The Harassed Waitress Problem

Harrah Essed Wei Therese

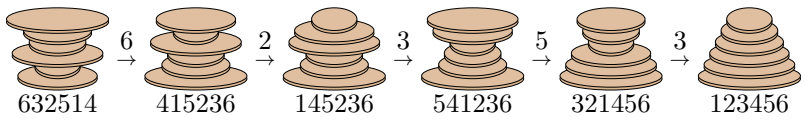
Italian House of Pancakes

Abstract. It is known that a stack of n pancakes can be rearranged in all $n!$ ways by a sequence of $n!-1$ flips, and that a stack of n 'burnt' pancakes can be rearranged in all $2^n n!$ ways by a sequence of $2^n n!-1$ flips. Unfortunately, the known algorithms are too difficult to be used by the waitstaff of a busy restaurant. How can humans can determine the next flip from the current stack and no extra information? We provide such successor rules that run in $O(n)$-time using no memory. More broadly, we discuss how iteration and computational complexity provide helpful constraints when solving Hamilton cycle problems in highly symmetric graphs, and how simple greedy algorithms can produce globally optimal Gray codes.

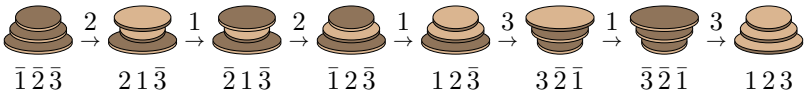
Keywords: pancake sorting, greedy algorithm, Gray code, permutations, prefix-reversal, symmetric group, Cayley graph, Hamilton cycle

1 Introduction

Jacob Goodman, writing under the name Harry Dweighter (*"harried waiter"*), introduced the original pancake problem: Given a stack of n pancakes of various sizes, what is the minimum number of flips required to sort the pancakes from smallest to largest? In this problem, the individual pancakes are numbered $1, 2, \ldots, n$ by increasing size; a stack of pancakes can be represented by a permutation in one-line notation. Each 'flip' of the topmost i pancakes corresponds to a *prefix-reversal* of length i in the permutation. For example, the following illustration shows how the stack 632514 can be sorted in 5 flips:



A well-studied variation features 'burnt' pancakes, which have two distinct sides. In this problem, a stack is represented by a *signed permutation* in one-line notation, with i and $\bar{i}$ being used when the burnt side of pancake i is facing down or up, respectively. Each 'flip' of the topmost i pancakes corresponds to a *sign-complementing prefix-reversal* in the signed permutation. For example, the following illustration shows how the stack $\bar{3}\,\bar{2}\,\bar{1}$ can be sorted in 7 flips:



Recently it was shown that Goodman's original *harried waiter problem* is NP-hard to solve in general [2] while the complexity of the burnt variation is unknown. If arbitrary substacks are allowed to be flipped, then the unburnt sorting problem is APX-hard [1] and the burnt sorting problem can be solved in polynomial-time [4].

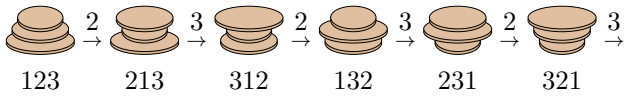
Research on pancake sorting had humble beginnings — Goodman formulated the problem while sorting a stack of towels — but has a number of interesting applications including genomics (see Fertin et al [3]) and in vivo computing (see Haynes [5] for an introduction to the 'e.Hop' restaurant), and has been discussed by the media (see Singh [9]).

1.1 The Harassed Waitress Problem

Zaks [16] asked the following question: *Can a stack of n pancakes be rearranged in all $n!$ ways by a sequence of $n! - 1$ flips?* To differentiate this problem from Goodman's, we refer to it as the *harassed waitress problem*. The following passage from [16] explains its relevant results:

> Using our algorithms the poor ~~waiter~~ waitress will be able to generate, in $n!$ such steps, all possible $n!$ stacks (returning to the original one) ... in $(k-1)/k!$ of them he will reverse the top k pancakes, which amounts to less than 2.8 pancakes reversed on the average.

For example, Zaks's solution for $n = 3$ is as follows:



Zaks's result is a *Gray code of permutations using prefix-reversals*, and the Gray code is *cyclic* since the first and last stacks differ by a prefix-reversal. Equivalently, Zaks's solution gives a Hamilton cycle in the *pancake network*, whose vertices are the permutations of n with adjacencies between those that differ by a prefix-reversal. The simplicity of Zaks's solution is interesting given the fact that the shortest path problem in this graph is NP-hard.

As with Goodman's problem, there is also a natural 'burnt' variation. The underlying graph is the *burnt pancake network*, and successful orders are *Gray codes of signed permutations using sign-complementing prefix-reversals*.

The aforementioned solutions can be generated one stack at a time by efficient algorithms. Unfortunately, the algorithms are designed for computers. We would like to have a simple *successor rule* that maps each stack to the next stack in a particular solution. More specifically, we are interested in the following question:

> **How efficiently can we compute the next flip from the current stack with no additional information given?**

To motivate this question it is helpful to focus on the harassed waitress. We suppose that our heroine is working at a busy restaurant and may need to stop

Fig. 1. The most important question for solving the harassed waitress problem.

and restart her task many times. These interruptions do not afford her the luxury of recalling the context of the previous flips made – she has no memory!

Another issue one may consider is the total number of pancakes that the waitress must flip throughout a given solution. In particular, we are interested in solutions that flip either the minimum or maximum possible number of pancakes overall (or equivalently the average number of pancakes in each flip). Un-*fun*-tunately, we do not have the space to address this issue.

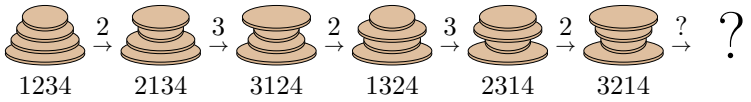
1.2 New Results

We provide four results (assume worst-case analysis unless specified).

