let $I \subseteq [n]$ such that $\sum_{k \in I} w_k \leq \alpha \sum_{k=1}^{n} w_k$. Consider the following cases.

- $i \notin I$, $j \notin I$: $\sum_{k \in I} t'_k = \sum_{k \in I} t_k < \beta \sum_{k=1}^{n} t_k = \beta \sum_{k=1}^{n} t'_k$;
- $i \in I$, $j \in I$: $\sum_{k \in I} t'_k = \sum_{k \in I} t_k < \beta \sum_{k=1}^{n} t_k = \beta \sum_{k=1}^{n} t'_k$;

- $i \in I$, $j \notin I$: $\sum_{k\in I} t'_k < \sum_{k\in I} t_k < \beta \sum_{k=1}^n t_k = \beta \sum_{k=1}^n t'_k$;
- $i \notin I$, $j \in I$: let $I' = I\backslash\{j\}\cup\{i\}$, $\sum_{k\in I} t'_k = \sum_{k\in I'} t_k < \beta \sum_{k=1}^n t_k = \beta \sum_{k=1}^n t'_k$, where the inequality holds since $\sum_{k\in I'} w_k < \sum_{k\in I} w_k \le \alpha \sum_{k=1}^n w_k$.

Hence, $(t'_1, ..., t'_n)$ is a valid assignment.

Therefore, we can sort all $t_i$ using, for example, insertion sort to get an optimal sorted assignment $(t_1, ..., t_n)$. □

We also prove a slightly stronger statement.

**Theorem 3.** *Assume $w_1 \le w_2 \le ... \le w_n$. There exists an optimal solution $(t_1, ..., t_n)$ such that $t_1 \le t_2 \le ... \le t_n$ and, moreover, for any $i, j$ with $w_i = w_j$, $|t_i - t_j| \le 1$.*

*Proof.* Let $(t_1, ..., t_n)$ be any solution and take any $i, j$ such that $w_i = w_j$ and $t_i + 2 \le t_j$. Define $(t'_1, ..., t'_n)$ such that $t'_i = t_i + 1$, $t'_j = t_j - 1$ and $t'_k = t_k$ for all other $k$. We will show that this is also a valid solution.

Let $I \subseteq [n]$ such that $\sum_{k\in I} w_k \le \alpha \sum_{k=1}^n w_k$ and consider the following cases.

- $i \notin I$, $j \notin I$: $\sum_{k\in I} t'_k = \sum_{k\in I} t_k < \beta \sum_{k=1}^n t_k = \beta \sum_{k=1}^n t'_k$;
- $i \in I$, $j \in I$: $\sum_{k\in I} t'_k = \sum_{k\in I} t_k < \beta \sum_{k=1}^n t_k = \beta \sum_{k=1}^n t'_k$;
- $i \notin I$, $j \in I$: $\sum_{k\in I} t'_k < \sum_{k\in I} t_k < \beta \sum_{k=1}^n t_k = \beta \sum_{k=1}^n t'_k$;
- $i \in I$, $j \notin I$: let $I' = I\backslash\{i\}\cup\{j\}$, $\sum_{k\in I} t'_k < \sum_{k\in I'} t_k < \beta \sum_{k=1}^n t_k = \beta \sum_{k=1}^n t'_k$, where the inequality holds since $\sum_{k\in I'} w_k = \sum_{k\in I} w_k \le \alpha \sum_{k=1}^n w_k$.

Hence, $(t'_1, ..., t'_n)$ is a valid assignment.

By Theorem 2, take an optimal solution $(t_1, ..., t_n)$ with $t_1 \le t_2 \le ... \le t_n$. One can see that we can iteratively achieve $|t_i - t_j| \le 1$ for any block of indices $\{l, l+1, ..., r\}$ with equal weight, while preserving the sortedness of the whole array. □

We will also find useful the following, first mentioned in [TF23, v2 appendix C]. Similarly to the definition of $\mathsf{OPT}_{\le\alpha,<\beta}(\vec{w})$ from section 3, define $\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<\beta}(\vec{w})$ to be the smallest maximum number of tickets that the adversary can obtain in a valid solution:

$$\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<\beta}(\vec{w}) = \min_{\substack{(t_1,...,t_n)\in\mathbb{Z}^n_{\ge 0}\\ \text{is valid}}} \max_{\substack{I\subseteq[n]\wedge\\ \sum_{i\in I} w_i \le \alpha \sum_{i=1}^n w_i}} \sum_{i\in I} t_i.$$

**Theorem 4.** *For all $0 < \alpha < \beta < 1$ and $\vec{w} \in \mathbb{N}^n$, we have*

$$\mathit{OPT}_{\le\alpha,<\beta}(\vec{w}) = \left\lfloor \frac{\mathit{OPT}^{T_\mathsf{A}}_{\le\alpha,<\beta}(\vec{w})}{\beta} \right\rfloor + 1; \quad \mathit{OPT}^{T_\mathsf{A}}_{\le\alpha,<\beta}(\vec{w}) = \left\lceil \beta \cdot \mathit{OPT}_{\le\alpha,<\beta}(\vec{w}) \right\rceil - 1.$$

*Proof.* Denote $T = \mathsf{OPT}_{\le\alpha,<\beta}(\vec{w})$ and $T_\mathsf{A} = \mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<\beta}(\vec{w})$.

We first prove that $T = \lfloor T_\mathsf{A}/\beta \rfloor + 1$. Let $(t_1, ..., t_n)$ be a valid ticket assignment where the maximum total number of tickets the adversary can obtain is $T_\mathsf{A}$. Decrease all $t_i$ as much as possible as long as $T_\mathsf{A} < \beta \sum_{i=1}^n t_i$; call the resulting ticket

assignment $(t'_1, ..., t'_n)$, which must be valid. Then $T \leq \sum_{i=1}^{n} t'_i = \lfloor T_{\mathsf{A}}/\beta \rfloor + 1$. Now take a valid ticket assignment $(t_1, ..., t_n)$ with $\sum_{i=1}^{n} t_i = T$. We have

$$T_{\mathsf{A}} \leq \max_{\substack{I \subseteq [n] \wedge \\ \sum_{i \in I} w_i \leq \alpha \sum_{i=1}^{n} w_i}} \sum_{i \in I} t_i < \beta T$$

and thus $T \geq \lfloor T_{\mathsf{A}}/\beta \rfloor + 1$. Hence, $T = \lfloor T_{\mathsf{A}}/\beta \rfloor + 1$.

We will now prove that $T_{\mathsf{A}} = \lceil \beta T \rceil - 1$. Since $T = \lfloor T_{\mathsf{A}}/\beta \rfloor + 1$, $T > T_{\mathsf{A}}/\beta$ which implies $T_{\mathsf{A}} < \beta T$ which implies $T_{\mathsf{A}} \leq \lceil \beta T \rceil - 1$. Also since $T = \lfloor T_{\mathsf{A}}/\beta \rfloor + 1$, $T - 1 \leq T_{\mathsf{A}}/\beta$ which implies $T_{\mathsf{A}} \geq \beta T - \beta \geq T_{\mathsf{A}} \geq \lceil \beta T \rceil - 1 - \beta$ which implies $T_{\mathsf{A}} \geq \lceil \beta T \rceil - 1$. □

Similarly, we have the following for the $(\leq \alpha, \leq \beta)$-weight reduction problem.

**Theorem 5.** *For all* $0 < \alpha < \beta < 1$ *and* $\vec{w} \in \mathbb{N}^n$*, we have*

$$\mathit{OPT}_{\leq\alpha,\leq\beta}(\vec{w}) = \max\left\{1, \left\lceil \frac{\mathit{OPT}^{T_A}_{\leq\alpha,\leq\beta}(\vec{w})}{\beta} \right\rceil\right\}; \quad \mathit{OPT}^{T_A}_{\leq\alpha,\leq\beta}(\vec{w}) = \left\lfloor \beta \cdot \mathit{OPT}_{\leq\alpha,\leq\beta}(\vec{w}) \right\rfloor.$$

# 8 Optimized bruteforce

Given that all previously known weight reduction algorithms have a worst case approximation factor $\Omega(n)$, we seek algorithms with improved approximation. In this section we present the first sub-exponential exact algorithm that is practical for some real-world weight distributions, as well as the first polynomial time algorithm with approximation factor at most $n/(c \log^2 n)$ for any $c > 0$.

## 8.1 Sub-exponential time exact solution

In this section we show an algorithm that solves the $(\leq \alpha, < \beta)$-weight reduction problem exactly (i.e., that finds an optimal solution) in time

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

where $C = \pi\sqrt{6}/3 \approx 2.57$ and $R \leq O(n)$ is the cost of the output (i.e., the optimal total number of tickets). Given that $R$ is often relatively small and definitely smaller than $n$, this algorithm is sometimes practical for real-world distributions, as the numerical evaluations in section 10 show. Thus, when a small solution is expected, it might make sense to run the exact algorithm for a number of steps and switch to a fallback scheme if necessary. The algorithm could also be very useful for research purposes, for example, to test some hypothesis about the problem.

Assume the weights in the input are sorted. Using the characterization of optimal solutions in Theorem 2, the algorithm for each $i = 1, 2, ...,$ enumerates all sorted ticket assignments with total number $i$ and for each, checks its validity using the knapsack algorithm. It outputs the first found valid ticket assignment.

The number of sorted ticket assignments with total number $i$ is bounded by the number of ways $p(i)$ to represent $i$ as a sum of positive integers, where $p$ is called partition function[1]. It is known that

$$p(k) \sim \frac{1}{4\sqrt{3}k} \exp\left(\frac{\pi\sqrt{6k}}{3}\right).$$

Also, the knapsack algorithm checks the validity of a ticket assignment in time $O(i^2)$ since the ticket assignment has at most $i$ non-zero entries. Hence, the running time of this algorithm is

$$O\left(\sum_{i=1}^{R} i^2 p(i)\right)$$

which is $O\left(R^{3/2}e^{C\sqrt{R}}\right)$ by the following lemma.

