A Study of Permutations Permissible by LIFO Service Disciplines *

Simon Hawkin Ashok Agrawala

Institute for Advanced Computer Studies
and Department of Computer Science
University of Maryland
College Park, MD 20742
{cema,agrawala}@cs.umd.edu

November 11, 1998

Abstract

We study permutations of the job order performed by various LIFO service disciplines. The sets of such permutations are shown to be equivalent to sets of string permutations with simple characteristics. In particular, it is easy to test whether a given permutation belongs to these sets. Several algorithms that efficiently perform such tests are presented.

*This work is supported in part by DARPA/Army ARO under contract DABT 6396C0075 and Mississippi State University under contract 96144601 to the Computer Science Department at the University of Maryland. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or MSU.

1 Introduction

Measuring the network dynamics [1, 2] has been an ongoing effort in the Department of Computer Science in the University of Maryland, and similar projects have been under way in many other research laboratories[3, 4]. In the study of the Internet, as other complex systems, internal details often cannot be examined directly. Instead, their characteristics have to be deduced based on the (external) observation of their behavior. This paper presents a technique that makes a step in that direction.

The traffic in a packet-switched network like the Internet is shaped by the servers on the path between the end-points of every connection. Each server takes time to process a packet, and therefore needs a buffer to store packets that await their turn. This buffer is naturally considered a queue of packets. While the first-in-first-out (FIFO) queue organization is common, sometimes the order of packets is not maintained. The simplest service discipline that may change the order of serviced jobs is the last-in-first-out (LIFO) discipline that employs a stack organization of the queue. A study of various LIFO service disciplines can therefore give us insight into the lower bounds on the complexity of the observed server.

2 Setup and Basic Definitions

We consider a server with a LIFO (**l**ast-**i**n-**f**irst-**o**ut) discipline, or simply, a LIFO server. It contains one service point and a queue which is organized as a stack. One job may be processed at a time. When a job arrives, but the server is busy with another job, the contention is resolved by sending one of the conflicting jobs to the queue. If the job that was executing remains active, the server is *non-preemptive*. Otherwise, the server is *preemptive*. After a preempted job comes back from the queue to the service point, it either resumes execution from the last point it was active or starts it all over again. We call the former case *partial preemption* and the latter, *complete preemption*. We also assume that the service discipline does not change during the life time of the server. Finally, the server can be either *work-conserving* or *non-work-conserving*. In the former case, whenever there is a job available for execution (either arriving from the arrival stream or the job on top of the stack) and the service point is available, the appropriate job starts service. In the latter case, the server may sit idle even when there are jobs available for execution.

The arrival stream naturally defines a total order on the set of job numbers. For convenience of presentation, we will use the job number as its "name", that is, refer to a job with arrival number X simply as job X. Similarly, we sometimes call the job with the maximal index a maximal job, and so on.

Note that two jobs can arrive simultaneously. If jobs A and $B > A$ arrive at the same time, the job B is considered arriving later than A, consistent with the job enumeration. When the arrival times are important, it is more precise to say that the job B *becomes available* after job A. Also, observe that no two jobs have the same departure time, since no two jobs can be serviced simultaneously.

An outside observer can monitor the arrival and departure streams of jobs. Each job i has the following parameters: arrival time a_i, service time s_i, departure time d_i. When the exact times are not known, the arrival and departure streams become *strings*, or sequences. We can study both finite strings $s_1 \dots s_M$, produced by finite "bunches" of jobs, as well as infinite strings $s_1 \dots$. In both finite and infinite cases, a contiguous piece of a string $s_i s_{i+1} \dots s_{i+k}$ is called a *substring*. A non-contiguous sequence of elements that preserves their original order $s_{i_1}, s_{i_2}, \dots$ $(i_1 < i_2 < \dots)$ is called a *subsequence*. At every valid position k, a string[1] $s = \{s_i\}_{i=1}^{M}$ is split in the *tail* and *trunk*. The former is the substring of s that starts at position k, that is, $\{s_i\}_{i=k}^{M}$; the latter is the substring of s that ends at position k, that is, $\{s_i\}_{i=1}^{k}$. As usual, the first element s_1 of a string is called a *head*; the rest $\{s_i\}_{i=2}^{M}$ is simply the tail of the string. Similarly, the part $\{s_i\}_{i=1}^{M-1}$ of the string that does not include only the last element may be called the trunk of the string.

