THE COMBINATORICS OF ABSTRACT CONTAINER DATA TYPES

Dominic H. Tulley

The Combinatorics of Abstract Container Data Types

A thesis submitted to the

UNIVERSITY OF ST ANDREWS

for the degree of

DOCTOR OF PHILOSOPHY

By

Dominic H Tulley

School of Mathematical and Computational Sciences

University of St Andrews

August 1996

Th C142

“eleven plus two

=

twelve plus one”

I, Dominic H Tulley, hereby certify that this thesis, which is approximately 30,000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

date 30 August 1996 *signature of candidate*

I was admitted as a research student in October 1993 and as a candidate for the degree of Doctor of Philosophy in October 1994; the higher study for which this is a record was carried out in the University of St Andrews between 1993 and 1996.

date 30 August 1996 *signature of candidate*

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date 30 August 1996 *signature of supervisor*

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

date 30 August 1996 *signature of candidate*

Abstract

The study of abstract machines such as Turing machines, push down automata and finite state machines has played an important role in the advancement of computer science. It has led to developments in the theory of general purpose computers, compilers and string manipulation as well as many other areas. The language associated with an abstract machine characterises an important aspect of the behaviour of that machine. It is therefore the principal object of interest when studying such a machine.

In this thesis we consider abstract container data types to be abstract machines. We define the concept of a language associated with an abstract container data type and investigate this in the same spirit as for other abstract machines. We also consider a model which allows us to describe various abstract container data types. This model is studied in a similar manner.

There is a rich selection of problems to investigate. For instance, the data items which the abstract container data types operate on can take many forms. The input stream could consist of distinct data items, say $1, 2, \ldots, n$, or it could be a word over the binary alphabet. Alternatively it could be a sequence formed from the data items in some arbitrary multiset. Another consideration is whether or not an abstract data type has a finite storage capacity.

It is shown how to construct a regular grammar which generates (an encoded form of) the set of permutations which can be realised by moving tokens through a network. A one to one correspondence is given between ordered forests of bounded height and members of the language associated with a bounded capacity priority queue operating on binary data. A number of related results are also proved; in particular for networks operating on binary data, and priority queues of capacity 2.

Acknowledgements

I am greatly indebted to my supervisor, Professor Mike Atkinson. Without his support, encouragement, insight and insistence on deadlines throughout the last three years this thesis may well never have been written.

I also owe many thanks to Duncan Shand. Not the least of which is for managing to share an office with me for three years. Without the many useful discussions, plentiful advice and coffee this thesis would certainly not be as it is.

Contents

Notation

σ	The input sequence to a data type
τ	The output sequence from a data type
n	The length of the input sequence
N	A general transportation graph
B_k	A buffer of capacity k
S_k	A stack of capacity k
P_k	A priority queue of capacity k
k	The storage capacity of the data type in question
$L(D)$	The language associated with the data type D
$L_n(D)$	The elements of length n in $L(D)$
$S(\tau)$	The set of sequences σ such that (σ, τ) is allowable
$T(\sigma)$	The set of sequences τ such that (σ, τ) is allowable
$S_k(\tau)$	The set of sequences σ such that (σ, τ) is k-allowable
$T_k(\sigma)$	The set of sequences τ such that (σ, τ) is k-allowable
$s(\tau)$	$\lvert S(\tau)\rvert$
$t(\sigma)$	$\lvert T(\sigma)\rvert$
$s_k(\tau)$	$\lvert S_k(\tau)\rvert$
$t_k(\sigma)$	$\lvert T_k(\sigma)\rvert$
α^r	The reverse of a sequence α
$\bar{\alpha}$	The complement of a sequence α

1 Introduction

Abstract machines such as Turing machines, push-down automata, and finite-state machines are well known and well understood in computer science. They have been used to study general purpose computers, compilers, and string manipulation although their importance goes well beyond these three applications. The central theoretical issue for these and other machines is the characterisation of the languages associated with them. The aim of this thesis is to study abstract container data types in the same spirit.

Abstract data types are central to the point of view adopted in object-oriented programming which now permeates all large software projects. Although abstract data typing was initially adopted primarily for its use as a software design tool it has always been recognised that each data type has a rigorous mathematical definition.

Each abstract data type is characterised by the set of operations that can be performed on it. Therefore an abstract data type can be regarded as an abstract machine whose instruction set is the set of operations it supports. Some of these operations may supply input or output while others may examine or change the state of the abstract data type. The precise specification of these instruction sets has been studied in considerable depth by algebraic means, see [EMCO92a, EMCO92b] for a survey. However, the classical abstract machines are studied at a much deeper level; their behaviour in response to *arbitrary* sequences of instructions, or programs, is studied and this behaviour is captured by the idea of the language recognised by the machine. As yet, such a study has hardly begun for abstract data types although, as indicated below, there is a very natural extension of the language notion to abstract data types.