**Lemma 3.** *For any $c > 0$, $\sum_{i=1}^{k} ie^{c\sqrt{i}} = O\left(k^{3/2}e^{c\sqrt{k}}\right)$.*

*Proof.*

$$\sum_{i=1}^{k} ie^{c\sqrt{i}} \leq \int_{1}^{k+1} xe^{c\sqrt{x}}\, dx\ [=]$$

letting $y = c\sqrt{x}$,

$$[=] \int_{y=c}^{y=c\sqrt{k+1}} \left(\frac{y}{c}\right)^2 e^y\, d\left(\left(\frac{y}{c}\right)^2\right) = \int_{c}^{c\sqrt{k+1}} \left(\frac{y}{c}\right)^2 e^y \frac{2y}{c^2}\, dy =$$

$$\frac{2}{c^4}\int_{c}^{c\sqrt{k+1}} y^3 e^y\, dy = \frac{2}{c^4}\int_{y=c}^{y=c\sqrt{k+1}} y^3\, d(e^y)\ [=]$$

letting $z = e^y$, for sufficiently large $k$,

$$[=] \frac{2}{c^4}\int_{e^c}^{e^{c\sqrt{k+1}}} \ln^3 z\, dz = \frac{2}{c^4}\Big(z\big(\ln^3 z - 3\ln^2 z + 6\ln z - 6\big)\Big)\Big|_{e^c}^{e^{c\sqrt{k+1}}} \leq$$

$$\frac{2}{c^4} \cdot e^{c\sqrt{k+1}} \cdot \left(c\sqrt{k+1}\right)^3 = \frac{2}{c} \cdot (k+1)^{3/2} \cdot e^{c\sqrt{k+1}} = O\left(k^{3/2}e^{c\sqrt{k}}\right).$$

□

We note that it might be possible to reduce this time complexity by a factor of $R^{1/2}$ by reusing dynamic programming computations as in section 5. Lastly, see subsection 5.3 on how to handle unsorted inputs for an additive running time increase of $O(n + R\log R)$.

[1] https://en.wikipedia.org/wiki/Integer_partition

### 8.2 $O(n/\log^2 n)$-approximation in polynomial time

Using the idea in subsection 8.1 we can similarly construct an algorithm with approximation factor $\frac{n}{c\log^2 n}$ and polynomial running time

$$O\left(n^{\frac{\pi\sqrt{6}}{3}\sqrt{c}\log e\sqrt{\frac{\alpha(1-\alpha)}{\beta-\alpha}}}\log^3 n + n^2\right)$$

for any $c > 0$. The algorithm works by trying all sorted ticket assignments with total number at most $L$, to be defined below, and if no valid assignment was found, proceeding to using faster Swiper as fallback.

By [TF23], Swiper always finds a solution with cost at most

$$B = \frac{\alpha(1-\alpha)}{\beta-\alpha}n + 1.$$

By setting

$$L = \left\lceil \frac{Bc\log^2 n}{n} \right\rceil - 1,$$

the approximation factor of the algorithm is at most $\frac{B}{L+1} \le \frac{n}{c\log^2 n}$.

Similarly to subsection 8.1, the running time of the bruteforce is at most

$$O\left(n + L^{3/2}\exp\left(\frac{\pi\sqrt{6}}{3}\sqrt{L}\right)\right) = O\left(n + \log^3 n \cdot \exp\left(\frac{\pi\sqrt{6}}{3}\sqrt{c}\log n\sqrt{\frac{B}{n}}\right)\right) =$$

$$O\left(n + n^{\frac{\pi\sqrt{6}}{3}\sqrt{c}\log e\sqrt{\frac{B}{n}}} \cdot \log^3 n\right) = O\left(n + n^{\frac{\pi\sqrt{6}}{3}\sqrt{c}\log e\sqrt{\frac{\alpha(1-\alpha)}{\beta-\alpha}}}\log^3 n\right),$$

while the running time of faster Swiper is $O(n^2)$.

# 9 Linear programming based polynomial time approximation

In this section we show that for any constant $0 < \delta < 1 - \alpha/\beta$ there exists a polynomial time approximation algorithm that returns a solution with cost at most

$$\left\lfloor \frac{\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<(1-\delta)\beta}}{\beta} \right\rfloor + 1 = \left\lfloor (1-\delta)\cdot\frac{\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<(1-\delta)\beta}}{(1-\delta)\beta} \right\rfloor + 1 \le$$
$$\left\lfloor (1-\delta)\cdot\mathsf{OPT}_{\le\alpha,<(1-\delta)\beta} \right\rfloor + 1 \le \mathsf{OPT}_{\le\alpha,<(1-\delta)\beta}, \tag{6}$$

where the first inequality is due to Theorem 4.

The algorithm is based on the standard approach of representing an optimization problem as an integer linear program, relaxing it to allow fractional solutions, solving the relaxed LP in polynomial time and rounding the solution to an integer vector in a problem specific manner [WS11]. We will use the ellipsoid method to solve

an LP with exponentially many constraints. In this mode of operation, we are required to implement a "separation oracle" that the ellipsoid method will invoke. On input some vector $\vec{x} \in \mathbb{Q}^n$ in the multidimensional space, the separation oracle either reports that $\vec{x}$ satisfies all LP constraints or returns some violated constraint. Whenever the separation oracle is implemented by a polynomial time algorithm, the ellipsoid method also runs in polynomial time; in particular, its running time is independent of the number of LP constraints. See [WS11, section 4.3] for an overview of the ellipsoid method with a separation oracle.

For our problem, however, we face an issue that we cannot implement a polynomial time separation oracle that calculates the prescribed output exactly. The input will be a vector $\vec{t} \in \mathbb{Q}^n$ and we need to check if it satisfies the criteria of a valid ticket assignment. While we can test a ticket assignment $(t_1, ..., t_n) \in \mathbb{Z}^n_{\geq 0}$ for validity in time $O\big(n \cdot \sum_{i=1}^n t_i\big)$, we do not know how to efficiently do so for a fractional $\vec{t}$. We first present a sketch of our weight reduction algorithm that uses an inefficient separation oracle and in subsection 9.2 we show how to get around this issue.

## 9.1 Sketch

Assume $w_1 \leq w_2 \leq ... \leq w_n$ and define $k = \lceil 1/\delta \rceil$. We first efficiently bruteforce the set of indices $i$ with $t_i = g$ for each $0 \leq g \leq k-2$, having Theorem 2 in mind. Now consider the following satisfiability LP with some constant $T$, some $t_i$ fixed by the "guess" and the other $t_i$ constrained to be at least $k-1$.

$$
\begin{aligned}
\text{minimize} \quad & 0 \\
\text{subject to} \quad & \sum_{i=1}^{n} t_i = T \\
& \sum_{i \in I} t_i \leq \big\lfloor (1 - 1/k)\beta T \big\rfloor \quad \forall I \subseteq [n] \text{ s.t. } \sum_{i \in I} w_i \leq \alpha \sum_{i=1}^{n} w_i \\
& t_i = g_i \qquad\qquad \forall i : 1 \leq i < j \\
& t_i \geq k - 1 \qquad\qquad \forall i : j \leq i \leq n
\end{aligned}
\tag{7}
$$

For each $T \in \{1, 2, ..., B\}$, where $B = O(n)$ is a bound on $\mathsf{OPT}_{\leq \alpha, < (1-1/k)\beta}$ from [TF23], if there is a solution $(t'_1, ..., t'_n)$ to this LP, round it down to get $(t_1, ..., t_n)$, let $T_{\mathsf{A}} = \big\lfloor (1-1/k)\beta T \big\rfloor$, reduce all $t_i$ as much as possible as long as $T_{\mathsf{A}} < \beta \sum_{i=1}^n t_i$ to get $(t^+_1, ..., t^+_n)$ and record $(t^+_1, ..., t^+_n)$ as the best result for the current guess $(g_1, ..., g_{j-1})$. At the end of the algorithm, return the smallest result across all guesses.

We will now prove the bound in Equation 6 on the cost of the returned solution. By Theorem 2, fix an optimal solution $(t^*_1, ..., t^*_n)$ with $t^*_1 \leq t^*_2 \leq ... \leq t^*_n$ to the $(\leq \alpha, \leq (1 - 1/k)\beta)$-weight reduction problem. By definition, $\sum_{i=1}^n t^*_i = \mathsf{OPT}_{\leq \alpha, \leq (1-1/k)\beta} \leq \mathsf{OPT}_{\leq \alpha, < (1-1/k)\beta}$. Let $(g_1, ..., g_{j-1}) \in \{0, ..., k-2\}^{j-1}$ be the guess such that $(g_1, ..., g_{j-1}) = (t^*_1, ..., t^*_{j-1})$ and $t^*_i \geq k - 1$ for $i \geq j$.

For this guess, the algorithm will terminate the search on some $T$ at most $\mathsf{OPT}_{\leq \alpha, \leq (1-1/k)\beta}$. This is because if $T = \mathsf{OPT}_{\leq \alpha, \leq (1-1/k)\beta}$, then $(t^*_1, ..., t^*_n)$ is a valid

solution to the above LP. Therefore, the algorithm will find a solution $(t'_1, ..., t'_n)$ to the LP with $\sum_{i=1}^{n} t'_i = T \leq \mathsf{OPT}_{\leq\alpha,\leq(1-1/k)\beta}$.