1. With a minimum-flip strategy, our waitress can determine how many pancakes to flip at each step in $O(n)$-time. On average, she uses $O(1)$-time.
2. With a minimum-flip strategy, our waitress can determine how many burnt pancakes to flip at each step in $O(n)$-time. On average, she uses $O(1)$-time.
3. With a maximum-flip strategy, our waitress can determine how many pancakes to flip at each step in $O(n)$-time. On average, she uses $O(1)$-time if she considers two flips at a time.
4. With a maximum-flip strategy, our waitress can determine how many burnt pancakes to flip at each step in $O(n)$-time. On average, she uses $O(1)$-time if she considers two flips at a time.

Our results are focused on the complexity of determining the next flip and not performing the flip in a data structure (see [12] for a *fun* $O(1)$-time implementation of prefix-reversals). The results are based on four greedy algorithms given by Sawada and Williams [8, 7]. The algorithms build a list of stacks one at a time, starting from $1\,2\cdots n$. The next stack is created by taking the last stack in the list and applying the 'best' flip that creates a 'new' stack. In this

context 'new' means that the stack is not already in the list, and 'best' means minimum or maximum depending on the algorithm. The new stack is appended to the list, and the algorithm terminates when a new stack cannot be created. For example, let us illustrate one step of the minimum flip algorithm when $n = 4$ starting from the following list:



We cannot flip the top two pancakes of 3214 since 2314 is already in the list. Similarly, we cannot flip the top three pancakes since 1234 is already in the list. However, we can flip the top four pancakes, and so the resulting new stack 4123 is added to the list. Eventually, this approach lists all stacks. All four greedy algorithms are illustrated by Table 1 in the Appendix. While these greedy descriptions are simple, they are only practical for waitresses with photographic memories! Just for *fun*, we implemented our successor algorithms in C and included them in the Appendix.

2 Successor Rules For Four Greedy Flip Strategies

For each of the greedy flip strategies to list stacks of (burnt) pancakes, we recall the recursive definitions provided in [7]. These recursive definitions are used to prove the correctness of the successor rules. First, some notation is required.

Let $\mathbb{P}(n)$ denote the set of permutations of $\{1, 2, \ldots, n\}$ and let $\overline{\mathbb{P}}(n)$ denote the set of signed permutations of $\{1, 2, \ldots, n\}$. For example, $\mathbb{P}(3) = \{123, 132, 213, 231, 312, 321\}$ and $\overline{\mathbb{P}}(2) = \{12, 21, \bar{1}2, 2\bar{1}, 1\bar{2}, \bar{2}1, \bar{1}\bar{2}, \bar{2}\bar{1}\}$. Given a (signed) permutation $\mathbf{p} = p_1 p_2 \cdots p_n$, we will use the following notation:

- $\mathsf{flip}_j(\mathbf{p}) = p_j p_{j-1} \cdots p_1 p_{j+1} \cdots p_n$, a flip (prefix reversal) of length j,
- $\overline{\mathsf{flip}}_j(\mathbf{p}) = \bar{p}_j \bar{p}_{j-1} \cdots \bar{p}_1 p_{j+1} \cdots p_n$, a signed flip (prefix reversal) of length j,
- $\mathbf{p} \cdot n$ denotes the concatenation of the symbol n to the permutation $\mathbf{p}$.

2.1 Minimum flip for permutations

Given $\mathbf{p} = p_1 p_2 \cdots p_n \in \mathbb{P}(n)$, let $\mathbf{q}_i = p_{i+1} \cdots p_n p_1 \cdots p_{i-1}$ denote a rotation of the permutation $\mathbf{p}$ with the element p_i removed. Consider the following definition:

$$\mathsf{Min}(\mathbf{p}) \;=\; \mathsf{Min}(\mathbf{q}_n) \cdot p_n,\; \mathsf{Min}(\mathbf{q}_{n-1}) \cdot p_{n-1}, \ldots,\; \mathsf{Min}(\mathbf{q}_1) \cdot p_1, \qquad (1)$$

with base case $\mathsf{Min}(p_1) = p_1$ when $n = 1$. This recursive listing corresponds to a greedy minimum flip strategy [7] for permutations, where the first and last strings differ by flip_n. It is used to prove the correctness of the upcoming successor rule.

A permutation $\mathbf{p} \in \mathbb{P}(n)$ is *increasing* if it corresponds to a rotation of the word $12 \cdots n$. It is *decreasing* if it is a reversal of an increasing permutation. Specifically, the set of all n increasing permutations is:

$$\{12 \cdots n, 23 \cdots n1, 34 \cdots n12, \ldots, n12 \cdots n{-}1\}.$$

A k-permutation is any string of length k over the set $\{1, 2, 3, \ldots, n\}$ with no repeating symbols. A k-permutation is *increasing* (*decreasing*) if it is a subsequence of an increasing (decreasing) permutation. For instance, 5124 is increasing, but 5127 is not.

Remark 1. If $\mathbf{p}$ is increasing (decreasing) then both $\mathsf{flip}_{n-1}(\mathbf{p})$ and $\mathsf{flip}_n(\mathbf{p})$ are decreasing (increasing).

Given a permutation $\mathbf{p}'$, let $\mathsf{succ}(\mathbf{p}')$ denote the successor of $\mathbf{p}'$ in $\mathsf{Min}(\mathbf{p})$ when the listing is considered to be circular.

Lemma 1. *Let* $\mathbf{p}' = p'_1 p'_2 \cdots p'_n$ *be a permutation in the (circular) listing* $\mathsf{Min}(\mathbf{p})$*, where* $\mathbf{p} = p_1 p_2 \cdots p_n$ *is increasing. Then:*

$\mathsf{succ}(\mathbf{p}') = \mathsf{flip}_j(\mathbf{p}')$*, where* $p'_1 p'_2 \cdots p'_j$ *is the longest prefix of* $\mathbf{p}'$ *that is decreasing.*

Proof. We focus on the permutations whose successor is the result of a flip of size n and then apply induction (the base case when $n = 2$ is easily verified). Consider the recursive definition for $\mathsf{Min}(\mathbf{p})$ in (1). Given a permutation $\mathbf{p}'$, its successor will be $\mathsf{flip}_n(\mathbf{p}')$ if and only if it is the last permutation in one of the recursive listings of the form $\mathsf{Min}(\mathbf{q}_i) \cdot p_i$. Clearly, at most one permutation in each recursive listing can be decreasing. By showing that the last permutation in each listing is the one that is decreasing, we verify the successor rule for flips of size n.