Our aim is to provide a framework to unify several important early ad hoc results of Knuth, Pratt, Tarjan, Even and Itai [Knu73a, Pra73, Tar72, EI71]. Knuth and Pratt both investigated single abstract container data types (stacks and deques) and successfully described the associated language (albeit using a different terminology). Tarjan, Even and Itai generalised this investigation to allow networks of stacks and queues but only found partial results. Our unifying framework places these results in a generalised context: that of associating a language with an abstract data type.

There are an infinite number of data types and it seems to be infeasible to give a general theory of their associated languages which has deep implications for all of them. However, in practice, only a small number of abstract data types, such as stacks, queues, arrays and dictionaries, recur frequently in software and algorithm design. Therefore it is perhaps more profitable to study only those which have demonstrable software utility.

This thesis concentrates on abstract *container* data types: those for which Insert and Delete operations are defined. Such abstract data types act as data transformers, outputting their input data in a permuted order. If, except for house-keeping operations, Insert and Delete are the only operations supported by the data type then the functional behaviour of the abstract data type is essentially defined by the possible ways in which it can permute the data. A sequence of Insert and Delete operations constitutes a program for the abstract data type when it is regarded as a machine.

For such an abstract data type it is not desirable to define its associated language to be the set of input sequences that lead to an accepting state, since that discards so much essential information about the output. Instead, we propose that the associated language should be defined to be the set of (input, output) pairs of sequences that can arise from the execution of an abstract data type program. As we shall see, there are a number of questions about the language associated with an abstract data type whose formulation and solution require combinatorial machinery. These questions form the basis of the work presented here.

1.1 Definitions

Let A be any abstract container data type. We shall consider only programs for A which begin and end with A in the empty state: this represents the normal way that a container data type would be used. Such a program, formed only from Insert and Delete operations, must satisfy two other conditions. The first is that every initial segment must contain at least as many Inserts as Deletes to ensure that a Delete operations is never executed when A is empty. The second is that the Insert and

Delete operations in the program should be equinumerous to ensure that the final state of A is the empty state. Let σ be any sequence of length n and let P be any program with n Inserts and n Deletes satisfying the above conditions. The execution of P with σ as the input sequence results in the members of σ being inserted into A in order of their occurrence in σ. It also results in the Delete operations generating a sequence τ which we call the output of P. A pair (σ, τ) which is related by a program P in this way is called *allowable*, or A-allowable when the abstract data type cannot be deduced from context. As an example consider a stack operating on the input sequence $1, 2, 3$. If it executes the program "Insert Insert Delete Insert Delete Delete" then the output sequence will be $2, 3, 1$, and thus $((1, 2, 3), (2, 3, 1)$ is a Stack-allowable pair. Direct enumeration of all valid programs for input length 3 quickly verifies that $((1, 2, 3), (3, 1, 2))$ is not an allowable pair. The concept of allowability and related notions are central to the work in this thesis.

One such related notion is the set of allowable pairs which is denoted by $L(A)$ and is called the language associated with the data type A. If two data types have the same language over some alphabet then we say they are *permutationally equivalent*. In the study of these languages it is often necessary to partition them by the size of the pairs. In this context the phraseology is a little counter intuitive; an allowable pair of length n refers to an allowable pair (σ, τ) with $Length(\sigma) = Length(\tau) = n$. Basic combinatorial questions about $L(A)$ include:

1. How many A-allowable pairs of length n are there?
2. Is there a characterisation of the A-allowable pairs that enables them to be recognised quickly?
3. Is there an efficient algorithm that, given an input sequence σ, can determine how many A-allowable pairs (σ, τ) there are? And, dually,
4. Is there an efficient algorithm that, given an output sequence τ, can determine how many A-allowable pairs (σ, τ) there are?

It is often the case that an allowable pair can be produced by more than one program. A queue computing the allowable pair (σ, σ) could execute the program

$(Insert\ \ Delete)^n$ or it could execute the program $Insert^n\ Delete^n$. These are but two of the $c_n = \binom{2n}{n}/(n+1)$ programs which compute the pair. A stack on the other hand can only compute each allowable pair in one way. When considering the behaviour of these data types it is sometimes necessary to specify which of the many possible programs is being executed. To this end we informally introduce a *standard computation.* The idea of a standard computation is that, when attempting to permute σ into τ, the data type never executes an Insert operation when it can produce another element of the output using a Delete. This corresponds to the data type storing as few data items as possible at all times. It is not possible to give a formal definition of a standard computation which covers all the data types since the definition is dependent on the data type. Where necessary we shall formalise the definition.