For any $I \subseteq [n]$ that the adversary is allowed to corrupt,

$$\sum_{i\in I} t_i \leq \sum_{i\in I} t'_i \leq \left\lfloor (1-1/k)\beta T \right\rfloor = T_{\mathsf{A}} = \left\lfloor (1-1/k)\beta \sum_{i=1}^{n} t'_i \right\rfloor \leq (1-1/k)\beta \sum_{i=1}^{n} t'_i < \beta \sum_{i=1}^{n} t_i,$$

where the last inequality holds by the assumption that every $t'_i$ is either an integer or at least $k-1$, and that at least one $t'_i$ is non-zero. Therefore, $(t^+_1, ..., t^+_n)$ is a valid solution to the $(\leq \alpha, < \beta)$-weight reduction problem. Moreover, $T_{\mathsf{A}} \leq \left\lfloor (1-1/k)\beta \cdot \mathsf{OPT}_{\leq\alpha,\leq(1-1/k)\beta} \right\rfloor = \mathsf{OPT}^{T_{\mathsf{A}}}_{\leq\alpha,\leq(1-1/k)\beta} \leq \mathsf{OPT}^{T_{\mathsf{A}}}_{\leq\alpha,<(1-\delta)\beta}$, where the equality is due to Theorem 5. Hence,

$$\sum_{i=1}^{n} t^+_i = \left\lfloor \frac{T_{\mathsf{A}}}{\beta} \right\rfloor + 1 \leq \left\lfloor \frac{\mathsf{OPT}^{T_{\mathsf{A}}}_{\leq\alpha,<(1-\delta)\beta}}{\beta} \right\rfloor + 1.$$

## 9.2 Full algorithm

We now show how to circumvent the issue that we cannot construct a polynomial time separation oracle for the LP in Equation 7. We utilize the round-or-cut framework [CFLP00, AAKZ22] where at each iteration of the ellipsoid method, we either produce a separating hyperplane between the candidate feasible solution $\vec{t} \in \mathbb{Q}^n$ and the polytope or return another point $\vec{t'}$ that is slightly different from $\vec{t}$ but that will suffice for our weight reduction algorithm.

Configure the ellipsoid method with the LP in Equation 7 (see details in [GLS12]) and implement the separation oracle as follows. On input $(t_1, ..., t_n) \in \mathbb{Q}^n$, check the constraints in lines 1, 3 and 4 in Equation 7 as usual; instead of the constraints in line 2, check relaxed constraints

$$\sum_{i\in I} \lfloor t_i \rfloor \leq \left\lfloor (1-1/k)\beta T \right\rfloor \quad \forall I \subseteq [n] \text{ s.t. } \sum_{i\in I} w_i \leq \alpha \sum_{i=1}^{n} w_i.$$

using knapsack in polynomial time. If some check failed, we return one of constraints from the original LP in Equation 7 as a separating hyperplane; otherwise, we terminate the search and return $(t_1, ..., t_n)$ as the solution.

The result of this search will be either a point $(t_1, ..., t_n)$ that satisfies

$$\sum_{i=1}^{n} t_i = T$$

$$\sum_{i\in I} \lfloor t_i \rfloor \leq \left\lfloor (1-1/k)\beta T \right\rfloor \quad \forall I \subseteq [n] \text{ s.t. } \sum_{i\in I} w_i \leq \alpha \sum_{i=1}^{n} w_i$$

$$t_i = g_i \qquad \forall i : 1 \leq i < j$$

$$t_i \geq k-1 \qquad \forall i : j \leq i \leq n$$

or an error certifying that the LP in Equation 7 has no solution. The rest of the algorithm and the proof of correctness remains the same.

### 9.3 Time complexity

We briefly argue that the algorithm runs in polynomial time. Using Theorem 2, bruteforcing the set of indices $i$ with $t_i = j$ for each $0 \le j \le k-2$ takes time $O(n^{k-1})$. Bruteforcing $T$ adds another factor of $O(n)$ by construction. Finally, solving the LP takes polynomial time.

Despite this, the algorithm is completely impractical since for reasonable parameters, the running time would be $\Omega(n^{10})$.

### 9.4 Approximation factor discussion

We have already proved that the algorithm finds a solution with cost at most

$$\left\lfloor \frac{\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<(1-\delta)\beta}}{\beta} \right\rfloor + 1.$$

Using Theorem 4, the approximation factor for the total number of tickets is at most

$$\frac{\left\lfloor \frac{\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<(1-\delta)\beta}}{\beta} \right\rfloor + 1}{\left\lfloor \frac{\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<\beta}}{\beta} \right\rfloor + 1} < \frac{\frac{\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<(1-\delta)\beta}}{\beta} + 1}{\frac{\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<\beta}}{\beta}} = \frac{\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<(1-\delta)\beta}}{\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<\beta}} + \frac{\beta}{\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<\beta}}.$$

While $\beta/\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<\beta}$ can be made as small as $1/\log^2 n$ by subsection 8.2, we show in Appendix B that, unfortunately, $\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<(1-\delta)\beta}/\mathsf{OPT}^{T_\mathsf{A}}_{\le\alpha,<\beta}$ can be $\Omega(n)$ in the worst case.

## 10 Numerical evaluation

We implemented algorithms from this paper and from the literature in the Rust programming language and evaluated them based on the total number of tickets in the output and the running time. The stake distributions of Algorand, Aptos, Filecoin and Tezos blockchains were used as inputs, borrowed from the original Swiper implementation's git repository[2]; they were used for evaluations in the Swiper paper [TdS24, TF23].

The implemented algorithms from our work are faster Swiper (subsection 5.1), super Swiper (subsection 5.2) and the exact algorithm from subsection 8.1. The implemented algorithms from the literature are Swiper and gcdWR. Even though the original implementation of Swiper is publicly available, it is written in Python, a much slower programming language. To make the comparison fair, we reimplemented Swiper in Rust with the two implementations having identical output and almost identical logic; the only difference is that we did not implement the quasi-linear time approximate knapsack solver that Swiper runs before invoking the full solver, but [TF23] states that this optimization improves the running time only by a factor of 3 and only when the number of tickets in the output is large. We include

[2] https://github.com/DCL-TelecomParis/swiper

both implementations of Swiper in our evaluations. The original implementation in Python was slightly modified to solve the $(\leq \alpha, < \beta)$-weight reduction problem instead of the variant where the inequality involving $\alpha$ is strict. The LP based algorithm from section 9 was not implemented due to its impractically large time complexity and the unavailability of an ellipsoid method implementation with a separation oracle, and the general technical complexity of such an implementation.

All our implementations, as well as the original Swiper code in Python, are single-threaded. We assume sorted weights in the input, and thus our faster Swiper implementation has time complexity $O(R^2)$ as opposed to $O(n + R^2)$ in the full version of the algorithm, with a similar difference for super Swiper and the exact scheme. Even though, we didn't implement subsection 5.3 to handle unsorted inputs, we believe it does not significantly affect the measured running times since the stake distributions used all have less than 43000 entries. Our code is publicly available[3].

We ran the tests on AMD Threadripper PRO 7955WX. Tables 1, 2, 3 and 4 have two values for each evaluation: the total number of tickets and the running time. The exact algorithm was modified to search for a solution with the total number tickets up to 143. In case a solution wasn't found, the tables have a blank cell for the total number of tickets and the running time reported is the time duration before the search terminated.

## 10.1 Discussion

In Table 1 we indeed see that our faster Swiper algorithm is faster than the original Swiper. The speedup on Algorand's stake distribution ranges from 17x when the cost of the output is large to 13000x when the output cost is small. The huge efficiency gap in the small output case can be attributed to expensive rational number arithmetic when linearly scaling weights down in Swiper (time complexity $O(n \log n)$), which can dominate even the total cost of knapsack (time complexity $O(n^2 \log n)$). In contrast, our faster Swiper implementation does only $O(R \log R)$ rational number operations (inside gen.next() – see subsection 5.1). The cost of the output is comparable between Swiper and faster Swiper, but not exactly the same, as one or the other can get stuck in "local minimums" depending on luck.

Next, we observe that our super Swiper algorithm sometimes significantly improves the quality of the output over Swiper. For instance, for Aptos' stake distribution (Table 2), Swiper outputs 85 tickets for parameters $\alpha = 1/4, \beta = 1/3$ whereas super Swiper outputs 58; for $\alpha = 3/10, \beta = 1/3$, Swiper and faster Swiper output 346 tickets whereas super Swiper outputs 277. It is guaranteed that the output of super Swiper is never worse than that of Swiper and faster Swiper; moreover, in our measurements it is always better than that of gcdWR and is always within 6% of the optimal solution (based on the exact algorithm when it is feasible). At the same time, the running time of super Swiper is similar to that of faster Swiper and is often even smaller.

Finally, it is worth noting that while the exact algorithm has sub-exponential running time, it can nonetheless be practical when the optimal solution is small

[3] https://github.com/tolikzinovyev/weight-reduction

| | $\beta = 1/3$ | | | $\beta = 1/2$ | | | |
|---|---|---|---|---|---|---|---|
| | $\alpha = 0.2$ | $\alpha = 0.25$ | $\alpha = 0.3$ | $\alpha = 0.3$ | $\alpha = 0.35$ | $\alpha = 0.4$ | $\alpha = 0.45$ |
| Swiper (Python) | 235 | 745 | 22393 | 203 | 383 | 1203 | 17591 |
| | 4.75s | 5.14s | 6.25s | 4.32s | 5.15s | 4.74s | 5.70s |
| Swiper (Rust) | 235 | 745 | 22393 | 203 | 383 | 1203 | 17591 |
| | 1.10s | 1.22s | 1.78s | 1.03s | 1.14s | 1.19s | 1.65s |
| faster Swiper | 232 | 745 | 22660 | 203 | 387 | 1205 | 17949 |
| | 79.6µs | 542µs | 65.0ms | 111µs | 291µs | 2.01ms | 96.8ms |
| super Swiper | 232 | 745 | 22384 | 201 | 383 | 1171 | 17581 |
| | 75.7µs | 461µs | 115ms | 84.5µs | 202µs | 1.41ms | 118ms |
| gcdWR | 1180 | 6237 | 178906 | 223 | 407 | 1180 | 17648 |
| | 3.09ms | 2.83ms | 2.44ms | 3.29ms | 3.29ms | 3.11ms | 2.67ms |
| exact | | | | | | | |
| | 4.40h | 4.39h | 4.38h | 8.63h | 8.63h | 8.62h | 8.59h |

Table 1: Evaluation on the Algorand stake distribution

(e.g., in the case of Aptos). It could thus be used in conjunction with a fallback scheme when the optimal solution turns out larger than a threshold.

# 11 Conclusion

In this paper we made further progress in solving the problem of deterministic weight reduction. On the practical side, faster Swiper is a very efficient algorithm with time complexity $O(n + R^2)$ that outputs solutions comparable to the optimal on real-world inputs while super Swiper outputs solutions extremely close to optimal in a similar time $O(n + R^2 \log R)$. The latter would be a great candidate for deployment in, e.g., proof-of-stake blockchains.