We are given that the initial permutation is increasing. Also, note that the last permutation in $\mathsf{Min}(\mathbf{q}_n) \cdot p_n$ is $\mathsf{flip}_{n-1}(\mathbf{p})$. Thus, by Remark 1 this last permutation is decreasing. By applying the flip of size n to this last permutation, Remark 1 implies that the resulting permutation, which is the first permutation of $\mathsf{Min}(\mathbf{q}_{n-1}) \cdot p_{n-1}$, will be increasing. Repeating this argument for $i = n-1, n-2, \ldots, 1$ verifies our claim that the last permutation in each recursive listing is decreasing; it is true for the final recursive listing since the last permutation in $\mathsf{Min}(\mathbf{p})$ differs from the first by a flip of size n.

Thus, the successor rule is correct for all permutations whose successor is the result of a flip of size n. For all other permutations whose successor is not a flip of size n, the successor rule follows from induction. □

As an example, consider the permutation 3764512 with respect to the listing $\mathsf{Min}(12 \cdots n)$. The prefix 3764 is the longest one that is decreasing, thus $j = 4$ and the next permutation in the listing is $\mathsf{flip}_4(3764512)$. Determining the value j in this successor rule can easily be determined in $O(n)$ time by applying the pseudocode given in Algorithm 1.

Theorem 1. Successor($\mathbf{p}$) *returns the size of the flip required to obtain the successor of* $\mathbf{p}$ *in the (circular) listing* $\mathsf{Min}(12 \cdots n)$ *in* $O(n)$ *time.*

This function runs in expected $O(1)$ time when the permutation is passed by reference because the average flip size is bounded above by the constant e [7]. Thus, by repeatedly applying this successor rule, our waitress can iterate through all $n!$ stacks of pancakes in constant amortized time starting from $\mathbf{p} = 12 \ldots n$. She will return to the initial stack after she completes a flip of size n and the top pancake $p_1 = 1$.

Algorithm 1 Computing the successor of $\mathbf{p}$ in the listing $\mathsf{Min}(12\cdots n)$

```
1: function SUCCESSOR(p)
2:     incr ← 0
3:     for j ← 1 to n − 1 do
4:         if p_j < p_{j+1} then incr ← incr + 1
5:         if incr = 2 or (incr = 1 and p_{j+1} < p_1) then return j
6:     return n
```

2.2 Minimum Flips for Signed Permutations

A recursive formulation for signed permutations is similar to the formulation for the non-signed case with a minor change to some notation. Let $\mathbf{q} = q_1q_2\cdots q_{2n} = \bar{p}_1\bar{p}_2\cdots\bar{p}_np_1p_2\cdots p_n$ be a circular string of length $2n$. Let $\mathbf{q_i}$ denote the length $n-1$ subword ending with q_{i-1}. For instance, $\mathbf{q_3} = p_4p_5\cdots p_n\bar{p}_1\bar{p}_2$. Consider the following recursive definition:

$$\overline{\mathsf{Min}}(\mathbf{p}) = \overline{\mathsf{Min}}(\mathbf{q_{2n}})\cdot q_{2n},\ \overline{\mathsf{Min}}(\mathbf{q_{2n-1}})\cdot q_{2n-1},\ldots,\ \overline{\mathsf{Min}}(\mathbf{q_1})\cdot q_1, \tag{2}$$

where $\overline{\mathsf{Min}}(p_1) = p_1, \bar{p}_1$. This listing corresponds to a greedy minimum flip strategy [7] for signed permutations, where the first and last strings differ by a flip of size n.

We say a signed permutation $\mathbf{p} \in \overline{\mathbb{P}}(n)$ is *increasing* if it corresponds to a length n subword of the circular string $\bar{1}\bar{2}\cdots\bar{n}12\cdots n$. It is *decreasing* if it is a reversal of an increasing permutation. For example, the set of all $2n$ increasing signed permutations is

$$\{\bar{1}\bar{2}\bar{3}\cdots\bar{n}, \bar{2}\bar{3}\cdots\bar{n}1, \bar{3}\bar{4}\cdots\bar{n}12, \ldots, n\bar{1}\cdots\overline{n-1}\}.$$

A signed k-permutation is any string of length k over the set $\{1, 2, \ldots, n, \bar{1}, \bar{2}, \ldots\bar{n}\}$ with no repeating symbols when taking absolute value. A signed k-permutation is *increasing* (*decreasing*) if it is a subsequence of an increasing (decreasing) signed permutation. For example, $567\bar{2}\bar{4}$ is increasing, but $\bar{4}567$ is not.

Remark 2. If a signed permutation $\mathbf{p}$ is increasing (decreasing) then both $\overline{\mathsf{flip}}_{n-1}(\mathbf{p})$ and $\overline{\mathsf{flip}}_n(\mathbf{p})$ are decreasing (increasing).

Given a signed permutation $\mathbf{p'}$, let $\mathsf{succ}(\mathbf{p'})$ denote the successor of $\mathbf{p'}$ in $\overline{\mathsf{Min}}(\mathbf{p})$ when the listing is considered to be circular. A proof of the following lemma uses Remark 2 and follows the exact same inductive style as the proof for Lemma 1.