In practice abstract container data types tend to have a bounded size, either enforced by their implementation or the physical limits of the hardware. It therefore makes sense to consider the above questions when no more than k data items can be stored at any time in the abstract data type. We correspondingly introduce the idea of *k-allowability* by defining the language $L(A_k)$ of a *k-bounded* abstract data type A_k. We let $L(A_k)$ denote the set of allowable pairs (σ, τ) for which there is a program P which can transform σ into τ without requiring more than k data items to be stored at any one time.

There are further variations we can introduce to the questions already raised. We have not yet stipulated what form the input sequence takes; it could be taken as a sequence of distinct elements, as a word over the binary alphabet, or as a reordering of an arbitrary multiset. If it consists of distinct data items we can assume without loss of generality that they are $1, 2, \ldots, n$. Similarly, in the multiset case, we can assume without loss of generality that the multiset is $S = \{1^{a_1}, 2^{a_2}, \ldots, r^{a_r}\}$.

It is clear that if (σ, τ) and (α, β) are k-allowable pairs then $(\sigma\alpha, \tau\beta)$ is k-allowable. In this case, when the data type is executing a program to transform the input into the output, it is possible that it will become empty when τ has been produced. The converse of this is that if a data type becomes empty part way through a program then the allowable pair it is computing can be reduced into two distinct and shorter allowable pairs. If it is not possible for the data type to become empty when transforming σ into τ then we say (σ, τ) is *irreducible* otherwise we say it is *reducible.*

As noted previously there are some data structures whose behaviour is independent of the value of the data items being processed. In general, if the data structure is oblivious to the data value and all the data items are distinct, it is not necessary to consider allowable pairs. In this case, if $(\sigma, \pi(\sigma))$ is an allowable pair for some permutation π and some sequence σ, then $(\sigma, \pi(\sigma))$ is allowable for all $n!$ possible sequences σ. The language of a data structure is therefore characterised entirely by the set of *allowable permutations* π. In the cases where this is possible we shall use the notation $L(A)$ to refer to the set of allowable permutations rather than pairs. Similarly we shall use $L(A_k)$ in the bounded capacity case.

There are various methods for characterising allowable permutations. One of the more successful techniques is that of *pattern avoidance* as used by Pratt ([Pra73]) and Knuth ([Knu73a, 2.2.1; Q 5]). There are other applications of pattern avoidance beyond allowability. In [KSW96] it is used to investigate (r, s) permutations. These are the permutations which can be partitioned into r increasing and s decreasing, possibly empty, subsequences. In [BBL93] a characterisation of the separable, or $(1, 1)$, permutations is given in terms of pattern avoidance. In [SS85] and [Rot75] some work is done on counting the permutations avoiding certain patterns or sets of patterns. A pattern of length m is a permutation $\rho = \rho_1, \rho_2, \ldots, \rho_m$ of $1, 2, \ldots, m$. A sequence $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ is said to *contain* the pattern ρ if there is a subsequence σ' of σ such that $|\sigma'| = m$ and $\rho_i < \rho_j$ if and only if $\sigma'_i < \sigma'_j$. As an example, the sequence $5, 4, 2, 3, 1$ contains the pattern $3, 1, 2$ because it contains the subsequence $5, 2, 3$, but on the other hand $5, 4, 3, 2, 1$ does not contain the pattern $1, 2, 3$. If ρ does not occur within σ we shall say that σ *avoids* ρ.

It is shown in [BBL93] that the general pattern matching problem for permutation patterns is NP-complete. All is not lost however, for it is also shown there that special cases can have efficient solutions. In particular, as stated above, it is shown that separable permutations, those which avoid 3142 and 2413, can be recognised and counted in polynomial time.

1.2 Thesis Overview

The work presented in this thesis falls into three categories. The first of these covers transportation networks, the second covers the oblivious abstract container data types and the third is non-oblivious abstract container data types. This reflects the structure of the thesis, with a chapter devoted to each.

In Chapter 2 we introduce *transportation networks.* These are graph theoretic models representing the movement of data from a source node to a destination node through a network of connected internal nodes. We show that the set of allowable permutations for any fixed network is a regular set. We then give a method for finding a grammar which generates encodings of the elements of that set. In the binary case we show that any transportation network is permutationally equivalent to a single buffer of some fixed capacity. A corollary of this is that bounded capacity unrestricted deques, input restricted deques, output restricted deques and stacks are all equivalent to buffers of some size in the binary case.