On the theoretical side, we proved that all known algorithms based on linear scaling suffer from $\Omega(n)$ approximation ratio in the worst case. We presented the first polynomial time algorithm with approximation factor $n/(c \log^2 n)$, for any $c > 0$, and the first exact algorithm with sub-exponential running time $O\left(n+R^{3/2}e^{C\sqrt{R}}\right)$. Given that $R$ is often much smaller than $n$, our exact algorithm can also be practical for production or research purposes. Finally, our linear programming based algorithm could be useful in the context of future research.

It could be interesting to try integrating a quasilinear time approximate knapsack solver, as done in the quasilinear mode Swiper [TF23], to decrease the exponent of $R$ by one in the time complexity of faster Swiper, super Swiper and the exact scheme. Additionally, we would like to know if there exists an algorithm with approximation factor $o(n/\log^2 n)$. The answer is likely yes, but more exploration is required.

Finally, it would be good to investigate the related variants of committee selection

| | $\beta = 1/3$ | | | $\beta = 1/2$ | | | |
|---|---|---|---|---|---|---|---|
| | $\alpha = 0.2$ | $\alpha = 0.25$ | $\alpha = 0.3$ | $\alpha = 0.3$ | $\alpha = 0.35$ | $\alpha = 0.4$ | $\alpha = 0.45$ |
| Swiper (Python) | 22 | 85 | 346 | 23 | 29 | 95 | 279 |
| | 5.37ms | 7.32ms | 6.94ms | 4.87ms | 6.19ms | 6.76ms | 7.06ms |
| Swiper (Rust) | 22 | 85 | 346 | 23 | 29 | 95 | 279 |
| | 1.61ms | 2.07ms | 1.96ms | 1.40ms | 1.88ms | 1.80ms | 1.96ms |
| faster Swiper | 22 | 58 | 346 | 23 | 31 | 95 | 279 |
| | 8.12µs | 13.7µs | 120µs | 7.86µs | 8.47µs | 32.6µs | 159µs |
| super Swiper | 22 | 58 | 277 | 23 | 29 | 95 | 241 |
| | 6.55µs | 13.0µs | 116µs | 6.43µs | 7.13µs | 31.3µs | 88.4µs |
| gcdWR | 138 | 246 | 579 | 34 | 34 | 138 | 321 |
| | 8.02µs | 7.85µs | 7.87µs | 8.46µs | 8.77µs | 8.02µs | 8.02µs |
| exact | 22 | 55 | | 23 | 29 | 91 | |
| | 133µs | 127ms | 4.43h | 236µs | 1.08ms | 45.9s | 8.65h |

Table 2: Evaluation on the Aptos stake distribution

| | $\beta = 1/3$ | | | $\beta = 1/2$ | | | |
|---|---|---|---|---|---|---|---|
| | $\alpha = 0.2$ | $\alpha = 0.25$ | $\alpha = 0.3$ | $\alpha = 0.3$ | $\alpha = 0.35$ | $\alpha = 0.4$ | $\alpha = 0.45$ |
| Swiper (Python) | 1282 | 3091 | 10288 | 1149 | 1835 | 3533 | 8133 |
| | 281ms | 324ms | 388ms | 287ms | 301ms | 322ms | 430ms |
| Swiper (Rust) | 1282 | 3091 | 10288 | 1149 | 1835 | 3533 | 8133 |
| | 29.1ms | 39.6ms | 97.2ms | 31.7ms | 35.8ms | 49.6ms | 139ms |
| faster Swiper | 1282 | 3088 | 10288 | 1149 | 1837 | 3533 | 8139 |
| | 2.67ms | 8.21ms | 65.2ms | 3.69ms | 4.18ms | 11.9ms | 37.8ms |
| super Swiper | 1279 | 3088 | 10282 | 1147 | 1835 | 3527 | 8133 |
| | 1.14ms | 5.62ms | 45.2ms | 1.40ms | 3.13ms | 8.57ms | 42.2ms |
| gcdWR | 3700 | 6516 | 17425 | 1289 | 1882 | 3700 | 8190 |
| | 166µs | 162µs | 158µs | 169µs | 169µs | 165µs | 162µs |
| exact | | | | | | | |
| | 4.39h | 4.45h | 4.40h | 8.43h | 8.44h | 8.42h | 8.43h |

Table 3: Evaluation on the Filecoin stake distribution

| | $\beta = 1/3$ | | | $\beta = 1/2$ | | | |
|---|---|---|---|---|---|---|---|
| | $\alpha = 0.2$ | $\alpha = 0.25$ | $\alpha = 0.3$ | $\alpha = 0.3$ | $\alpha = 0.35$ | $\alpha = 0.4$ | $\alpha = 0.45$ |
| Swiper (Python) | 49 | 133 | 589 | 37 | 75 | 143 | 455 |
| | 28.5ms | 28.3ms | 31.3ms | 20.1ms | 29.2ms | 25.8ms | 28.9ms |
| Swiper (Rust) | 49 | 133 | 589 | 37 | 75 | 143 | 455 |
| | 3.33ms | 3.79ms | 4.59ms | 2.71ms | 3.97ms | 3.43ms | 4.27ms |
| faster Swiper | 49 | 133 | 589 | 41 | 77 | 145 | 455 |
| | 11.8µs | 52.2µs | 396µs | 12.8µs | 29.6µs | 67.4µs | 188µs |
| super Swiper | 37 | 127 | 523 | 35 | 75 | 143 | 427 |
| | 10.3µs | 25.9µs | 322µs | 10.6µs | 24.6µs | 50.9µs | 197µs |
| gcdWR | 148 | 333 | 1331 | 46 | 87 | 148 | 470 |
| | 13.8µs | 12.8µs | 11.3µs | 14.7µs | 13.7µs | 13.8µs | 12.3µs |
| exact | 37 | 127 | | 35 | 75 | 139 | |
| | 6.19ms | 1.04h | 4.97h | 5.15ms | 6.46s | 6.40h | 8.97h |

Table 4: Evaluation on the Tezos stake distribution

where we require the weaker ("randomized") security from Equation 1 or where the goal is to minimize the number of distinct committee members (see section 2). [GKR23] makes progress in the "randomized" setting presenting schemes performing very well in practice, but, unfortunately, nothing is said about their approximation factor. It would be interesting to prove that a fully deterministic solution can be optimal when the weaker security is required.

## 12 Acknowledgements

The author thanks his advisor Leo Reyzin for help with preparing this paper and contribution to Appendix C, as well as Nathan Klein, Rico Zenklusen for helpful discussions.

This material is based upon work supported in part by a gift from DARPA under Agreements No. HR00112020021, HR00112020023 and HR001120C0085. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

## References

[AAKZ22] Georg Anegg, Haris Angelidakis, Adam Kurpisz, and Rico Zenklusen. A technique for obtaining true approximations for k-center with covering constraints. *Math. Program.*, 192(1–2):3–27, March 2022. doi:10.1007/s10107-021-01645-y.

[ACD+25] Ittai Abraham, Eli Chouatt, Ivan Damgård, Yossi Gilad, Gilad Stern, and Sophia Yakoubov. Asynchronous algorand: Reaching agreement with near linear communication and constant expected time. Cryptology ePrint Archive, Report 2025/303, 2025. URL: https://eprint.iacr.org/2025/303.

[BCDH23] Elette Boyle, Ran Cohen, Deepesh Data, and Pavel Hubácek. Must the communication graph of MPC protocols be an expander? *Journal of Cryptology*, 36(3):20, July 2023. doi:10.1007/s00145-023-09460-8.

[BCG24] Elette Boyle, Ran Cohen, and Aarushi Goel. Breaking the $O(\sqrt{n})$-bit barrier: Byzantine agreement with polylog bits per party. *Journal of Cryptology*, 37(1):2, January 2024. doi:10.1007/s00145-023-09484-0.

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 742–762. Springer, Berlin, Heidelberg, August 2015. doi:10.1007/978-3-662-48000-7_36.

[BGT13] Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. Communication locality in secure multi-party computation - how to run sublinear algorithms in a distributed setting. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 356–376. Springer, Berlin, Heidelberg, March 2013. doi:10.1007/978-3-642-36594-2_21.

[BHS23] 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.

[Bra85] Gabriel Bracha. An $O(\lg n)$ expected rounds randomized byzantine generals protocol. In *17th ACM STOC*, pages 316–326. ACM Press, May 1985. doi:10.1145/22145.22180.

[BSGH13] Nicolas Braud-Santoni, Rachid Guerraoui, and Florian Huc. Fast byzantine agreement. In Panagiota Fatourou and Gadi Taubenfeld, editors, *32nd ACM PODC*, pages 57–64. ACM, July 2013. doi:10.1145/2484239.2484243.

[BW06] Amos Beimel and Enav Weinreb. Monotone circuits for monotone weighted threshold functions. *Information Processing Letters*, 97(1):12–18, 2006. URL: https://www.sciencedirect.com/science/article/pii/S0020019005002620, doi:10.1016/j.ipl.2005.09.008.

[CDGK22] Ignacio Cascudo, Bernardo David, Lydia Garms, and Anders Konring. YOLO YOSO: Fast and simple encryption and secret sharing

in the YOSO model. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part I*, volume 13791 of *LNCS*, pages 651–680. Springer, Cham, December 2022. doi:10.1007/978-3-031-22963-3_22.

[CFLP00] Robert D. Carr, Lisa Fleischer, Vitus J. Leung, and Cynthia A. Phillips. Strengthening integrality gaps for capacitated network design and covering problems. In David B. Shmoys, editor, *11th SODA*, pages 106–115. ACM-SIAM, January 2000.

[CHOR22] Ran Cohen, Iftach Haitner, Eran Omri, and Lior Rotem. From fairness to full security in multiparty computation. *Journal of Cryptology*, 35(1):4, January 2022. doi:10.1007/s00145-021-09415-x.