Lemma 2. *Let $\mathbf{p'} = p'_1p'_2\cdots p'_n$ be a signed permutation in the (circular) listing $\overline{\mathsf{Min}}(\mathbf{p})$, where $\mathbf{p} = p_1p_2\cdots p_n$ is increasing. Then:*

$\mathsf{succ}(\mathbf{p'}) = \mathsf{flip}_j(\mathbf{p'})$, *where $p'_1p'_2\cdots p'_j$ is the longest prefix of $\mathbf{p'}$ that is decreasing.*

Pseudocode for such a successor function is given in Algorithm 2.

Algorithm 2 Computing the successor of $\mathbf{p}$ in the listing $\overline{\mathsf{Min}}(12\cdots n)$

```
1: function SUCCESSOR(p)
2:     incr ← 0
3:     for j ← 1 to n − 1 do
4:         if |p_j| < |p_{j+1}| then incr ← incr + 1
5:         if incr = 2 or (incr = 1 and |p_{j+1}| < |p_1|) then return j
6:         if |p_j| < |p_{j+1}| and SIGN(p_j) = SIGN(p_{j+1}) then return j
7:         if |p_j| > |p_{j+1}| and SIGN(p_j) ≠ SIGN(p_{j+1}) then return j
8:     return n
```

Theorem 2. SUCCESSOR($\mathbf{p}$) *returns the size of the flip required to obtain the successor of* $\mathbf{p}$ *in the listing* $\overline{\mathsf{Min}}(12\cdots n)$ *in* $O(n)$ *time.*

Observe that this function runs in expected $O(1)$ time when the permutation is passed by reference because the average flip size is bounded above by the constant $\sqrt{e}$ [7]. Thus, by repeatedly applying this successor rule, our waitress can iterate through all $2^n n!$ stacks of burnt pancakes in constant amortized time starting from $\mathbf{p} = 12\ldots n$. She will return to the initial stack after she completes a flip of size n and the top pancake $p_1 = 1$.

2.3 Maximum Flips for Permutations

Define the *bracelet order* of permutation $\mathbf{p_1} \in \mathbb{P}(n)$ as:

$$\mathsf{brace}(\mathbf{p_1}) = \mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_{2n}} \text{ such that } \mathbf{p_i} = \begin{cases} \mathsf{flip}_n(\mathbf{p_{i-1}}) & \text{if } i \text{ is even} \\ \mathsf{flip}_{n-1}(\mathbf{p_{i-1}}) & \text{if } i > 1 \text{ is odd.} \end{cases}$$

The last string in $\mathsf{brace}(\mathbf{p_1})$ is $\mathsf{flip}_{n-1}(\mathbf{p_1})$. A *bracelet class* is a set containing the strings in a bracelet order $\mathsf{brace}(\mathbf{p_1})$. The following lemma is proved in [7]:

Lemma 3. *If* $\mathbf{p_1}$ *and* $\mathbf{p_2}$ *are distinct permutations in* $\mathbb{P}(n-1)$*, then* $\mathbf{p_1} \cdot n$ *and* $\mathbf{p_2} \cdot n$ *are in the same bracelet class if and only if* $\mathbf{p_2} = \mathsf{flip}_{n-1}(\mathbf{p_1})$*.*

We now give a recursive definition to list $\mathbb{P}(n)$:

$$\mathsf{Max}(n) = \mathsf{brace}(\mathbf{q_1} \cdot n), \mathsf{brace}(\mathbf{q_3} \cdot n), \mathsf{brace}(\mathbf{q_5} \cdot n), \ldots, \mathsf{brace}(\mathbf{q_{m-1}} \cdot n), \quad (3)$$

where $\mathsf{Max}(n-1) = \mathbf{q_1}, \mathbf{q_2}, \ldots, \mathbf{q_m}$ and $\mathsf{Max}(1) = 1$. This listing corresponds to a greedy maximum flip strategy [7] for permutations, where the first and last strings differ by a flip of size 2. The recursive definition is used to prove the correctness of the upcoming successor rule.

Given a permutation $\mathbf{p} = p_1 p_2 \cdots p_n$, let $\mathsf{succ}(\mathbf{p})$ denote the successor of $\mathbf{p}$ in $\mathsf{Max}(n)$. One may observe that every second permutation in $\mathsf{Max}(n)$, starting with the first, contains the subsequence 123, 231, or 312; or in other words, they contain the subsequence 123 when $\mathbf{p}$ is considered circularly. If a permutation contains such a subsequence we say it has *property* $\overrightarrow{123}$.

Lemma 4. *For $n \geq 3$:*

$$\mathsf{succ}(\mathbf{p}) = \begin{cases} \mathsf{flip}_n(\mathbf{p}) & \text{if } \mathbf{p} \text{ has property } \overrightarrow{123} \\ \mathsf{flip}_{max(j-1,2)}(\mathbf{p}) & \text{otherwise,} \end{cases} \tag{4}$$

where j is the largest index such that $p_j \neq j$.

Proof. This successor rule is easy to verify for $n = 3$. By induction, assume the successor rule is correct for $\mathsf{Max}(n-1)$, where $n > 3$. Additionally, by induction, assume the rule is correct when applied to the first $r-1$ permutations in $\mathsf{Max}(n)$. We must show that the successor of permutation $\mathbf{p} = p_1 p_2 \cdots p_n$ at rank r is given by (4). Observe that the first r permutations will alternately have, and not have the property $\overrightarrow{123}$. This is because (4) always flips at least two of the values 1,2, and 3. Thus, $\mathbf{p}$ has property $\overrightarrow{123}$ if and only if r is odd. We consider two cases depending on whether r is odd or even.

If r is odd, we have established that $\mathbf{p}$ has property $\overrightarrow{123}$. By (3) and the definition of a bracelet class, $\mathsf{succ}(\mathbf{p}) = \mathsf{flip}_n(\mathbf{p})$, which verifies (4).