Oblivious abstract container data types are investigated in Chapter 3. Here we consider buffers, deques and stacks. The allowable permutations of a bounded buffer are characterised and counted. We also consider the binary input case and derive a recurrence for the number of allowable pairs. For deques we apply the techniques of Chapter 2 to the bounded capacity case for several fixed capacities. This gives rise to some empirical data which behaves as we would expect. It also agrees with existing results pertaining to the unbounded case. In the previous chapter stacks are shown to be equivalent to buffers in the binary case. Here we apply the techniques from Chapter 2 to a stack with permutation inputs and find the asymptotic behaviour of the number of allowable pairs for stacks of capacity $1, 2, \ldots, 5$. Finally, we point out a result of [dBKR72] which applies here.

Chapter 4 covers the non-oblivious container data types; priority queues and double ended priority queues. One of the original motivating problems for this thesis was the analysis of a bounded capacity priority queue operating on permutation inputs. In this chapter we present a recurrence and generating function for the number of allowable pairs for a priority queue of capacity 2 on permutation inputs. We also have some success is analysing a priority queue of capacity 2 over multiset inputs. We give a one to one correspondence between the multiset allowable pairs and trees of a certain form. From this we then derive a recurrence for the number of these trees satisfying certain conditions. Section 4.2 breaks slightly from the standard model. We investigate the case where the priority queue has a permutation input sequence but the priority of these data items is independent of the value. The priorities are assigned separately and chosen from the binary alphabet. Here we present the analysis of several special cases, a conjecture for the general result and some numerical evidence to support it.

The next section of Chapter 4 investigates priority queues with binary inputs. We characterise the allowable pairs and go on to give a one to one correspondence between k-allowable binary pairs and ordered forests of height at most $k+1$. A closed form expression for the number of such forests is given in [dBKR72]. We then consider the composition of priority queues before deriving algorithms to solve questions 3 and 4 posed in Section 1.1 above. The final section of this chapter presents a small number of results relating to double ended priority queues.

Finally, in Chapter 5, we present our conclusions and outline some areas which warrant further investigation.

Some of the work in this thesis appears in [ATar, AT94, ALTar] for which my supervisor was a coauthor. His contribution was no more than is normally expected in a supervisory capacity.

2 Transportation Networks

Many problems in computer science involving the transfer of data items between various locations can be modelled using directed graphs. The nodes of the graph represent the locations where data can be stored and the edges represent the possibility that data might be transferred from one location to another. Examples include:

1. Distributed Computing Networks. The nodes are computers, the edges are high-speed data communication channels, and the data items are files or fragments of files.

2. Parallel Computers. The nodes are processing elements, the edges are the data highways represented by the architecture (hypercube, mesh, butterfly etc.), and the data items are the bits, bytes and words which flow along the highways.

3. Models of Transportation Systems. The nodes are cities, the edges are roads, and the data items are commodities being shipped from one city to another.

4. Data Structures. We shall say more about this example later but for now it will be enough to consider binary search trees. The nodes are the memory cells containing a key and associated values. The edges connect each node to its left and right children, and the data items are the (key, value) pairs which move around the tree as it is updated.

This chapter addresses some features common to all these situations. In the next section we pose an abstract problem concerned with the movement of distinct *tokens* around the nodes of a graph and the ways in which they may be permuted. Then, in Section 2.2, we solve the problem in terms of regular sets, the theory for which can be found in [HU79]. An example of the application of this solution is then given in Section 2.3. Finally a more restricted case, where the tokens are chosen from the binary alphabet, is considered and a necessary and sufficient condition for a pair of binary sequences to be allowable is found. Later, in Sections 3.2 and 3.3, the theory is applied to extend the work of [Knu73a], [Tar72] and [Pra73].

2.1 Problem Formulation

Let N be a finite directed graph with a designated input node and a designated output node. The input node has no incoming edges and the output node has no outgoing edges. The remaining nodes are called internal nodes. To avoid trivialities we also assume that every internal node is on at least one path from the input node to the output node. Such a graph will be called a transportation network.

Each internal node is allowed to contain 0 or 1 tokens. In applications of this problem the tokens are items of data but in our abstract formulation we shall assume only that tokens are denoted by non negative, not necessarily distinct, integers.

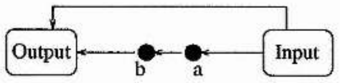
The input node generates an ordered sequence $\sigma = \sigma_1, \sigma_2, \sigma_3, \ldots$ of tokens. How this is done is unimportant at present; the tokens could be generated by local computation at the input node or may have been received from an external source. The tokens are then moved from one node to another along the directed edges of the network until they arrive in some order at the output node. Tokens cannot be stored on the edges of the network. A token σ_i on a node x can only move to a node y if the following conditions hold:

1. there is an edge from x to y.
2. x is the input node and tokens $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$ have already been moved from the input node or x is an internal node.
3. y is an internal node and y does not currently contain any token, or y is the output node.