[CK24] Pyrros Chaidos and Aggelos Kiayias. Mithril: Stake-based threshold multisignatures. In Markulf Kohlweiss, Roberto Di Pietro, and Alastair R. Beresford, editors, *CANS 2024, Part I*, volume 14905 of *LNCS*, pages 239–263. Springer, Singapore, September 2024. doi:10.1007/978-981-97-8013-6_11.

[CKR+23] Ran Canetti, Sebastian Kolby, Divya Ravi, Eduardo Soria-Vazquez, and Sophia Yakoubov. Taming adaptivity in YOSO protocols: The modular way. In Guy N. Rothblum and Hoeteck Wee, editors, *TCC 2023, Part II*, volume 14370 of *LNCS*, pages 33–62. Springer, Cham, November / December 2023. doi:10.1007/978-3-031-48618-0_2.

[CKRZ24] 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.

[CM19] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777:155–183, 2019. In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I. URL: https://www.sciencedirect.com/science/article/pii/S030439751930091X, doi:10.1016/j.tcs.2019.02.001.

[CS04] John F. Canny and Stephen Sorkin. Practical large-scale distributed key generation. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 138–152. Springer, Berlin, Heidelberg, May 2004. doi:10.1007/978-3-540-24676-3_9.

[DCX+23] Sourav Das, Philippe Camacho, Zhuolun Xiang, Javier Nieto, Benedikt Bünz, and Ling Ren. Threshold signatures from inner product argument: Succinct, weighted, and multi-threshold. In Weizhi Meng,

Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 356–370. ACM Press, November 2023. doi:10.1145/3576915.3623096.

[DKM+17] Varsha Dani, Valerie King, Mahnush Movahedi, Jared Saia, and Mahdi Zamani. Secure multi-party computation in large networks. *Distrib. Comput.*, 30(3):193–229, June 2017. doi:10.1007/s00446-016-0284-9.

[DPTX24] Sourav Das, Benny Pinkas, Alin Tomescu, and Zhuolun Xiang. Distributed randomness using weighted VRFs. Cryptology ePrint Archive, Report 2024/198, 2024. URL: https://eprint.iacr.org/2024/198.

[DPTX25] 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.

[FHASW24] Nils Fleischhacker, Mathias Hall-Andersen, Mark Simkin, and Benedikt Wagner. Jackpot: Non-interactive aggregatable lotteries. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part VI*, volume 15489 of *LNCS*, pages 365–397. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0938-3_12.

[FMT24] 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.

[Fre93] G.N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104(2):197–214, 1993. URL: https://www.sciencedirect.com/science/article/pii/S0890540183710308, doi:10.1006/inco.1993.1030.

[GHK+21] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. YOSO: You only speak once - secure MPC with stateless ephemeral roles. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 64–93, Virtual Event, August 2021. Springer, Cham. doi:10.1007/978-3-030-84245-1_3.

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132757.

[GJM+23] Sanjam Garg, Abhishek Jain, Pratyay Mukherjee, Rohit Sinha, Mingyuan Wang, and Yinuo Zhang. Cryptography with weights: MPC, encryption and signatures. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part I*, volume 14081 of *LNCS*, pages 295–327. Springer, Cham, August 2023. doi:10.1007/978-3-031-38557-5_10.

[GJM+24] Sanjam Garg, Abhishek Jain, Pratyay Mukherjee, Rohit Sinha, Mingyuan Wang, and Yinuo Zhang. hinTS: Threshold signatures with silent setup. In *2024 IEEE Symposium on Security and Privacy*, pages 3034–3052. IEEE Computer Society Press, May 2024. doi:10.1109/SP54263.2024.00057.

[GKR23] 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.

[GLS12] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric algorithms and combinatorial optimization*, volume 2. Springer Science & Business Media, 2012. doi:10.1007/978-3-642-78240-4.

[HKZ24] D. Ellis Hershkowitz, Nathan Klein, and Rico Zenklusen. Ghost value augmentation for k-edge-connectivity. In Bojan Mohar, Igor Shinkar, and Ryan O'Donnell, editors, *56th ACM STOC*, pages 1853–1864. ACM Press, June 2024. doi:10.1145/3618260.3649715.

[KKK+10] Bruce M. Kapron, David Kempe, Valerie King, Jared Saia, and Vishal Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. *ACM Trans. Algorithms*, 6(4), September 2010. doi:10.1145/1824777.1824788.

[KLST11] Valerie King, Steven Lonargan, Jared Saia, and Amitabh Trehan. Load balanced scalable byzantine agreement through quorum building, with full information. In Marcos K. Aguilera, Haifeng Yu, Nitin H. Vaidya, Vikram Srinivasan, and Romit Roy Choudhury, editors, *Distributed Computing and Networking*, pages 203–214, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-17679-1_18.

[KS09] Valerie King and Jared Saia. From almost everywhere to everywhere: Byzantine agreement with $\tilde{O}(n^{3/2})$ bits. In Idit Keidar, editor, *Distributed Computing*, pages 464–478, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-04355-0_47.

[KS11] Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: Scalable byzantine agreement with an adaptive adversary. *J. ACM*, 58(4), July 2011. doi:10.1145/1989727.1989732.

[KSSV06] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *17th SODA*, pages 990–999. ACM-SIAM, January 2006.

[TdS24] Andrei Tonkikh and Luciano Freitas de Souza. Swiper: a new paradigm for efficient weighted distributed protocols. In Ran Gelles, Dennis Olivetti, and Petr Kuznetsov, editors, *43rd ACM PODC*, pages 283–294. ACM, June 2024. doi:10.1145/3662158.3662799.

[TF23] 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.

[WS11] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, USA, 1st edition, 2011. doi:10.1017/CBO9780511921735.

# A More detailed overview of prior work

Swiper [TdS24] defines three variants of the problem.

- The $(\alpha_w, \alpha_n)$-weight restriction problem with $\alpha_w < \alpha_n$ is nearly identical to the problem we consider: given $w_1, ..., w_n$ as input, output $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that for all $A \subseteq [n]$ with $\sum_{i \in A} w_i < \alpha_w \sum_{i=1}^{n} w_i$, we have $\sum_{i \in A} t_i < \alpha_n \sum_{i=1}^{n} t_i$.
- The $(\beta_w, \beta_n)$-weight qualification problem with $\beta_w > \beta_n$ requires that given $w_1, ..., w_n$ as input the algorithm outputs $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that for all $H \subseteq [n]$ with $\sum_{i \in H} w_i > \beta_w \sum_{i=1}^{n} w_i$, we have $\sum_{i \in H} t_i > \alpha_n \sum_{i=1}^{n} t_i$. This is equivalent to the $(1 - \beta_w, 1 - \beta_n)$-weight restriction problem.
- Lastly, the $(\alpha, \beta)$-weight separation problem with $\alpha < \beta$ requires that given $w_1, ..., w_n$ as input the algorithm outputs $t_1, ..., t_n \in \mathbb{Z}_{\geq 0}$ such that for all $A \subseteq [n]$ with $\sum_{i \in A} w_i < \alpha \sum_{i=1}^{n} w_i$ and all $H \subseteq [n]$ with $\sum_{i \in H} w_i > \beta \sum_{i=1}^{n} w_i$, we have $\sum_{i \in A} t_i < \sum_{i \in H} t_i$.

The authors of Swiper prove that all three variants admit an efficiently computable solution of cost $\sum_{i=1}^{n} t_i \leq O(n)$ (assuming $\alpha$ and $\beta$ parameters are constant). Such a solution can be found by linearly scaling down $(w_1, ..., w_n)$ and rounding each $w_i$ to an integer according to a simple rule. Swiper goes a step further and tries to compute a solution as small as possible by running a binary search on the scaling factor and testing the correctness of a vector $(t_1, ..., t_n)$ at each iteration. The time complexity of Swiper is $O(n^2 \log n)$.

[BHS23] considers the problem of weight reduction in the context of weighted ramp secret sharing. They present a scheme parameterized by $(\alpha, \beta)$ with $\alpha < \beta$ such that for all $A \subseteq [n]$ with $\sum_{i \in A} w_i \leq \alpha \sum_{i=1}^{n} w_i$ we have $\sum_{i \in A} t_i \leq (\alpha + \frac{\beta - \alpha}{5}) \sum_{i=1}^{n} t_i$ and for all $H \subseteq [n]$ with $\sum_{i \in H} w_i \geq \beta \sum_{i=1}^{n} w_i$ we have $\sum_{i \in H} t_i \geq (\beta - \frac{\beta - \alpha}{4}) \sum_{i=1}^{n} t_i$. Their algorithm finds a solution with cost $\sum_{i=1}^{n} \leq O(n)$ by simply linearly scaling all

weights down and rounding up to an integer, all in time $O(n)$. It doesn't attempts to find solutions better than $O(n)$.

[FMT24] presents an algorithm for weight reduction in the context of distributed key generation. Their algorithm outputs a solution such that for all $A \subseteq [n]$ with $\sum_{i\in A} w_i < \alpha \sum_{i=1}^{n} w_i$, we have $\sum_{i\in A} t_i < \beta \sum_{i=1}^{n} t_i$ where $\alpha = 1/3, \beta = 1/2$, and like the above approaches, it also always outputs a solution of cost at most $O(n)$. It works by adjusting each $w_i$ such that all $w_i$ become a multiple of some, hopefully large, integer gcd, and then divides the result by gcd. We call this algorithm gcdWR. In subsection A.1 we show that this is in essence linear scaling with rounding as in Swiper, but unlike Swiper, each iteration of the binary search does not test a vector $(t_1, ..., t_n)$ for correctness precisely, but uses a heuristic with false negatives. The time complexity is $O(n \log n)$. In section 10 it can be seen that gcdWR outputs comparable solutions to Swiper when $\beta = 1/2$, but doesn't generalize well for other values of $\beta$. Since we implement and test gcdWR, we give a more detailed overview and explain how we generalize it below in subsection A.1.