If r is even, we have established that $\mathbf{p}$ does not have property $\overrightarrow{123}$. Consider two cases depending on the last element p_n. If $p_n \neq n$, then by Lemma 3, $\mathbf{p}$ will not be the last permutation in a bracelet class from (3) and thus $\mathsf{succ}(\mathbf{p}) = \mathsf{flip}_{n-1}(\mathbf{p})$, which verifies (4). If $p_n = n$, then r being even implies that $\mathbf{p}$ is the last permutation in a bracelet class from (3) by Lemma 3. Thus, $\mathsf{succ}(\mathbf{p})$ will correspond to $\mathsf{succ}(p_1 p_2 \cdots p_{n-1})$ in $\mathsf{Max}(n-1)$ with n appended to the end. Since $p_1 p_2 \cdots p_{n-1}$ does not have property $\overrightarrow{123}$, by induction $\mathsf{succ}(p_1 p_2 \cdots p_{n-1}) = \mathsf{flip}_{max(j-1,2)}(p_1 p_2 \cdots p_{n-1})$ where j is the largest index such that $p_j \neq j$. Thus, since $p_n = n$, $\mathsf{succ}(\mathbf{p})$ is equal to $\mathsf{flip}_{max(j-1,2)}(\mathbf{p})$ where j is the largest index such that $p_j \neq j$, satisfying (4). □

Pseudocode for a successor rule based on this lemma is given in Algorithm 3.

Algorithm 3 Computing the successor of $\mathbf{p}$ in the listing $\mathsf{Max}(n)$

```
1: function SUCCESSOR(p)
2:     for j ← 1 to n do
3:         if p_j = 1 then pos_1 ← j
4:         if p_j = 2 then pos_2 ← j
5:         if p_j = 3 then pos_3 ← j
6:     if (pos_1 < pos_2 < pos_3) or (pos_2 < pos_3 < pos_1) or (pos_3 < pos_1 < pos_2)
   then return n
7:     j ← n
8:     while p_j = j and j > 3 do j ← j − 1
9:     return j − 1
```

Theorem 3. SUCCESSOR*(*$\mathbf{p}$*)* *returns the successor of the permutation* $\mathbf{p}$ *in the listing* $\mathsf{Max}(n)$ *in* $O(n)$ *time.*

By applying the observations from this successor rule, our waitress can apply a very simple and elegant algorithm to generate $\mathsf{Max}(n)$. The main idea is to visit two permutations at a time; pseudocode is given in Algorithm 4. Since the average flip length approaches $n-\frac{1}{2}$, the while loop iterates less than once on average. Thus, this simple algorithm runs in constant amortized time per flip.

Algorithm 4 Exhaustive algorithm to list the ordering $\mathsf{Max}(n)$ of $\mathbb{P}(n)$

```
1: procedure GEN
2:     p ← 12···n
3:     repeat
4:         VISIT(p)
5:         p ← flip_n(p)
6:         VISIT(p)
7:         j ← n
8:         while p_j = j do  j ← j − 1
9:         p ← flip_{j−1}(p)
10:    until j = 2
```

2.4 Maximum Flips for Signed Permutations

Define the *signed bracelet order* of permutation $\mathbf{p_1} \in \overline{\mathbb{P}}(n)$ as:

$$\overline{\mathsf{brace}}(\mathbf{p_1}) = \mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_{4n}} \text{ such that } \mathbf{p_i} = \begin{cases} \overline{\mathsf{flip}}_n(\mathbf{p_{i-1}}) & \text{if } i \text{ is even} \\ \overline{\mathsf{flip}}_{n-1}(\mathbf{p_{i-1}}) & \text{if } i > 1 \text{ is odd.} \end{cases}$$

Using this definition, we arrive at a similar recurrence to list $\overline{\mathbb{P}}(n)$ as the unsigned case in the previous section:

$$\overline{\mathsf{Max}}(n) = \mathsf{brace}(\mathbf{q_1} \cdot n), \mathsf{brace}(\mathbf{q_3} \cdot n), \mathsf{brace}(\mathbf{q_5} \cdot n), \ldots, \mathsf{brace}(\mathbf{q_{m-1}} \cdot n), \quad (5)$$

where $\overline{\mathsf{Max}}(n-1) = \mathbf{q_1}, \mathbf{q_2}, \ldots, \mathbf{q_m}$ and $\overline{\mathsf{Max}}(1) = 1, \bar{1}$. This listing corresponds to a greedy maximum flip strategy [7] for signed permutations, where the first and last strings differ by a flip of size 1.

Given a permutation $\mathbf{p} = p_1 p_2 \cdots p_n$, let $\mathsf{succ}(\mathbf{p})$ denote the successor of $\mathbf{p}$ in $\overline{\mathsf{Max}}(n)$. To find an efficient successor rule for this listing, observe that every second permutation, starting with the first, contains the subsequence $12, 2\bar{1}, \bar{1}\bar{2}$, or $\bar{2}1$. If a permutation contains such a subsequence we say it has *property* $\overrightarrow{12}$.

Lemma 5. *For $n \geq 2$:*

$$\mathsf{succ}(\mathbf{p}) = \begin{cases} \mathsf{flip}_n(\mathbf{p}) & \textit{if } \mathbf{p} \textit{ has property } \overrightarrow{12} \\ \mathsf{flip}_{max(j-1,1)}(\mathbf{p}) & \textit{otherwise,} \end{cases} \quad (6)$$

where j is the largest index such that $p_j \neq j$.

Algorithm 5 Computing the successor of $\mathbf{p}$ in the listing $\overline{\mathsf{Max}}(n)$

```
1: function SUCCESSOR(p)
2:     for j ← 1 to n do
3:         if |p_j| = 1 then pos_1 ← j
4:         if |p_j| = 2 then pos_2 ← j
5:     if pos_1 < pos_2 and SIGN(p_{pos_1}) = SIGN(p_{pos_2}) then return n
6:     if pos_1 > pos_2 and SIGN(p_{pos_1}) ≠ SIGN(p_{pos_2}) then return n
7:     j ← n
8:     while p_j = j and j > 2 do j ← j − 1
9:     return j − 1
```

A proof of this lemma is similar to the one for Lemma 4. Pseudocode for a successor rule based on this lemma is given in Algorithm 5.