A LIFO server turns an arrival string into a departure string, effectively performing a permutation. We will call such permutation a LIFO permutation, and a departure string after a LIFO server, a LIFO string. For example, BAC is a LIFO permutation of ABC, when the server is preemptive, job B arrives before job A has completed its execution, and job C arrives after both A and B departed. Not all permutations are LIFO; for example, the departure string ZXY cannot be produced from the arrival string XYZ by a LIFO server with any service discipline. The proof will be presented in theorem 1. This turns out to be rather general: whenever an arrival string has a subsequence XYZ turned ZXY in the departure string, the server is not LIFO. We call such subsequence a ZXY (sub)sequence, and the string that has it as a subsequence, a ZXY string[2]. The main result of this paper is that the reverse is also true for preemptive servers: that is, the string is LIFO if and only if it is non-ZXY. With a slight modification, this result holds for non-preemptive servers as well.

The paper presents the main result which ties LIFO strings to non-ZXY strings and thus provides a simple way to deduce whether the server under observation can be a LIFO server. This is followed by several algorithms that efficiently perform such tests, for both finite and infinite streams of jobs.

[1] Here we assume $M = \infty$ if the string is infinite.

[2] Pronounced [*ZEE-ksi*].

3 LIFO And Non-ZXY Strings: Same Thing (Almost)

The main question we are looking to answer is: which permutations can occur at a LIFO server? Notice that the exact arrival, service and departure times are assumed unknown. Thus, the question can be reformulated: we want to know which permutations can be performed by a LIFO server for some values of $\{a_i\}$, $\{s_i\}$ and $\{d_i\}$, $1 \leq i \leq M$. This is similar to problem 5 in section 2.2.1 of volume I in [5] which however considers only the queue (stack) without a service point, and therefore does not apply to various service disciplines we study here.

In this section, we show the equivalence between LIFO permutations and a class of non-ZXY permutations.

3.1 LIFO Strings Are Non-ZXY

This subsection will prove one part of the equivalence: that a ZXY string cannot be a LIFO string. The proof evolves through the following lemmas.

Lemma 1 If at some point of execution job X is below job Y in stack, then whenever X and Y are both in stack, X is below Y.

Proof: The order does not change until Y is on top of stack. After that, either some other jobs arrive and new jobs are pushed on stack (above Y), leaving the order intact; or the server becomes available and Y leaves the stack to start or continue its execution. The following can happen after this. *Case 1*. Y completes execution and departs. Therefore, it is not in stack. *Case 2*. Another job arrives and preempts Y. Y is pushed on top of stack and is therefore still above X. *Case 3*. Another job arrives and does not preempt Y. Therefore, Y is not in stack.

In all these cases, whenever X and Y are together in stack, X remains below Y.
Q.E.D.

Lemma 2 If job X is pushed in stack while job Y executes, job Y departs before job X departs.

Proof: Suppose X is pushed on top of stack. The 3 cases mentioned in the proof of the previous lemma can happen after that. In case 1, Y departs before X departs. In case 2, Y is pushed in stack above X. By the previous lemma, it remains above X while in stack. X can only leave stack when it is on top; at that moment, Y is not in stack (or it would be below X) and is not executing (or X would not leave stack). Therefore, when X leaves stack, Y has completed its execution and departed. Therefore, X departs after X. Finally, case 3 does not change the positions of jobs X and Y. **Q.E.D.**

Lemma 3 If job X is in stack when job Y arrives, job Y departs before job X departs.

Proof: If job Y is pushed in stack when it arrives, we have the conditions of lemma 2. If job Y executes, it can either be pushed in stack by conflicting arriving jobs, or complete execution and depart. In the first case, we again have the conditions of lemma 2; in the second case, job Y departs before job X departs.
Q.E.D.

Now we can prove this important

Theorem 1 A ZXY permutation cannot be a $LIFO$ permutation.

Proof: Suppose a ZXY permutation was created by a LIFO server.