Finally, [DPTX25] employs deterministic weight reduction in the context of threshold weighted verifiable unpredictable functions, weighted publicly verifiable secret sharing and weighted distributed key generation. The algorithm, described in the first version of their ePrint manuscript [DPTX24, v1], also performs linear scaling with rounding, though it solves an inverse problem: minimizing the gap between $\alpha$ and $\beta$ when given a bound on the total new weight $\sum_{i=1}^{n} t_i$.

## A.1 gcdWR

As defined in [FMT24], gcdWR solves the $(< 1/3, < 1/2)$-weight reduction problem. We instead provide a modification-generalization of the algorithm that solves $(\leq \alpha, < \beta)$-weight reduction for $0 < \alpha < \beta \leq 1/2$.

Assume the weights in the input are $w_1, w_2, ..., w_n \in \mathbb{N}$. Let $\text{gcd} \in \mathbb{N}$ and define

$$w'_i = \begin{cases} w_i = (w_i \bmod \text{gcd}), & \text{if } w_i \bmod \text{gcd} < \text{gcd}/2, \\ w_i + \text{gcd} - (w_i \bmod \text{gcd}), & \text{otherwise;} \end{cases}$$

$$\Delta_i = |w_i - w'_i|;$$

$$\Delta = \sum_{i=1}^{n} \Delta_i.$$

The algorithm runs a binary search to find a locally maximal gcd for which $\Delta$ is below some threshold value, which will guarantee that $(w'_1, ..., w'_n)$ is a valid ticket assignment. The algorithm then outputs $\left(\frac{w'_1}{\text{gcd}}, ..., \frac{w'_n}{\text{gcd}}\right)$.

**Theorem 6.** *Let $W = \sum_{i=1}^{n} w_i$. If $\Delta < \frac{\beta-\alpha}{1-\beta} W$, then for any adversarial set $A \subseteq [n]$ with $\sum_{i\in A} w_i \leq \alpha W$, $\sum_{i\in A} w'_i < \beta \sum_{i=1}^{n} w'_i$.*

*Proof.* Take any $A \subseteq [n]$ with $\sum_{i\in A} w_i \leq \alpha W$ and define $H = [n] \setminus A$. Also define $W_S = \sum_{i\in S} w_i$, $W'_S = \sum_{i\in S} w'_i$, $W' = \sum_{i=1}^{n} w'_i$, $\Delta_S = \sum_{i\in S} \Delta_i$.

We know that $W_A \leq \alpha W = \alpha(W_A + W_H)$ which is equivalent to $(1-\alpha)W_A \leq \alpha W_H$, and we need to prove that $W'_A < \beta W' = \beta(W'_A + W'_H)$ which is equivalent to

$(1-\beta)W'_A < \beta W'_H$. Since $W'_A \le W_A + \Delta_A$ and $W'_H \ge W_H - \Delta_H$, it is sufficient to prove that $(1-\beta)(W_A+\Delta_A) < \beta(W_H - \Delta_H)$, which is equivalent to

$$(1-\beta)\Delta_A + \beta\Delta_H < \beta W_H - (1-\beta)W_A. \tag{8}$$

Since $W_A \le \alpha W$, we have $W_H = W - W_A \ge (1-\alpha)W$ and

$$\beta W_H - (1-\beta)W_A \ge \beta(1-\alpha)W - (1-\beta)\alpha W = (\beta-\alpha)W. \tag{9}$$

Also,

$$\begin{aligned}(1-\beta)\Delta_A + \beta\Delta_H = (1-\beta)\Delta_A + \beta(\Delta - \Delta_A) = \\ \beta\Delta + (1-2\beta)\Delta_A \le \beta\Delta + (1-2\beta)\Delta = (1-\beta)\Delta.\end{aligned} \tag{10}$$

Combining Equation 8, Equation 9 and Equation 10, it is sufficient to show that $(1-\beta)\Delta < (\beta-\alpha)W$. It follows by an assumption on $\Delta$. □

By setting $\Delta_{\max} := \left\lceil \frac{\beta-\alpha}{1-\beta} W \right\rceil$ and the left threshold of the binary search $\mathrm{gcd} := \max\left\{2\Delta_{\max}/n, 1\right\}$, gcdWR guarantees to find a solution with the total number of tickets $O(n)$ in time $O(n \log n)$.

We notice that this algorithm is quite similar to Swiper: it outputs $t_i = \lfloor s w_i + c \rfloor$ where in the case of gcdWR, $c := 1/2$ and $s := 1/\mathrm{gcd}$. However, unlike Swiper, it doesn't achieve the worst case number of tickets from Equation 3 due to a suboptimal choice of $c$. It also does not precisely determine the validity of ticket assignments during binary search through knapsack, but instead opts for a heuristical test that may yield false negatives; in fact, numerical evaluations in section 10 show that the performance of gcdWR suffers when $\beta < 1/2$, likely due to the inequality in Equation 10 becoming loose. On the bright side, gcdWR has running time $O(n \log n)$ as opposed to $O(n^2 \log n)$ of Swiper (we shall note that Swiper has a quasilinear mode with time complexity $\tilde{O}(n)$, but we do not touch upon it in this paper).

# B Smoothness analysis

**Theorem 7.** *Let $\alpha \in \mathbb{R}$, $r, s \in \mathbb{N}$ such that $0 < \alpha < r/s < 1/2$. For all sufficiently large $n$ there exists a weight vector $\vec{w} = (w_1, ..., w_n)$ such that the following hold.*

$$OPT^{T_A}_{\le\alpha, \le r/s}(\vec{w}) = r;$$

$$OPT^{T_A}_{\le\alpha, <r/s}(\vec{w}) > \frac{\left((1-\frac{r}{s})r+1\right)\alpha}{r+1-s\alpha} n - O(1) > \frac{\alpha n}{2}.$$

*Proof.* Take any sufficiently large $n$ and let $w_1 = w_2 = ... = w_{n-s} = 1$ and $w_{n-s+1} = ... = w_n = w = \left\lceil \frac{\alpha}{r+1-s\alpha} n \right\rceil = \frac{\alpha}{r+1-s\alpha} n + \varepsilon$ where $0 \le \varepsilon < 1$. The total weight is $W = \sum_{i=1}^{n} w_i = n - s + sw$ and the adversary is permitted to have weight

$$\begin{aligned}W_{\mathsf{A}} = \lfloor \alpha W \rfloor = \lfloor \alpha n - s\alpha + s\alpha w \rfloor = \\ \left\lfloor \alpha n - s\alpha + \frac{s\alpha^2}{r+1-s\alpha} n + s\alpha\varepsilon \right\rfloor = \left\lfloor \frac{(r+1)\alpha}{r+1-s\alpha} n + s\alpha(\varepsilon - 1) \right\rfloor.\end{aligned}$$

We first prove $\mathsf{OPT}^{T_\mathsf{A}}_{\leq\alpha,\leq r/s}(\vec{w}) = r$. $\vec{t} = (t_1, t_2, ..., t_n)$ where $t_1 = ... = t_{n-s} = 0$ and $t_{n-s+1} = ... = t_n = 1$ is a valid solution since $W_\mathsf{A} < (r+1)w$ and so the adversary can get at most $r$ out of $s$ tickets. Additionally, for sufficiently large $n$, $W_\mathsf{A} \geq rw$, and so the adversary can get $r$ tickets. Therefore, $\mathsf{OPT}^{T_\mathsf{A}}_{\leq\alpha,\leq r/s}(\vec{w}) = r$.

We will now show that

$$\mathsf{OPT}^{T_\mathsf{A}}_{\leq\alpha,<r/s}(\vec{w}) > \frac{((1-\frac{r}{s})r+1)\alpha}{r+1-s\alpha}n - O(1). \tag{11}$$

Take an optimal ticket vector $\vec{t} = (t_1, ..., t_n)$ satisfying the properties in Theorem 3. In particular, $t_1 \leq t_2 \leq ... \leq t_n$ and $t_n \leq t_{n-s+1} + 1$. Let $k$ be the number of non-zero numbers in $\{t_1, ..., t_{n-s}\}$. If $k \geq W_\mathsf{A}$, then the adversary can get $W_\mathsf{A}$ tickets and Equation 11 is proven.

Now assume $k < W_\mathsf{A}$. $\frac{r}{s}k > W_\mathsf{A} - rw$ since otherwise the adversary can corrupt the top $(W_\mathsf{A} - rw)$ small and the top $r$ large parties to get $r/s$ fraction of the tickets. Then $k > \frac{s}{r}(W_\mathsf{A} - rw) = \frac{s}{r}\big((r+1)w - O(1) - rw\big) = \frac{s}{r}w - O(1)$. Define

$$l = \left\lceil \frac{W_\mathsf{A} - k}{w} \right\rceil.$$

Since $\frac{s}{r} > 2$ and $w = \Omega(n)$, $\frac{W_\mathsf{A}-k}{w} < \frac{(r+1)w - \frac{s}{r}w + O(1)}{w} < r - 1$ and thus $l \leq r - 1$. Then $W_\mathsf{A} - lw > 0$ and also, from the definition of $l$, $W_\mathsf{A} - lw \leq k$. The adversary corrupts the top $l$ large and the top $(W_\mathsf{A} - lw)$ small parties.