Theorem 4. SUCCESSOR*(*$\mathbf{p}$*)* *returns the successor of the permutation* $\mathbf{p}$ *in the listing* $\overline{\mathsf{Max}}(n)$ *in* $O(n)$ *time.*

By applying the observations from this successor rule, our waitress can apply a simple and elegant algorithm to generate $\overline{\mathsf{Max}}(n)$. The main idea is to consider two consecutive pancake stacks; pseudocode is given in Algorithm 6. Since the average flip length approaches $n-\frac{1}{2}$, the while loop iterates less than once on average. Thus, this simple algorithm runs in constant amortized time per flip.

Algorithm 6 Exhaustive algorithm to list the ordering $\overline{\mathsf{Max}}(n)$ of $\overline{\mathbb{P}}(n)$

```
1: procedure GEN
2:     p ← 12···n
3:     repeat
4:         VISIT(p)
5:         p ← flip_n(p)
6:         VISIT(p)
7:         j ← n
8:         while p_j = j do j ← j − 1
9:         p ← flip_{j−1}(p)
10:    until j = 1
```

3 The Bigger Picture

A classic conjecture attributed to Lovász is the following: *Every connected vertex-transitive graph has a Hamilton path.* Several well-known variations of this conjecture exist including the following: *Every connected Cayley graph has a Hamilton cycle.* Despite significant attention, these conjectures have proven to be quite stubborn. For this reason, there is value in developing novel approaches. One such

approach to develop a suitable successor rule as the **first step**. For example, our heroine could create a rule for modifying a stack of pancakes, and then determine if it creates all possible stacks. Although this approach involves trial and error, and equal parts of art and science, it has lead to a number of recent successes:

1. *Cool-lex order.* The following rule uses rotations to cyclically create all $\binom{n}{w}$ binary strings of length n and weight w: *Rotate the shortest prefix ending in* 010 *or* 011 *one position to the right (or the entire string if there is no such prefix).* The rule runs in amortized $O(1)$-time with no additional storage, and $O(1)$-time with $O(\log n)$ bits of memory that can be recomputed in amortized $O(1)$-time or worst-case $O(n)$-time. This result has led to applications involving computer words, binary strings, multiset permutations, k-ary trees, necklaces and Lyndon words, fixed-weight de Bruijn sequences, and bubble languages. For a 'fun' introduction see Stevens and Williams [10, 11].
2. *The sigma-tau Gray code.* A simple generating set for the symmetric group S_n is the rotation $\sigma = (1\ 2\ \cdots\ n)$ and the swap of the first two symbols $\tau = (1\ 2)$. The directed Cayley graph does not contain a Hamilton cycle for odd values of n and the remaining Hamiltonicity problems were open for forty years (see Problem 6 in [6]). Williams [14] recently solved the problems with successor rules that can be applied in worst-case $O(n)$-time with no additional storage, or worst-case $O(1)$-time with $O(\log n)$ bits of memory that can be recomputed in worst-case $O(n)$-time.
3. *A new de Bruijn sequence.* k-ary de Bruijn sequences are in one-to-one correspondence with Eulerian cycles in the k-ary de Bruijn graph. Equivalently, they are in one-to-one correspondence with Hamilton cycles in the corresponding line graph. Recently, Wong discovered a simple successor rule for creating such a Hamilton cycle when $k = 2$ [15]: Given a current string $b_1 b_2 \cdots b_n$ the next string is $b_2 b_3 \cdots b_n \overline{b_1}$ if $b_2 b_3 \cdots b_n 1$ is a necklace, and otherwise the next vertex is the rotation $b_2 b_3 \cdots b_n b_1$. The successor rule can be generalized to arbitrary k, and the result generates each symbol of a new de Bruijn sequence in $O(n)$-time using no additional memory.

Solving mathematical problems often reduces to choosing the right type of constraints. A key ingredient to developing the above results was computational complexity. More specifically, the authors considered aggressive measures of efficiency to ensure that only the simplest possible successor rules were considered. For example, we have mentioned several successor rules for permutations that run in $O(1)$-time and use $O(\log n)$ bits of additional memory. This is significant because the rules cannot uniquely determine the permutation they are being applied to! Thus, the rule must implicitly group the permutations into nontrivial equivalence classes, and must exploit symmetries in the graph to function properly. More generally, the authors' underlying assumption is the following:

> *If a Hamilton graph has 'simple' description, then at least one of its Hamilton paths or cycles has a 'simple' successor rule.*

To investigate this assumption it will be helpful to build a catalogue of successor rules and their computational complexities. The entries given by this article are

particularly interesting because the associated shortest path problem is NP-hard, the Gray codes are conjectured to unique in a greedy sense [7], and the *fun* story helps us focus on the importance of simplicity. Eventually, the authors believe that theorems of the following form will be developed: *If a graph is of type X, then it has a Hamiltonian successor rule with computational complexity Y.*