Case 1: a preemptive server. Since each job X and Y arrived before but departed after Z, both X and Y were in stack when Z executed. The moment Y arrived, either X was in stack and departed after Y by lemma 3 or X was executing and preempted by Y and departed after Y by lemma 2. Neither creates a ZXY permutation.

Case 2: a non-preemptive server. The job Y completed execution after job X. At the moment of Y's arrival, X was not in stack, since otherwise Y would have departed before X. Therefore, Y arrived after X started (and maybe completed) execution. Since the moment Z arrived[3] was after Y arrived, X has already started its execution by this time. If it had completed, it would have departed before Z. Otherwise, Z would be pushed in stack and depart after X by lemma 2. Neither case creates a ZXY permutation.
Q.E.D.

The result of theorem 1 holds for both preemptive and non-preemptive LIFO servers.

As an aside note, suppose the server chooses arbitrarily between the preemptive and non-preemptive discipline. The following scenario will produce a ZXY permutation. Job X arrives and starts execution. Job Y arrives and is pushed in stack. Job Z arrives and starts execution, preempting X. Z departs; X is popped off stack. X departs; Y is popped off stack. Y departs. This leads us to an

Observation 1 The non-LIFO property of a ZXY string is not only a function of the queue organization, but also of the consistency of the service discipline.

3.2 Non-ZXY Strings Are LIFO: Preemptive Servers

The other part of the equivalence between LIFO strings and a class of non-ZXY strings will be proved separately for different service disciplines. This subsection concentrates on preemptive LIFO servers. The results are modified for non-preemptive disciplines in the next subsection.

[3] More precisely: became available.

Lemma 4 Consider a bunch of M jobs whose execution by a LIFO server produced a departure string $s = c_1 \ldots c_H M b_1 \ldots b_F$. Delete job M from the bunch and execute by the same server. This will produce a departure string $c_1 \ldots c_H b_1 \ldots b_F$, that is, just like s with M deleted.

Proof: Every job c has departed before M arrived, so the trunk of the departure string does not change. Every job b has arrived before M has arrived and was in stack while M was executing. The following cases are possible.

The job J that was executing when M arrived was pushed on stack. M started execution and continued without interruption, then departed. After this, J resumed execution. The stack remained unchanged as compared with the moment M arrived. Therefore, the order of execution and departure of the jobs in the tail was maintained. The complete departure string is thus $c_1 \ldots c_H b_1 \ldots b_F$.
Q.E.D.

Corollary 1 Deleting job M from a LIFO string of length M leaves a LIFO string of length $M - 1$.

Lemma 4 and the corollary tell us how to proceed to a shorter LIFO string. The following lemmas show how we can build a longer string.

Lemma 5 Consider a LIFO string s of M jobs with job M at position k. Add job $M + 1$ to the bunch. In the new departure string s', job $M + 1$ may only occur at a position $i \geq k$.

Proof: If job $M + 1$ occurs at a position $i \leq M$ in s' and job M at a position $p > i$, then $p = i + 1$, or else the jobs at positions i, $i + 1$, and p would form a ZXY subsequence. Suppose $i < k$. By pigeonholing, there is a job J at position $j < k$ in s with position $j' > i + 1$ in s'.

Execution of jobs $1, \ldots M$ was not affected by job $M + 1$ until it arrived. The following cases are possible.

Case 1. If J had departed before $M + 1$ arrived, according to s, it would have departed before job $M + 1$ arrived (and departed) in s' as well, which did not happen.

Case 2. If J was executing and preempted when $M + 1$ arrived, it would be pushed on stack above job M (which was in stack because it did not depart before $M + 1$). Then J would have departed before M in s', which did not happen.

Case 3. Suppose J was in stack when $M + 1$ arrived. According to s', M had not departed by that time, but either was in stack below J or was executing. Had $M + 1$ not arrived, job M would still have departed before J, which did not happen, according to s.
Q.E.D.

Lemma 6 Under the assumptions of lemma 5, job $M + 1$ can occur at *any* position $i \geq k$.

Proof: by inductive construction. Consider a bunch of length M. Cases $M = 1, 2$ trivially form the base of induction. Now, suppose the lemma has been proved for bunches of length up to M. Taking one such bunch (with job M at position k) and any $i > k$, we will construct a bunch of length $M + 1$ such that job $M + 1$ will be *inserted* at position i.