Condition 1 implies that tokens move from the input node to the output node along a path in the network, condition 2 implies that tokens leave the input node in the order specified by the input sequence $\sigma_1, \sigma_2, \sigma_3, \ldots$ and condition 3 ensures that every internal node contains at most one token at any time.

Example 2.1 Figure 1 shows a simple network with 2 internal nodes and one possible movement of 3 tokens through it. The tokens arrive at the output node in the order 3, 1, 2. It is not difficult to see that different choices of moves could result in any arrival order other than 3, 2, 1. The general technique developed in this chapter allows us to show, for example, that if x_n is the number of possible output orders of n distinct tokens for the network in Figure 1 then

$$x_0 = x_1 = 1,$$
$$x_n = 3x_{n-1} - x_{n-2}, \text{ for } n \geq 2.$$



Time Step	output	b	a	input
0				123
1			1	23
2		1		23
3		1	2	3
4	3	1	2	
5	31		2	
6	31	2		
7	312			

Figure 1: Action of a simple network

If the input stream is a fixed, finite sequence and the tokens all arrive at the output node then the output node will contain some reordering τ of the input stream. We say that τ is generated by the transportation network from σ and that (σ, τ) is N-*allowable* or just allowable if N can be derived from the context. As noted before, if we are using permutation inputs, we need only consider allowable permutations. In this case we say τ is N-allowable or simply τ is allowable.

In the following sections we consider two separate problems derived from this model. In Section 2.2 we consider permutation inputs and investigate the allowable permutations. In this case the set of all permutations which can be generated by a transportation network N will be denoted by $L(N)$ and the set of all permutations of $1, 2, \ldots, n$ in $L(N)$ will be denoted by $L_n(N)$. The main result in this section is a description of the set $L(N)$ which allows the membership problem (given a sequence

π determine whether $\pi \in L(N)$) to be answered in time proportional to the length of π. It also permits $|L_n(N)|$ to be calculated. Section 2.3 contains an example of the use of this result.

The second problem is studied in Section 2.4. Here we allow the tokens in the input sequence to be chosen from the binary alphabet. This means the input sequence cannot be fixed as an arbitrary set of distinct tokens and so we must consider allowable pairs (σ, τ) of binary sequences. We show that any transportation network is permutationally equivalent to a single buffer of some specific capacity. It is also shown that this equivalence applies to stacks, deques, input restricted deques and output restricted deques.

2.2 A Network with Permutation Inputs

When the inputs are distinct we think of them simply as the integers and when there are n of them we consider them to be $1, 2, \ldots, n$. In this and the following section we shall take this as the input sequence and investigate the allowable permutations. We show how a grammar may be defined which generates (an encoded form of) the permutations in $L(N)$. In the next section we work through an extended example, showing how the technique can be applied.

Our technique depends on an encoding of the permutations in $L(N)$. For every permutation $\tau = \tau_1\tau_2 \ldots \tau_n$ we define its encoded form $\chi(\tau) = c_1c_2 \ldots c_n$ by defining c_i to be the rank of τ_i in $\tau_i\tau_{i+1} \ldots \tau_n$. For example the encoded form of 2761453 is 2651221. It is easy to see that every code sequence $c_1c_2 \ldots c_n$ which encodes a permutation satisfies $c_{n-i} \leq i + 1$ and that every sequence of positive integers of length n which fulfils this condition is the encoding of some permutation. We let $C(N)$ and $C_n(N)$ denote the set of strings which are the encoded forms of the elements of $L(N)$ and $L_n(N)$ respectively. Our main theorem in this section is

Theorem 2.1 *$C(N)$ is a regular set.*

The proof of Theorem 2.1 is constructive. We show how a regular grammar can be defined which generates $C(N)$. We then use some standard constructions for regular languages to obtain a method for computing $|C_n(N)| = |L_n(N)|$.

We label the internal nodes of N in an arbitrary order as $1, 2, \ldots, k$. As a sequence $1, 2, \ldots, n$ of tokens is transferred, step by step, through the transportation network to the output node the network passes through a series of configurations. A configuration is a disposition of tokens among the internal nodes. Such a disposition may be described by an *occupancy function* $f : \{1, 2, \ldots, k\} \to \{0, 1, 2, \ldots, n\}$; $f(i)$ is the token residing on node i except in the case $f(i) = 0$ which represents that node i is empty. The configuration where $f(i) = 0$ for all i is denoted by ϵ; it is both the initial and the final configuration of the transportation network.