References

1. P. Berman and M. Karpinski. On some tighter inapproximability results. *Lecture Notes in Computer Science (ICALP 1999)*, 1644:200–209, 1999.
2. L. Bulteau, G. Fertin, and I. Rusu. Pancake flipping is hard. In B. Rovan, V. Sassone, and P. Widmayer, editors, *Mathematical Foundations of Computer Science 2012*, volume 7464 of *Lecture Notes in Computer Science*, pages 247–258. Springer Berlin Heidelberg, 2012.
3. G. Fertin, A. Labarre, I. Rusu, E. Tannier, and S. Vialette. *Combinatorics of Genome Rearrangements*. MIT Press, August 2009.
4. S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM*, 46(1):1–27, 1999.
5. K. A. Haynes. We Flip Them For You! E. coli House of Pancakes.
6. A. Nijenhuis and H. Wilf. *Combinatorial Algorithms*. Academic Press, New York, 1st edition edition, 1975.
7. J. Sawada and A. Williams. Greedy flipping of pancakes and burnt pancakes. *submitted manuscript*, 2013.
8. J. Sawada and A. Williams. Greedy pancake flipping. *Electronic Notes in Discrete Mathematics (LAGOS, 2013)*, 44(5):357–362, 2013.
9. S. Singh. Flipping pancakes with mathematics. *The Guardian*, 2013.
10. B. Stevens and A. Williams. The coolest order of binary strings. In *FUN '12: Sixth International Conference on FUN with Algorithms*, volume 7288 of *Lecture Notes in Computer Science*, pages 322–333. 2012.
11. B. Stevens and A. Williams. The coolest way to generate binary strings. *Theory of Computing Systems*, DOI: 10.1007/s00224-013-9486-8:28 pages, 2014.
12. A. Williams. O(1)-time unsorting by prefix-reversals in a boustrophedon linked list. In *FUN '10: Fifth International Conference on FUN with Algorithms*, volume 6099 of *Lecture Notes in Computer Science*, pages 368–379. 2010.
13. A. Williams. The greedy gray code algorithm. In *WADS '13: The Thirteenth Workshop on Algorithms and Data Structures*, volume 6037 of *Lecture Notes in Computer Science*, pages 525–536, London, ON, Canada, 2013.
14. A. Williams. Hamiltonicity of the Cayley digraph on the symmetric group generated by (1 2) and (1 2 $\cdots$ n). *arxiv.org/abs/1307.2549*, page 14 pages, 2013.
15. D. Wong. *Constructions for Universal Cycles (supervised by Joe Sawada)*. PhD thesis in Computer Science, University of Guelph, 2014.
16. S. Zaks. A new algorithm for generation of permutations. *BIT Numerical Mathematics*, 24(2):196–204, 1984.

Stack	flip_i	Rule
1234	2	12
2134	3	213
3124	2	31
1324	3	132
2314	2	23
3214	4	3214
4123	2	41
1423	3	142
2413	2	24
4213	3	421
1243	2	12
2143	4	2143
3412	2	34
4312	3	431
1342	2	13
3142	3	314
4132	2	41
1432	4	1432
2341	2	23
3241	3	324
4231	2	42
2431	3	243
3421	2	34
4321	4	4321

(i) Minimum flips

Stack	flip_i	Rule
1234	4	123_
4321	3	
2341	4	23_1
1432	3	
3412	4	3_12
2143	3	
4123	4	_123
3214	2	4
2314	4	231_
4132	3	
3142	4	31_2
2413	3	
1423	4	1_23
3241	3	
4231	4	_231
1324	2	4
3124	4	312_
4213	3	
1243	4	12_3
3421	3	
2431	4	2_31
1342	3	
4312	4	_312
2134	1	34

(ii) Maximum flips

Stack	$\overline{\mathsf{flip}}_i$	Rule
123	1	1
$\bar{1}23$	2	$\bar{1}2$
$\bar{2}13$	1	$\bar{2}$
213	2	21
$1\bar{2}3$	1	$\bar{1}$
$1\bar{2}3$	2	$1\bar{2}$
$2\bar{1}3$	1	2
$\bar{2}\bar{1}3$	3	$\bar{2}\bar{1}3$
$\bar{3}12$	1	$\bar{3}$
312	2	31
$\bar{1}\bar{3}2$	1	$\bar{1}$
$1\bar{3}2$	2	$1\bar{3}$
$3\bar{1}2$	1	3
$\bar{3}\bar{1}2$	2	$\bar{3}\bar{1}$
132	1	1
$\bar{1}32$	3	$\bar{1}32$
$\bar{2}\bar{3}1$	1	$\bar{2}$
$2\bar{3}1$	2	$2\bar{3}$
$3\bar{2}1$	1	3
$\bar{3}\bar{2}1$	2	$\bar{3}\bar{2}$
231	1	2
$\bar{2}31$	2	$\bar{2}3$
$\bar{3}21$	1	$\bar{3}$
321	3	321
$\bar{1}\bar{2}\bar{3}$	1	$\bar{1}$
$1\bar{2}\bar{3}$	2	$1\bar{2}$
$2\bar{1}\bar{3}$	1	2
$\bar{2}\bar{1}\bar{3}$	2	$\bar{2}\bar{1}$
$12\bar{3}$	1	1
$\bar{1}2\bar{3}$	2	$\bar{1}2$
$2\bar{1}\bar{3}$	1	$\bar{2}$
$21\bar{3}$	3	$21\bar{3}$
$3\bar{1}\bar{2}$	1	3
$\bar{3}\bar{1}\bar{2}$	2	$\bar{3}\bar{1}$
$13\bar{2}$	1	1
$\bar{1}3\bar{2}$	2	$\bar{1}3$
$\bar{3}1\bar{2}$	1	$\bar{3}$
$31\bar{2}$	2	31
$\bar{1}\bar{3}\bar{2}$	1	$\bar{1}$
$1\bar{3}\bar{2}$	3	$1\bar{3}\bar{2}$
$23\bar{1}$	1	2
$\bar{2}3\bar{1}$	2	$\bar{2}3$
$\bar{3}2\bar{1}$	1	$\bar{3}$
$32\bar{1}$	2	32
$\bar{2}\bar{3}\bar{1}$	1	$\bar{2}$
$2\bar{3}\bar{1}$	2	$2\bar{3}$
$3\bar{2}\bar{1}$	1	3
$\bar{3}\bar{2}\bar{1}$	3	$\bar{3}\bar{2}\bar{1}$