Construction. Consider the tail of the given bunch starting at k: $M b_1 \dots b_F$, where b_j is the departure position of some job f_j. We will also define $b_0 = M$. Of course, $0 \leq j \leq F$ and $0 \leq F < M$. To construct a bunch, we should provide all a_i and s_i, for all i. In this construction, we set $s_{M+1} = 1$ and keep a_i and s_i, $0 \leq i \leq M$, intact. We will also denote $\bar{s}_i$ the remaining part of execution of job i at a particular moment[4]. If job i has not completed execution, the value of $\bar{s}_i$ is positive whether the preemptive discipline is partial or complete.

To insert job $M + 1$ at position k in the new string, let $a_{M+1} = a_M + \sigma \bar{s}_M$ and set σ to any value between 0 and 1. Indeed, no job arrives between a_M and a_{M+1}, and at the moment a_{M+1} job M has not completed and is preempted by $M + 1$. Job $M + 1$ executes without preemption (no other job arrives) and leaves the server. After this, M resumes execution, and the stack is the same as it was before $M + 1$ arrived. No other jobs arrive. Therefore, only jobs from the stack are selected for execution, and their order of execution is defined by their order in stack. Thus, the tail of the new string is the same as the tail of the given string. Note that the trunk of the string is trivially not affected either. We inserted job $M+1$ in the given string at position k.

To set job $M + 1$ at position $k + 1$, let $a_{M+1} = a_M + \bar{s}_M + \sigma \bar{s}_{b_1}$ and set σ to any value between 0 and 1. In general, to set job $M + 1$ at position $k + i$ $(0 < i \leq M - k)$ let $a_{M+1} = a_M + \bar{s}_{b_0} + \ldots + \bar{s}_{b_{i-1}} + \sigma \bar{s}_{b_i}$, and set σ to any value between 0 and 1.. Indeed, job $M + 1$ starts after b_{i-1} has completed, but before b_i starts. Note that no new jobs arrive after job $M + 1$ has arrived, so only jobs from the stack $(b_i, \dots b_F)$ are selected for execution. Since their relative positions in the stack do not change, neither does their order of execution, so they depart in the original order. Thus, we inserted job $M + 1$ in the given string at position $k + i (1 \leq i \leq M - k)$.

Finally, to set job $M + 1$ at position $M + 1$, let $a_{M+1} = a_M + \sum_{i=0}^{F} \bar{s}_i + \sigma$, and set σ to any positive value. Job $M+1$ starts after all other jobs have completed, and therefore does not affect their execution. This completes the construction.
Q.E.D.

Now, we can prove the important

Lemma 7 An arbitrary non-ZXY permutation can be produced by a preemptive LIFO server.

Proof: by induction. The lemma is obviously true for non-ZXY strings of length 1 and 2. Assume the lemma proved for strings of length up to M. Consider a non-ZXY string s of length $M + 1$ and a string s' which is string s with job $M + 1$ deleted. String s' is of length M and is a LIFO by inductive assumption. String s can be produced from s' by inserting $M + 1$ and is therefore a LIFO string by lemma 6.

[4]That is, if job i was preempted and resumes execution at this moment, it will take $\bar{s}_i$ time to complete.

Q.E.D.

The main result for preemptive LIFO servers follows.

Theorem 2 A departure string is LIFO if and only if it is non-ZXY.

Proof: Follows directly from theorem 1 and lemma 7. **Q.E.D.**

3.3 Non-ZXY Strings That Are LIFO: Non-Preemptive Servers

The previous subsection established an equivalence relation between the non-ZXY strings and valid departure strings of LIFO servers with preemptive disciplines. This subsection studies non-preemptive LIFO servers. While a ZXY string is not LIFO in case of non-preemptive as well as preemptive LIFO servers (theorem 1), not all non-ZXY strings are valid LIFO strings in the non-preemptive case. However, an equivalence will be shown between a large class of non-ZXY strings and LIFO strings.