Let $S_\mathsf{A} = \sum_{i=n-s-(W_\mathsf{A}-lw)+1}^{n-s} t_i$. The adversary corrupts $T_\mathsf{A} = S_\mathsf{A} + \sum_{i=n-l+1}^{n} t_i > S_\mathsf{A} + l(t_n - 1) = S_\mathsf{A} + lt_n - O(1)$ tickets. Let $S = \sum_{i=n-s-k+1}^{n-s} t_i$. The total number of tickets is $T = \sum_{i=1}^{n} t_i \leq S + st_n$. It is required that $S_\mathsf{A} + lt_n - O(1) < T_\mathsf{A} < \frac{r}{s}T \leq \frac{r}{s}(S + st_n) = \frac{r}{s}S + rt_n$. Hence, $(r-l)t_n > S_\mathsf{A} - \frac{r}{s}S - O(1)$. We want to bound $S$ in terms of $S_\mathsf{A}$.

$$S = S_\mathsf{A} + \sum_{i=n-s-k+1}^{n-s-(W_\mathsf{A}-lw)} t_i \leq S_\mathsf{A} + \big(k - (W_\mathsf{A} - lw)\big) \cdot t_{n-s-(W_\mathsf{A}-lw)} \leq$$

$$S_\mathsf{A} + \big(k - (W_\mathsf{A} - lw)\big) \cdot \frac{S_\mathsf{A}}{W_\mathsf{A} - lw} = \frac{k}{W_\mathsf{A} - lw} \cdot S_\mathsf{A}.$$

Thus, $(r-l)t_n > \big(1 - \frac{r}{s} \cdot \frac{k}{W_\mathsf{A}-lw}\big)S_\mathsf{A} - O(1)$ and

$$t_n > \frac{1 - \frac{r}{s} \cdot \frac{k}{W_\mathsf{A}-lw}}{r-l} \cdot S_\mathsf{A} - O(1).$$

Therefore,

$$T_\mathsf{A} > S_\mathsf{A} + lt_n - O(1) \geq$$

$$S_\mathsf{A} + \frac{l}{r-l}\left(1 - \frac{r}{s} \cdot \frac{k}{W_\mathsf{A} - lw}\right)S_\mathsf{A} - O(1) =$$

$$\left(1 + \frac{l}{r-l}\left(1 - \frac{r}{s} \cdot \frac{k}{W_\mathsf{A} - lw}\right)\right)S_\mathsf{A} - O(1)$$

**Claim 1.**

$$1+\frac{l}{r-l}\left(1-\frac{r}{s}\cdot\frac{k}{W_A-lw}\right)\geq 0$$

*Proof.* The inequality is equivalent to

$$(r-l)+l\left(1-\frac{r}{s}\cdot\frac{k}{W_\mathsf{A}-lw}\right)\geq 0 \iff$$

$$r-l\cdot\frac{r}{s}\cdot\frac{k}{W_\mathsf{A}-lw}\geq 0 \iff$$

$$1\geq l\cdot\frac{1}{s}\cdot\frac{k}{W_\mathsf{A}-lw} \iff$$

$$W_\mathsf{A}-lw\geq\frac{lk}{s}.$$

By definition, $l<\frac{W_\mathsf{A}-k}{w}+1$. Therefore, $W_\mathsf{A}-lw>W_\mathsf{A}-(W_\mathsf{A}-k)-w=k-w$. We need to prove that this is greater or equal to $\frac{lk}{s}$ or, equivalently, that $\left(1-\frac{l}{s}\right)k\geq w$.

$\left(1-\frac{l}{s}\right)k\geq\frac{1}{2}k>\frac{1}{2}\left(\frac{s}{r}w-O(1)\right)=\frac{s}{2r}w-O(1)$. Since $\frac{s}{2r}>1$, this is greater than $w$ for sufficiently large $n$. □

Combining Claim 1 and the fact that $S_\mathsf{A}\geq W_\mathsf{A}-lw$, we have

$$T_\mathsf{A}>W_\mathsf{A}-lw+\frac{l}{r-l}\left(W_\mathsf{A}-lw-\frac{r}{s}k\right)-O(1)=$$

$$(r+1-l)w+\frac{l}{r-l}\left((r+1-l)w-\frac{r}{s}k\right)-O(1)=$$

$$\left(r+1-l+\frac{l}{r-l}\left(r+1-l-\frac{r}{s}\cdot\frac{k}{w}\right)\right)w-O(1)=$$

$$\left(r+1+\frac{l}{r-l}\left(1-\frac{r}{s}\cdot\frac{k}{w}\right)\right)w-O(1)=$$

$$\left(r+1-\frac{l}{r-l}\left(\frac{r}{s}\cdot\frac{k}{w}-1\right)\right)w-O(1).$$

If $\frac{r}{s}\cdot\frac{k}{w}-1\leq 0$, Equation 11 is proven. Assume $\frac{r}{s}\cdot\frac{k}{w}-1>0$. By definition, $l=\left\lceil\frac{W_\mathsf{A}-k}{w}\right\rceil=\frac{W_\mathsf{A}-k}{w}+\delta<r+1-\frac{k}{w}+\delta$ for some $0\leq\delta<1$. Then

$$T_\mathsf{A}>\left(r+1-\frac{r}{r-l}\left(\frac{r}{s}\cdot\frac{k}{w}-1\right)\right)w-O(1)=$$

$$\left(r+1-r\cdot\frac{\frac{r}{s}\cdot\frac{k}{w}-1}{\frac{k}{w}-1-\delta}\right)w-O(1)=$$

$$\left(r+1-r\cdot\frac{\frac{r}{s}(\frac{k}{w}-1-\delta)+\frac{r}{s}+\frac{r}{s}\delta-1}{\frac{k}{w}-1-\delta}\right)w-O(1)\ [\geq]$$

since $\delta < 1$ and $\frac{r}{s} < \frac{1}{2}$,

$$[\geq] \left(r + 1 - r \cdot \frac{r}{s}\right) w - O(1) =$$

$$\left(\left(1 - \frac{r}{s}\right) r + 1\right) w - O(1) \geq$$

$$\left(\left(1 - \frac{r}{s}\right) r + 1\right) \frac{\alpha}{r + 1 - s\alpha} n - O(1).$$

Hence, Equation 11 is proven.

It can be easily verified that $\left(\left(1 - \frac{r}{s}\right) r + 1\right) \frac{\alpha}{r+1-s\alpha} > \frac{\alpha}{2}$. Thus, the theorem is proven. □

**Corollary 1.** *Let $0 < \alpha < \beta < 1$ and $0 < \delta < 1 - \alpha/\beta$. For all sufficiently large $n$ there exists a weight vector $\vec{w} = (w_1, ..., w_n)$ such that*

$$\frac{OPT^{T_A}_{\leq\alpha,<(1-\delta)\beta}(\vec{w})}{OPT^{T_A}_{\leq\alpha,<\beta}(\vec{w})} > \Omega(n).$$

*Proof.* Let $s = \left\lceil \frac{1}{\delta\beta} \right\rceil$ and $r = \lceil (1-\delta)\beta s \rceil$. Then $(1-\delta)\beta \leq r/s < \beta$. By Theorem 7 for all sufficiently large $n$ there exists $\vec{w} = (w_1, ..., w_n)$ such that $\mathsf{OPT}^{T_\mathsf{A}}_{\leq\alpha,\leq r/s}(\vec{w}) = O(1)$ and $\mathsf{OPT}^{T_\mathsf{A}}_{\leq\alpha,<r/s}(\vec{w}) > \Omega(n)$. Then

$$\frac{\mathsf{OPT}^{T_\mathsf{A}}_{\leq\alpha,<(1-\delta)\beta}(\vec{w})}{\mathsf{OPT}^{T_\mathsf{A}}_{\leq\alpha,<\beta}(\vec{w})} \geq \frac{\mathsf{OPT}^{T_\mathsf{A}}_{\leq\alpha,<r/s}(\vec{w})}{\mathsf{OPT}^{T_\mathsf{A}}_{\leq\alpha,\leq r/s}(\vec{w})} > \Omega(n).$$

□

# C On optimality of super Swiper's running time

In this section, we address the question whether super Swiper's running time can improved by optimizing the search tree, an example of which is depicted in Figure 3. A search tree contains vertices with labels in $(\mathbb{N} \cup \{_\})^n$ where the root is $(_^n)$ and edges, signifying a call to $\mathsf{DP.apply}()$, connect a parent vertex with label $L$ to a child vertex with label $L'$ such that $L'$ replaces exactly one $_$ in $L$ by a number (the number of tickets). The search tree tests exactly those ticket assignments $\vec{t}$ such that $\vec{t}$ with 0 replaced by $_$ is a label of one of the vertices. We are interested in minimizing the number of edges in the search tree.

We show that for sufficiently large number of parties $n$, there exists a weight distribution such that the sequence of Swiper's ticket assignments $\vec{t_1}, \vec{t_2}, ..., \vec{t_R}$ that terminates requires a tree with $\Theta(n \log n)$ applies. Moreover, $\Theta(n \log n)$ applies are on ticket assignments with $\Omega(n)$ tickets, and therefore, take time $\Omega(n)$. We start by identifying a set of bad ticket assignments.

**Lemma 4.** *Let $\alpha, \beta \in \mathbb{R}$, $r, s \in \mathbb{N}$ such that $0 < \alpha < \beta = r/s < 1$, and let $c \in [0, 1)$ be the constant in Swiper's ticket assignment formula.*

*For sufficiently large $n$, there exists a weight vector $\vec{w} = (w_1, ..., w_n)$ such that any sequence of ticket assignmens $\vec{t_1}, ..., \vec{t_R}$ of increasing size defined according to Swiper rules and terminating with a valid ticket assignment will necessarily contain a ticket assignment $\vec{t} = \left(2^k 1^{l-k} 0^{n-l}\right)$ for all $k \in [0, l]$ where $l = \Theta(n)$.*

*Proof.* Define

$$w = \frac{6 - 2c}{1 - c}; \quad l = s \cdot \left\lfloor \frac{\alpha}{s\big((\beta - \alpha)w + \alpha\big)} n \right\rfloor = \Theta(n);$$

$$w_1 = ... = w_l = w; \quad w_{l+1} = ... = w_n = 1.$$

Denote $\delta$ to be Swiper's scaling factor, so that the number of tickets $t_i$ always satisfies

$$\lfloor \delta w_i + c \rfloor - 1 \leq t_i \leq \lfloor \delta w_i + c \rfloor.$$

The total number of tickets is $W = lw + (n - l)$ and the adversary is allowed to corrupt weight $W_{\mathsf{A}} = \lfloor \alpha l w + \alpha n - \alpha l \rfloor$, and it can be seen that this is at least $\beta l w$. Since $\beta = r/s$ and $l$ is a multiple of $s$, the adversary can claim $\beta$ fraction of the first $l$ weights. Therefore, for a ticket assignment to be valid, one of the small $(n - l)$ weights must have a ticket, but this is impossible while $\delta < 1 - c$.