Two configurations with occupancy functions f, g are defined to be equivalent if

- $f(i) = 0$ if and only if $g(i) = 0$
- $f(i) < f(j)$ if and only if $g(i) < g(j)$

A state of the transportation network is defined to be an equivalence class of configurations. The state corresponding to the equivalence class $\{\epsilon\}$ is denoted by q_0.

Lemma 2.1 *The number of states of a transportation network N is finite.*

Proof: Every configuration is equivalent to one in which the tokens on the internal nodes are $1, 2, \ldots, h$ with $h \leq k$. The occupancy function then takes values in $\{0, 1, 2, \ldots k\}$ and there are at most $(k+1)^k$ such functions. □

Let C denote the set of configurations, and E the set of edges of N. We shall define a partial function $\nu : C \times E \to C \times \{1, 2, \ldots, k, \lambda\}$. Let $c \in C$ and $e = (i, j) \in E$. Informally, $\nu(c, e)$ specifies the configuration that arises from configuration c if a token is transferred from node i to node j, and the new symbol (if any) that is placed at the output node. More precisely, $\nu(c, e)$ is defined whenever, in configuration c, node i contains a token t (if node i is the input node then t is the next input token to be transferred from the input node) and either node j is empty or node j is the output node. Under these conditions $\nu(c, e)$ is defined as (d, x) where

- d is the configuration that would arise from configuration c if t was transferred from i to j, and
- x is the empty symbol λ unless j is the output node in which case x is the rank of t among all the tokens residing on internal nodes in configuration c.

Lemma 2.2 *A permutation τ is in $L(N)$ if and only if there is a sequence of edges $e_1, e_2, \ldots, e_p$ and configurations $\epsilon = a_0, a_1, \ldots, a_p = \epsilon$ such that $\nu(a_{i-1}, e_i) = (a_i, x_i)$ for $i = 1, 2, \ldots, p$, and $x_1 x_2 \ldots x_p = \chi(\tau)$.*

Proof: Suppose $\tau_1\tau_2\ldots\tau_n = \tau \in L(N)$. Then there is a sequence of transfers along edges $e_1, e_2, \ldots, e_p$ which causes tokens $1, 2, \ldots, n$ on the input node to move through the transportation network until they have arrived on the output node in the order $\tau_1, \tau_2, \ldots, \tau_n$. Let $\epsilon = a_0, a_1, \ldots, a_p = \epsilon$ be the sequence of configurations induced in the transportation network by these transfers and let t_i be the token taking part in the i^{th} transfer. Then, certainly, $\nu(a_{i-1}, e_i) = (a_i, x_i)$ where

- if t_i is not being transferred to the output, $x_i = \lambda$ and
- if t_i is being transferred to the output, x_i is the rank of t_i among all the tokens on internal nodes in configuration a_{i-1}; and since the tokens remaining in the input node are greater than all these tokens x_i is actually the rank of t_i among all tokens not yet transferred to the output

Thus $x_1x_2\ldots x_p$ has the property that the j^{th} (non-empty) symbol is the rank of τ_j among $\tau_j, \tau_{j+1}, \ldots, \tau_n$; in other words, $x_1x_2\ldots x_p = \chi(\tau)$.

For the converse suppose, for some permutation $\tau = \tau_1\tau_2\ldots\tau_n$, there exist edges $e_1, e_2, \ldots, e_p$ and configurations $\epsilon = a_0, a_1, \ldots, a_p = \epsilon$ such that $\nu(a_{i-1}, e_i) = (a_i, x_i)$ for $i = 1, 2, \ldots, p$, and $x_1x_2\ldots x_p = \chi(\tau)$. Then, by definition of ν, it is possible to move tokens $1, 2, \ldots, n$ from the input node in p transfers using edges $e_1, e_2, \ldots, e_p$ in turn and passing through configurations $a_0, a_1, \ldots, a_p$. At a step which transfers a token t to the output (corresponding to the equation $\nu(a_{i-1}, e_i) = (a_i, x_i)$ say) the token has rank x_i among all tokens not yet placed in the output. Hence the permutation π defined by the sequence of transfers satisfies $\chi(\pi) = x_1x_2\ldots x_p = \chi(\tau)$

and therefore $\tau = \pi \in L(N)$ as required. □

Lemma 2.3 *Let a, b be equivalent configurations and let e be an edge of N. Suppose that $\nu(a,e)$ is defined and equal to (c,x). Then $\nu(b,e)$ is defined and equal to (d,y) where d is equivalent to c and $y = x$.*

Proof: Let e be the edge (r,s) and suppose r is not the input node. Let f be the occupancy function of a.