More concretely, consider permutations that maintain the first element. So, the head of the arrival string is also the head of the departure string. Call such departure string *headed*. We will show the equivalence between headed non-ZXY strings and departure strings of non-preemptive LIFO servers.

One part of the equivalence is easy, as shows the following

Lemma 8 A departure string from a non-preemptive LIFO server is a headed non-ZXY string.

Proof: The first job to become available for execution starts execution and is never preempted. Therefore, it is the first job to complete execution and depart. Thus, the departure string is headed.

Also, from theorem 1, the departure string is non-ZXY.

Q.E.D.

Our proof of the other part of the equivalence will follow the scheme of the proof for preemptive LIFO servers.

First, we will prove lemma 4 for the non-preemptive case.

Lemma 9 Consider a bunch of M jobs whose execution by a LIFO server produced a departure string $s = c_1 \ldots c_H M b_1 \ldots b_F$. Delete job M from the bunch and execute by the same server. This will produce a departure string $c_1 \ldots c_H b_1 \ldots b_F$, that is, just like s with M deleted.

Proof: When job M arrived, it was pushed on top of stack and did not affect the job J that was executing at the time. After J departed, M started (and completed) its execution, then departed. While M was executing, the stack was the same as before it arrived. Therefore, the

order of execution and departure of the jobs in the tail was maintained. The complete departure string is thus $c_1 \dots c_H b_1 \dots b_F$.
Q.E.D.

Corollary 1 also holds.
The following is an analogue of lemma 5.

Lemma 10 Consider a LIFO string s of M jobs with job M at position k. Add job $M+1$ to the bunch. In the new departure string s', job $M+1$ may only occur at a position $i \geq k$.

Proof: Suppose job $M+1$ is inserted at position $i < k$. Then, job M is at position $p \leq i+1$. Execution of jobs $1 \dots M$ was not affected by job $M+1$ until it arrived. By pigeonholing, there is a job J at position $j < k$ in s and position $j' > i+1$ in s'. Job J was on top of stack when job $M+1$ arrived; another job E was executing. Job E departed before J in s, so $E \neq M$. Since M has not departed before J in s, it was in stack below J. The stack did not change after job $M+1$ arrived, executed, and departed. Therefore, job M was still below J and would depart after it. This contradicts s'.
Q.E.D.

The next construction is similar to that in lemma 6.

Lemma 11 Under the assumptions of lemma 10, job $M+1$ can occur at *any* position $i \geq k$.

Proof: Cases $M = 1, 2$ trivially serve as a base of induction. Assume the lemma has been proved for strings of length up to M. We will take a bunch with departure string s of length M, insert job $M+1$ at a given position $i \geq k$, and show that it is still a valid LIFO string.

Consider the tail of s: $Mb_1 \dots b_F$, $0 \leq F < M$. We will call job M also b_0. Then, job b_i is at position $k+i$, $0 \leq i \leq M$.

As in the construction for lemma 6, the new string retains the arrival, service, and departure times of all jobs in the old string. It will manipulate the value of a_{M+1} as well as a_M. The values $\bar{s}_i$ are defined as in lemma 6.

Case 1: $i = k$. Notice that $k > 1$, since the string is headed. Let J be the job at position $k-1$. If $a_M < d_j$, there was a period of time before J departed when J was executing and M was the only job in stack. If $a_m \geq d_j$, there was a period of time starting at some τ when J was executing and the stack was empty. In the latter case, set a_M to $\tau + \sigma(d_j - a_M)$ for any σ between 0 and 1, to force the former case. Notice that this does not change the departure string. Now, set a_{M+1} to $a_M + \sigma(d_j - a_M)$ for any σ between 0 and 1. This makes $M+1$ arrive while M is on top of stack. After J departs, job $M+1$ executes and departs, then M and the rest of jobs in stack. Because all $f_j, j \geq 1$ were in stack when job M arrived, all $f_j, j \geq 0$ are in stack while job $M+1$ executes. Thus, the order of execution and departure of these jobs does not change.