(i) Minimum flips

Stack	$\overline{\mathsf{flip}}_i$	Rule
123	3	12 _
$\bar{3}\bar{2}\bar{1}$	2	
$23\bar{1}$	3	2 _ $\bar{1}$
$1\bar{3}\bar{2}$	2	
$3\bar{1}\bar{2}$	3	_ $\bar{1}\bar{2}$
$21\bar{3}$	2	
$\bar{1}\bar{2}\bar{3}$	3	$\bar{1}\bar{2}$ _
321	2	
$\bar{2}\bar{3}1$	3	$\bar{2}$ _ 1
$\bar{1}32$	2	
$\bar{3}12$	3	_ 12
$\bar{2}\bar{1}3$	1	3
$2\bar{1}3$	3	$2\bar{1}$ _
$\bar{3}1\bar{2}$	2	
$\bar{1}3\bar{2}$	3	$\bar{1}$ _ $\bar{2}$
$2\bar{3}1$	2	
$3\bar{2}1$	3	_ $\bar{2}1$
$\bar{1}2\bar{3}$	2	
$\bar{2}1\bar{3}$	3	$\bar{2}1$ _
$3\bar{1}2$	2	
$1\bar{3}2$	3	1 _ 2
$\bar{2}3\bar{1}$	2	
$\bar{3}2\bar{1}$	3	_ $2\bar{1}$
$1\bar{2}3$	1	3
$\bar{1}23$	3	$\bar{1}\bar{2}$ _
$\bar{3}21$	2	
$\bar{2}31$	3	$\bar{2}$ _ 1
$\bar{1}\bar{3}2$	2	
312	3	_ 12
$\bar{2}1\bar{3}$	2	
$12\bar{3}$	3	12 _
$3\bar{2}\bar{1}$	2	
$23\bar{1}$	3	2 _ $\bar{1}$
$13\bar{2}$	2	
$3\bar{1}\bar{2}$	3	_ $\bar{1}\bar{2}$
213	1	3
$\bar{2}13$	3	$\bar{2}1$ _
$\bar{3}1\bar{2}$	2	
132	3	1 _ 2
$\bar{2}\bar{3}\bar{1}$	2	
$32\bar{1}$	3	_ $2\bar{1}$
$1\bar{2}\bar{3}$	2	
$2\bar{1}\bar{3}$	3	$2\bar{1}$ _
$31\bar{2}$	2	
$\bar{1}3\bar{2}$	3	$\bar{1}$ _ $\bar{2}$
231	2	
$\bar{3}21$	3	_ $\bar{2}1$
$\bar{1}23$	0	23

(i) Maximum flips

Table 1. The two orders of burnt pancakes for $n = 3$. Each flip is determined directly using the relevant information in the successor rule.

```c
//-------------------------------------------------------------
// GENERATING (SIGNED) PERMUTATIONS BY MIN or MAX FLIPS
// BY APPLYING SUCCESSOR RULES
//-------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#define MAX_N 20

int n, k, a[MAX_N], sign[MAX_N], total, type, SIGNED = 0;

//-----------------------------------------------------------
void Input() {

	printf(" ----------------------\n");
	printf(" Permutation Generation \n");
	printf(" ----------------------\n");
	printf(" 1. Max Flip \n");
	printf(" 2. Min Flip\n");
	printf(" 3. Max Flip (Signed) \n");
	printf(" 4. Min Flip (Signed) \n");

	printf("\n ENTER selection #: "); scanf("%d", &type);

	if (type < 0 || type > 4) {
		printf("\n INVALID ENTRY\n\n");
		exit(0);
	}

	printf(" ENTER length n: ");
	scanf("%d", &n);

	k = 1;
	if (type == 3 || type == 4) { SIGNED = 1; k = 2; }
	printf("\n");
}
//---------------------------------------
void Print() {
	int i;

	for (i=1; i<=n; i++) {
		if (sign[i] == 0 || !SIGNED) printf(" %d ", a[i]);
		else printf("-%d ", a[i]);
	}
	printf("\n");
	total++;
}
//-----------------------------------------------
void Flip(int t) {
int i, b[MAX_N];

	for (i=1; i<=t; i++) b[i] = a[t-i+1];
	for (i=1; i<=t; i++) a[i] = b[i];

	//============================
	// Flip Signs for signed case
	//============================
	if (k > 1) {
		for (i=1; i<=t; i++) b[i] = sign[t-i+1];
		for (i=1; i<=t; i++) sign[i] = (b[i]+1) % k;
	}
}
//---------------------------------------
void MinFlip() {
int incr,j;

	do {
```

```
67              Print();
68              incr=0;
69              j=1;
70              while (j < n) {
71                      if (a[j] < a[j+1]) incr++;
72                      if (incr == 2 || (incr == 1 && a[j+1] < a[1])) break;
73                      j++;
74              }
75              Flip(j);
76      } while (!(j == n && a[1] == 1));
77  }
78  //---------------------------------------
79  void SignedMinFlip() {
80  int incr,j;
81      do {
82              Print();
83              incr=0;
84              j=1;
85              while (j < n) {
86                      if (a[j] < a[j+1]) incr++;
87                      if (incr == 2 || (incr == 1 && a[j+1] < a[1])) break;
88                      if (a[j] < a[j+1] && sign[j] == sign[j+1]) break;
89                      if (a[j] > a[j+1] && sign[j] != sign[j+1]) break;
90                      j++;
91              }
92              Flip(j);
93      } while (!(j == n && a[1] == 1 && sign[1] == 0));
94  }
95  //---------------------------------------
96  void MaxFlip() {
97  int j;
98      do {
99              Print(); Flip(n); Print();
100             j = n;
101             while (a[j] == j) j--;
102             Flip(j-1);
103     } while (j > 2);
104 }
105 //---------------------------------------
106 void SignedMaxFlip() {
107 int j;
108     do {
109             Print(); Flip(n); Print();
110             j = n;
111             while (a[j] == j && sign[j] == 0) j--;
112             Flip(j-1);
113     } while (j > 1);
114 }
115 //---------------------------------------
116 int main() {
117     int j;
118
119     Input();
120     //==============
121     // INITIAL PERM
122     //==============
123     for (j=1; j<=n; j++) a[j] = j;
124     for (j=1; j<=n; j++) sign[j] = 0;
125
126     if (type == 1) MaxFlip();
127     if (type == 2) MinFlip();
128     if (type == 3) SignedMaxFlip();
129     if (type == 4) SignedMinFlip();
130
131     printf("Total = %d\n\n", total);
132 }
```