Suppose that s is not the output node. Then $x = \lambda, f(r) = t \neq 0, f(s) = 0$ and the occupancy function f' of c differs from f only in that $f'(r) = 0, f'(s) = t$. From the equivalence of a and b the occupancy function g of b satisfies $g(r) = u \neq 0, g(s) = 0$. It follows that $\nu(b,e)$ is defined and has occupancy function g' differing from g only in that $g'(r) = 0, g'(s) = u$. Therefore, for all i, j, we have $f'(i) = 0$ if and only if $g'(i) = 0$ and $f'(i) < f'(j)$ if and only if $g'(i) < g'(j)$. Thus c and d are equivalent and $y = \lambda$.

If s is the output node the equivalence of c and d follows in the same way. From the equivalence of a and b the token that is being transferred from node r to the output has the same rank among the tokens of the internal nodes of a as it does among those of b; that is, $x = y$.

When r is the input node the proof is similar. The new data item taken from the input node is necessarily greater than any other data item stored in the network. The equivalence of c and d again follows from the above argument, taking this into account. In the case that s is the output node we have $x = y$ since the data item has the greatest value amongst all data items in the network. □

Let Q be the set of states. We define a partial function $\mu : Q \times E \to Q \times \{1, 2, \ldots, k, \lambda\}$. Let $q \in Q$ and $e \in E$. Thus q is an equivalence class of configurations. Let a be any configuration in this equivalence class. If $\nu(a,e)$ is defined and equal to (d,x) then we define $\mu(q,e)$ to be (r,x) where r is the equivalence class of configurations that contains d. Lemma 2.3 guarantees that μ is well-defined (independently of the choice of a in the equivalence class q). Lemma 2.4 follows immediately from these definitions.

Lemma 2.4 *Let $e_1, e_2, \ldots, e_p$ be any sequence of edges. Then there exists a sequence of configurations $\epsilon = a_0, a_1, \ldots, a_p = \epsilon$ such that $\nu(a_{i-1}, e_i) = (a_i, x_i)$ for $i = 1, 2, \ldots, p$ if and only if there exists a sequence of states $q_0, q_1, \ldots, q_p = q_0$ such that $\mu(q_{i-1}, e_i) = (q_i, x_i)$ for $i = 1, 2, \ldots, p$.* □

Lemmas 2.2 and 2.4 have the following consequence:

Lemma 2.5 *A permutation τ is in $L(N)$ if and only if there is a sequence of edges $e_1, e_2, \ldots, e_p$ and states $q_0, q_1, \ldots, q_p = q_0$ such that $\mu(q_{i-1}, e_i) = (q_i, x_i)$ for $i = 1, 2, \ldots, p$, and $x_1 x_2 \ldots x_p = \chi(\tau)$.* □

Proof: (of Theorem 2.1) We define a context-free grammar (N, T, P, S) as follows:

- The set N of non-terminals is (indexed by) the set of states
- The set T of terminal symbols is $\{1, 2, \ldots, k, \lambda\}$
- The set of productions P contains productions of the form $q \to xr$ for every defined value $\mu(q, e) = (r, x)$ of the function μ together with a production $q_0 \to \lambda$
- The goal symbol S corresponds to the state q_0

Derivations in this grammar have the form

$$q_0 \to x_1 q_1 \to x_1 x_2 q_2 \to \ldots \to x_1 x_2 \ldots x_p q_p \to x_1 x_2 \ldots x_p$$

Such a derivation exists if and only if there is a sequence of edges $e_1, e_2, \ldots, e_p$ for which $\mu(q_{i-1}, e_i) = (q_i, x_i), i = 1, 2, \ldots, p$. Therefore, by Lemma 2.5, the strings $x_1 x_2 \ldots x_p$ generated by this grammar are precisely those strings of the form $\chi(\tau)$ for $\tau \in L(N)$, that is, they are the strings of $C(N)$.

The grammar (N, T, P, S) is not itself regular since among its productions $q \to xr$ there may be some of the form $q \to r$, when $x = \lambda$. Standard results in language theory (see [HU79],p26, p218) show that there is, nevertheless, a regular grammar which defines the same language; thus $C(N)$, being generated by a regular grammar, is a regular set. $\square$