Case 2: $i = k+j$, $i < M+1$. For j = 1, set $a_{M+1} = a_M + \sigma \bar{s}_M$, for any σ between 0 and 1. In general, set $a_{M+1} = a_M + \bar{s}_M + \dots + \bar{s}_{f_{j-2}} + \sigma \bar{s}_{f_{j-1}}$. This forces job $M+1$ to arrive after job

f_{j-1} started execution and job f_j was on top of stack. It will therefore execute and depart after job f_{j-1} and before job f_j

Case 3: $i = M + 1$. To append job $M + 1$, simply set $a_{M+1} = d_{b_F} + \sigma$, for any positive value of σ.

Q.E.D.

Finally, an analogue of lemma 7 is proved literally in the same way:

Lemma 12 An arbitrary headed non-ZXY permutation can be produced by a non-preemptive LIFO server.

Proof: The lemma is obviously true for headed non-ZXY strings of length 1 and 2. Assume the lemma proved for strings of length up to M. Consider a headed non-ZXY string s of length $M + 1$ and a string s' which is string s with job $M + 1$ deleted. String s' is of length M and is a LIFO by inductive assumption. String s can be produced from s' by inserting $M + 1$ and is therefore a LIFO string by lemma 11.

Q.E.D.

The main result for LIFO servers with partial preemption follows directly from theorem 8 and lemma 12:

Theorem 3 A departure string is LIFO if and only if it is a headed non-ZXY string.

3.4 Non-Work-Conserving Servers

So far, the text tacitly assumed work-conserving servers. This section shows how to extend the results to non-work-conserving servers.

A non-work-conserving server may have idle time even when there are jobs available. The construction in lemmas 6 and 11 may introduce such idle periods. Suppose we have set a_{M+1} to $a_M + \sum_{i=0}^{i-1} \bar{s}_{b_j} + \sigma \bar{s}_{b_i}$, and there is an idle period of length τ. Recall that s_M and therefore $\bar{s}_M$ had an arbitrary value. Increase it by τ. This will ensure that the server will be available at the moment job $M + 1$ is available. The order of jobs in stack is not affected. Notice that no other job (besides $M + 1$) will dictate the value of s_M. Thus, we get rid of idle periods, and therefore execution of a non-work-conserving LIFO server is exactly the same as that of a work-conserving LIFO server with an otherwise identical service discipline.

Observation 2 Using a non-work-conserving server has the same effect as changing job service times with a work-conserving server, as far as the order of execution is concerned.

Therefore, we only consider work-conserving service disciplines in the rest of the text.

4 LIFO Detection Algorithms

This section presents efficient algorithms that actually test whether a given permutation could be produced by a LIFO server. For conciseness, we consider non-ZXY strings, that is, test the preemptive service discipline. For a non-preemptive server, one can easily test whether the first job to arrive is also the first job to depart. This test could be done in constant time once the head of the departure string has been observed.

4.1 Finite Streams

We first study finite strings. The results will be used in the next subsection which presents algorithms for infinite streams.

Lemma 13 The tail of a LIFO string of length M starting at M is a decreasing substring.

Proof: Suppose the tail is not decreasing. So, it has at least two jobs, A and B, in the original (increasing) order. Then, jobs M, A and B form a ZXY subsequence of the tail and the whole string, which contradicts the assumption that the string is LIFO.
Q.E.D.

The following algorithm tests whether a given string is a LIFO string.

Algorithm 1 ("Peeling"). Consider a departure string of a bunch of size M. If $M \leq 2$, stop successfully. If the tail after M is not decreasing, stop with failure. Otherwise, delete job M from the string and apply peeling recursively.

Note: if $M > 2$, the above algorithm may not succeed in less than $M - 1$ recursions.

Lemma 14 If the departure string is ZXY, peeling fails.

Proof: Consider a string with a ZXY subsequence $Z\ X\ Y$. If peeling stopped before iteration $M-Z$, it failed. Suppose it did not stop. At iteration $M-Z$, Z is the job with the largest index, so the tail starting at Z is considered. Since it contains $X < Y$ before Y, it is not decreasing, and peeling fails.
Q.E.D.

Lemma 15 If peeling fails, then the departure string is ZXY.

Proof: Consider the recursion at which peeling fails. Set Z to the maximal job index. Since the tail is not decreasing, it contains jobs X and $Y > X$ in that order. Z, X, Y form a ZXY subsequence of the departure string.
Q.E.D.