Hence, we can seen that the large weights must get 2 tickets each before the ticket assignment can be valid: if $\delta = \frac{1-c}{2}$, for $i \in [0, l]$, $t_i \geq \lfloor \delta w_i + c \rfloor - 1 = \left\lfloor \frac{1-c}{2} w_i + c \right\rfloor - 1 \geq 2$. The statement of the lemma follows. □

**Lemma 5.** *For all $n \geq 0$, testing the ticket assignments $\left\{(2^k 1^{n-k}) : k \in [0, n]\right\}$ requires a tree with a minimum of $N(n)$ edges, where $N$ is defined in Equation 4. $N(n)$ is both an upper and lower bound.*

*Proof.* The proof is by induction on $n$. The basis case $n = 0$ is clear. Assume $n \geq 1$ and that the statement is true for all $n'$ with $0 \leq n' < n$.

Take any optimal search tree containing vertices with labels $\left\{(2^k 1^{n-k}) : k \in [0, n]\right\}$. First notice that the labels of all leaf vertices must be from this set and that all vertices must have distinct labels; otherwise, the search tree is not optimal.

We will now modify the search tree, keeping the leaves and the number of edges same, such that each vertex label is of the form $(2^i _^{n-i-j} 1^j)$, as in Figure 3. First, notice that all vertex labels in the tree have separately 2's on the left side and 1's on the right side, or equivalently for a label $l$, $l_i = 2 \wedge l_j = 1$ implies $i < j$; otherwise, the tree is not optimal. Consider the following recursive algorithm fixing vertex labels starting with the root. While the current vertex $v$ has $_$ between 2's, let $i$ be the index of the leftmost 2 that follows a $_$ and $j$ be the index of the leftmost $_$ $(j < i)$, then swap indices $i$ and $j$ in the vertex labels in all of the subtree of $v$; clearly, the search tree remains valid and since originally any apply on index $j$ necessarily had value $t_j = 2$, the leaf labels remain unchanged. Thus, after all iterations, the 2's in $v$'s label are in a consecutive block on the very left. Similarly, we can modify the subtree so that the label of $v$ has the 1's on the very right. After all modifications, recurse on all of the children of $v$.

Now, for all vertices $v$, if its label is $(2^i _^{n-i-j} 1^j)$, define $\text{Range}(v) = [i, n-j]$, denoting the range of the number of 2's that the leaves in $v$'s subtree are allowed to have. Then, for any vertex $v$ with $\text{Range}(v) = [i, n-j]$ and any edge from $v$ to $u$, either a 2 is applied and $\text{Range}(u) = [i+1, n-j]$ or a 1 is applied and $\text{Range}(u) = [i, n-j-1]$. A node $v$ is a leaf iff $|\text{Range}(v)| = 1$. Also, note that since only two applies are allowed (2 on the leftmost _ and 1 on the rightmost _), all vertices can have only 0, 1 or 2 children. Additionally, for all vertices $v$, define $\text{Leaves}(v) = \{\#_u : u \text{ is a leaf in } v\text{'s subtree}\}$ where $\#_u \in \text{Range}(u)$. Note that $\text{Range}(\text{root}) = \text{Leaves}(\text{root}) = [0, n]$.

Finally, we modify the search tree, keeping the leaves and the number of edges same, such that for all vertices $v$, $\text{Leaves}(v) = [l, r]$ for some $0 \le l, r \le n$. Consider the following recursive algorithm starting with the root. Let $v$ be the current vertex and assume it satisfies the condition $\text{Leaves}(v) = [l, r]$ (the very first vertex, the root, already satisfies it). We will modify $v$'s subtree so that $v$'s children also satisfy the condition. If $v$ has zero or one child, there is nothing to do. Otherwise, $v$ has two children: the left child $x$ that has a 1 applied and the right child $y$ that has a 2 applied; we have $\text{Leaves}(v) = [l, r] = \text{Leaves}(x) \cup \text{Leaves}(y)$ and $\text{Leaves}(x) \cap \text{Leaves}(y) = \emptyset$. Letting $k = |\text{Leaves}(x)|$, our goal is to modify the subtrees of $x$ and $y$ such that $\text{Leaves}(x) = [l, l+k)$ and $\text{Leaves}(y) = [l+k; r]$.

**Claim 2.** *Assume* $\text{Leaves}(x)$ *is not of the right form and let* $[l', r']$ *be the rightmost consecutive block of indices in* $\text{Leaves}(x)$*, i.e.,* $[l', r'] \subseteq \text{Leaves}(x)$*,* $(l'-1) \notin \text{Leaves}(x)$*, and for all* $i \in \text{Leaves}(x)$*,* $i \le r'$*. The vertex labels in* $x$*'s subtree can be modified so that* $\text{Leaves}(x)$ *becomes* $\text{Leaves}(x) \setminus [l', r'] \cup [l'-1, r'-1]$.

*Proof.* Define $S = [l', r']$ and consider the following algorithm.

Make a copy of $x$'s subtree. While $S \neq \emptyset$, do the following, letting $i$ be the number of the loop iteration. Let $u_i$ be the leaf with label $\left(2^{\min S}, 1^{n-\min S}\right)$ and define $e_i$ to be the lowest edge that applies a 2 among the edges in the path from $x$ to $u_i$; such an edge must exist since $\text{Leaves}(x)$ contains a number smaller than $l'$. Then, in the copy of $x$'s subtree, have $e_i$ apply a 1 instead of 2 (the labels of the vertices below $e_i$ need to be modified). Finally, letting $z_i$ be the lower endpoint of $e_i$, assign $S \leftarrow S \setminus \text{Leaves}(z_i)$.

Let us argue that the algorithm produces a new subtree for $x$ where $\text{Leaves}(x)$ becomes $\text{Leaves}(x) \setminus [l', r'] \cup [l'-1, r'-1]$. For all $i$, since $e_i$ is the lowest edge on the path from $x$ to $u_i$ that applies a 2, the edges in the path from $z_i$ to $u_i$ all apply 1s, and thus, the number of 2s in $u_i$'s label equals $\min \text{Leaves}(z_i)$; this implies that $\text{Leaves}(z_i) \subseteq [l', r']$. Additionally, for all $i < j$, $\text{Leaves}(z_i) \cap \text{Leaves}(z_j) = \emptyset$ since $z_j$ cannot be in the subtree of $z_i$ by construction and $z_i$ cannot be in the subtree of $z_j$ because the label of $u_j$ has $\left(\min \text{Leaves}(z_j)\right)$ $2s$ but the label of $u_i$ has even fewer $2s$. Combining this with the fact that $\bigcup_i \text{Leaves}(z_i) = [l', r']$, we conclude that each leaf with the number of 2s in $[l', r']$ looses exacty one 2. □

Using Claim 2 iteratively, we modify the vertex labels in the subtree of $x$ such that $\text{Leaves}(x)$ becomes $[l, l+k)$. Analogously, we can modify the vertex labels in the subtree of $y$ such that $\text{Leaves}(y)$ becomes $[l+k; r]$. Recurse on both children of $v$.

We are almost done. Observe that the root vertex has exactly two children, denote them $x$ and $y$ with labels $(_^{n-1}1)$ and $(2_^{n-1})$ respectively; otherwise, Leaves(root) is incomplete. There exists $m \geq 1$ such that Leaves$(x) = [0, m)$ and Leaves$(y) = [m, n]$. The number of edges in the subtree of $x$ is exactly $n - m + N(m-1)$ since it is optimal to first apply $(n-m)$ 1s and then build an optimal search tree with the leaves of $x$. Similarly, the number of edges in the subtree of $y$ is exactly $m - 1 + N(n-m)$. Therefore, the total number of edges in the whole tree is $1 + \big(n - m + N(m-1)\big) + 1 + \big(m - 1 + N(n-m)\big) = n + 1 + N(m-1) + N(n-m)$. Moreover, by optimality, the total number of edges must equal

$$\min_{m:1 \leq m \leq n} \big\{n + 1 + N(m-1) + N(n-m)\big\};$$

by Lemma 2, this is equal to

$$\begin{cases} 2t + 2 + 2N(t) & \text{if } n = 2t+1 \text{ for some } t \\ 2t + 1 + N(t) + N(t-1) & \text{if } n = 2t \text{ for some } t. \end{cases}$$

By definition of $N$, the total number of edges is $N(n)$. □

We now present the result of this section.

**Theorem 8.** *Let $\alpha, \beta \in \mathbb{R}$, $r, s \in \mathbb{N}$ such that $0 < \alpha < \beta = r/s < 1$, and let $c \in [0, 1)$ be the constant in Swiper's ticket assignment formula.*

*For sufficiently large $n$, there exists a weight vector $\vec{w} = (w_1, ..., w_n)$ such that the sequence of Swiper's ticket assignments $\vec{t_1}, \vec{t_2}, ..., \vec{t_R}$ that terminates requires a tree with $\Theta(n \log n)$ applies. Moreover, $\Theta(n \log n)$ applies are on ticket assignments with $\Omega(n)$ tickets.*

*Proof.* By Lemma 4, there exists a weight distribution such that $\big\{(2^k 1^{l-k} 0^{n-l}) : k \in [l]\big\} \subseteq \big\{\vec{t_1}, \vec{t_2}, ..., \vec{t_R}\big\}$ where $l = \Theta(n)$. By Lemma 5, $N(l) = N(\Theta(n))$ applies is required, where $N$ is defined in Equation 4. By Lemma 1, this is $\Theta(n \log n)$.

We now argue that $\Theta(n \log n)$ applies are on ticket assignments with $\Omega(n)$ tickets. For any leaf vertex, the path from the root to it has exactly $l$ edges. Partitioning the set of all edges based on the distance from the root, we see that the partition $\{P_i\}_{i \in [l]}$ satisfies $|P_1| \leq |P_2| \leq ... \leq |P_l|$. Then $\bigcup_{i:k<i\leq l} P_i$, where $k = \lfloor l/2 \rfloor$, is a set of $\geq \frac{l-k}{l} \cdot \big|\bigcup_{i \in [l]} P_i\big| = \Theta(n \log n)$ edges where each edge does an apply on a ticket assignment of size $\geq k = \Theta(n)$. □