The following theorem is applicable to preemptive LIFO servers.

Theorem 4 Peeling succeeds if and only if the string is LIFO.

Proof: Follows directly from lemmas 14 and 15. **Q.E.D.**

As has been noted above, peeling is easily extended to non-preemptive servers by prepending the constant-time *test of heads* of the arrival and departure strings.

Theorem 5 The complexity of peeling is upper-bounded by the sorting complexity.

Proof: We will show that, putting the sorting aside, peeling of a string of length M can be done in $O(M)$ time and $O(1)$ space in addition to the string storage. Sorting is only required to find the position of the current maximal job.

Suppose we know that the tail starting at position $t = t_M$ (with job J_t) is decreasing. Let $p = p_M$ be the position of job M. If $p \geq t$, the peeling test will succeed, so it can immediately proceed to the recursion step. If $p < t$, the tail s_M starting at M is a concatenation of two substrings: $s_p = J_p \dots J_{t-1}$ and the tail s_t starting at position t. It is easy to see that s_M decreases if and only if both s_p and s_t decrease and $J_{t-1} > J_t$. Testing that s_p is decreasing takes $O(t-p)$ time; testing that $J_{t-1} > J_t$ takes constant time; hence, testing that s_M is decreasing takes $O(t-p)$ time. Assuming the test is successful, we set $t = t_{M-1}$ to p_M and proceed to the recursive step.

At the start of peeling, $t_M = M$. At iteration i, the cost is $O(t_i - p_i)$. The total peeling cost is $O\left(\sum_{i=2}^{M} C(t_i - p_i)\right)$ for a sufficiently large C. The sum reduces to

$$C(t_M - p_M) + \sum_{i=2}^{M-1} C(p_{i+1} - p_i)$$

$$= Ct_M - Cp_2 \leq C(t_M - 1) < CM$$

Q.E.D.

4.2 Infinite Streams

Consider an infinite (to the right) stream of jobs, observed from the start. Can the observed departure string be produced from the arrival string by a LIFO server? To answer this, we will split the stream of jobs into disjoint finite contiguous bunches in such manner that ensures jobs from different bunches would be executed independently by a LIFO server. Independent execution means that, while one job is executed, the other does not wait for its completion (*i.e.* is not in stack). In our algorithms, we monitor the departure stream and keep track of stack properties. Whenever we detect the stack becoming empty, we split the stream: one bunch is completed, another starts getting accumulated.

In general, there is no algorithm that would always detect whether an infinite departure string could be produced by a LIFO server. For example, if job X is kept in the queue longer that the observation period, it is not possible to deduce whether a ZXY sequence appears in the sequel. However, whenever a bunch is completed, such detection is possible for the appropriate finite substring. Also, sometimes a negative answer (that the server is not LIFO) is possible in the middle of the bunch processing.

Consider the following algorithm that splits a (possibly infinite) string into bunches by imitating a LIFO server.

Algorithm 2 Assume the server is LIFO. Keep track of the *maximal encountered* job M in the departure stream (initially set to the value of the first index in the departure stream) and the stack size s (initially set to the same value less one). Consider the current job d in the departure stream. The following cases are possible.

Case 1. If $d < M$, the job came from the stack, so we decrease s. If now $s = 0$, we split the stream.

Case 2. If $d > M$, job d came with another pack of $d - M$ new jobs, indexed $M + 1, \ldots d$. The last job d is executed, while all other jobs are pushed in stack. Thus, stack size s is increased by $d - M - 1$. We also update M to d.

The time complexity of the algorithm 2 is obviously constant per job and linear in the number of jobs observed. Note that only a constant amount of space is required.

If the server is not LIFO, can we detect that using this algorithm? One way is to detect a stack underflow ($s < 0$). Another, after a bunch has been split off, test it with peeling. (For non-preemptive LIFO servers, prepend it with the test of heads.) However, if the stack never becomes empty, a non-LIFO string may be missed; that is, it may not be detected in any pre-determined finite time.

The following algorithm performs on-the-fly LIFO detection by imitating the (logical) execution of a LIFO server based on the observed arrival and departure strings.

Algorithm 3 ("On-the-fly LIFO detection"). Keep the stack (initially empty), with its top element and size available with constant-time queries. Maintain also indices i and j of the arrival and departure streams, respectively (initially both set to 1). Consider a_i and d_j. The following cases are possible.

Case 1. If $a_i = d_j$, the job a_i was executed as it arrived. It did not change the stack. Accordingly, increment i and j and proceed.

Case 2. If $a_i > d_j$, the job d_j would be taken from the top of stack by a LIFO server. If the current top is not d_j, the server is not LIFO. Otherwise, pop d_j from the stack and increment j. Now, if stack becomes empty, a bunch can be split off the departure string.

Case 3. If $a_i < d_j$, this means job a_i was preempted, so we push a_i on stack and increment i.

The algorithm will also work in the finite case of the arrival and departure strings of length M with the following slight modification:

Modification 1 Pretend that all arriving jobs after M are equal to ∞ and dropped in the departure string.

Explanation: Until the end of one of the strings is reached, the execution is independent of the string length. If the end of only the departure (but not the arrival) string is reached, this means some jobs have been dropped, and the server is not LIFO. If the end of both strings is reached simultaneously, this means that case 1 holds, and the server is LIFO if and only if the stack is empty. Finally, consider the case when the end of the arrival string is reached but not of the departure string. If the strings were infinite, and the server is LIFO, the next arrived job would be greater than all previous encountered jobs, and case 2 would hold. It would flush the stack together with the departure string (as long as they are consistent). Our modification of the algorithm forces this to happen in the finite case as well.

Observation 3 Algorithms 2 and 3 split the same bunches.

Indeed, bunches are defined under the assumption of a LIFO server, which both algorithms imitate.

Theorem 6 Consider a bunch split off the departure stream by algorithm 3. The corresponding departure string could be produced by a LIFO server from the arrival string if and only if the algorithm did not detect it was non-LIFO.

Proof: By construction, as long as the algorithm executes successfully, the corresponding trunks of the arrival and departure strings correspond to a valid execution of a LIFO server. When a bunch is started, the stack is empty; thus, the jobs from the bunch do not interfere with previous job of the arrival stream. Similarly, because the stack is empty when the bunch is completed, all jobs from the bunch have been processed by this point.
Q.E.D.

5 Open Questions

This paper classified permutations that can be performed by a LIFO server with various service disciplines. The setup can be modified in several ways.

1. More advanced strategies can be considered. One natural extension is adding priorities to the jobs, thereby splitting the stack into several stacks. How will this affect the possible permutations?

2. Suppose the values of arrival times a_i, service times s_i, or both, are known. How will this abet the analysis?
3. The most interesting extension of this work would be an analysis of several LIFO servers working in team. Consider, for example, a sequence of several LIFO servers. What permutations of length M can they produce? What is the minimal number of LIFO servers required to produce an arbitrary permutation of length M? Such a team would be able to imitate any service discipline, for a given arrival string. Because LIFO is the simplest service discipline that changes the order of jobs, this would provide an insight into the lower bound of complexity required of any server to perform this permutation.

6 Conclusion

This paper studied permutations of job orders performed by various LIFO service disciplines. The set of such permutations was shown to be equivalent to a set of string permutations with simple characteristics. In particular, it is easy to test whether a given permutation belongs to the set. Finally, the paper presented several algorithms that efficiently perform such test on finite and infinite streams.

References

[1] D. Sanghi, O. Gudmundsson, A. Agrawala. *Study of Network Dynamics*. Computer Networks and ISDN Systems, v.26, no.3, 1993, pp 371–378.

[2] J. Pointek, F. Shull, R. Tesoriero, A. Agrawala. *NetDyn Revisited: A Replicated Study of Network Dynamics*. Computer Networks and ISDN Systems, v.29, no.7, 1997, pp 831–840.

[3] V. Paxson, J. Mahdavi, A. Adams, M. Mathis. *An Architecture for Large-Scale Internet Measurement*. IEEE Communications, v.36, no.8, August 1998, pp 48-54.

[4] *The Internet Performance and Analysis Project*. `http://www.merit.edu/ipma/docs/team.html`

[5] D. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA. Second ed., 1973.