We turn now to the question of computing the numbers $|L_n(N)|$ for all values of n. The method for calculating these numbers is a special case of a more general treatment valid for any context-free grammar [CS63]. However, we reproduce that part of this general theory here for convenience and to establish the basis for the case-studies in Sections 2.3, 3.2 and 3.3. There is a well-known equivalence ([HU79], p218) between regular grammars and finite state automata. In this equivalence transitions from a state p to a state q occasioned by a symbol x correspond either to productions $p \rightarrow xq$ (if q is not a final state) or productions $p \rightarrow x$ (if q is a final state). There is also a well-known construction ([HU79],p22) that produces an equivalent deterministic finite state automaton from a non-deterministic one. The regular grammar that is associated with a deterministic finite state automaton has the property that the productions on each fixed non-terminal have distinct initial terminals beginning their right-hand sides. The consequence of this is that $C(N)$ can be generated by a regular grammar with non-terminal symbols $S = S_1, S_2, \ldots, S_h$ and productions of the form

$$S_i \rightarrow t_1U_1|t_2U_2|\ldots|t_wU_w$$

where $t_1, t_2, \ldots, t_w$ are distinct together with a production $S \rightarrow \lambda$.

Let $s_n^{(i)}$ be the number of terminal strings of length n that can be derived from S_i. Note that $s_0^{(i)} = 1$ if $i = 1$ (since $S \rightarrow \lambda$) and $s_0^{(i)} = 0$ otherwise. Let z_{ij} be the number of productions of the form $S_i \rightarrow tS_j$ (i.e. there are z_{ij} terminals t that can appear in this form of production). Since the derivations that begin with the production $S_i \rightarrow tS_j$ produce a set of terminal strings that is disjoint from the set of terminal strings that are derived beginning from any other production on S_i (because none of those terminal strings will begin with t) we have the recurrence

$$s_n^{(i)} = \sum_{j=1}^{h} z_{ij}s_{n-1}^{(j)}$$

This gives us a set of h linear recurrences which, together with the h boundary conditions, completely define each $s_n^{(i)}$. In particular $s_n^{(1)} = |L_n(N)|$ can be found in this way.

There are standard techniques for solving sets of recurrence equations ([Rob91] Section 8.2). These methods tell us that the general solution depends on the eigenvalues of the matrix $[z_{ij}]$. The most straightforward case is when the eigenvalues $\mu_1, \mu_2, \ldots, \mu_h$ are distinct and in that case $s_n^{(i)}$ has the form $\sum_{j=1}^{h} k_{ij}\mu_j^n$ where the constants k_{ij} are determined by the initial values. When there are repeated eigenvalues there is a similar expression for $s_n^{(i)}$ in which the k_{ij} are replaced by polynomials in n of degrees depending on the multiplicities of the eigenvalues.

The outcome of this is that $|L_n(N)|$ is given by a formula of the form

$$p_1(n)\mu_1^n + p_2(n)\mu_2^n + p_3(n)\mu_3^n + \ldots$$

where $\mu_1, \mu_2, \ldots$ are certain distinct constants and $p_1(n), p_2(n), \ldots$ are certain polynomials all of which can be calculated. We shall choose the notation so that $|\mu_1| \geq |\mu_2| \geq \ldots$ and then $|L_n(N)|$ is asymptotic to $p(n)|\mu_1|^n$ as $n \to \infty$ (where $p(n)$ is the sum of those polynomials $p_i(n)$ for which $|\mu_i| = |\mu_1|$). We can therefore conclude the following.

Theorem 2.2 *There exists a constant $|\mu_1|$ depending only on N such that*

$$\frac{1}{n}\log(|L_n(N)|) \to \log|\mu_1| \text{ as } n \to \infty$$

Corollary 2.6

$$\frac{|L_n(N)|}{|L_{n-1}(N)|} \to |\mu_1| \text{ as } n \to \infty$$

There is an information-theoretic interpretation of this result. We might consider $|L_n(N)|$ as a measure of how much choice exists in moving n tokens from the input node to the output node. The larger $|L_n(N)|$ is the more possible outcomes there are from inserting $1, 2, \ldots, n$ into the network. Therefore the uncertainty of what the result will be is directly related to $|L_n(N)|$. In information theory uncertainty is measured in bits and we can define the entropy of the arrival permutation as $\log_2|L_n(N)|$ which is asymptotic to $n\log_2|\mu_1|$. To put this in another way the entropy per output token is $\log_2|\mu_1|$. By analogy with the results of [ALTar] we therefore define the entropy of the entire transportation network to be $\log_2 |\mu_1|$. The entropy is a rough measure of how much permutational capability is possessed by a transportation network.

It should be noted that the calculation of $|L_n(N)|$ has linear time complexity in n. However this understates the difficulty of carrying out the calculation since there is, potentially, a doubly exponential dependence on the size of N. This arises from a combination of two steps employed in the derivation of the grammar. Firstly, the number of states the network can reach can be exponential in the number of nodes in the network. Also it is well known that the conversion to a regular grammar, which is the next step in the derivation, generally results in an exponential increase in the number of non-terminal symbols. In Sections 2.3 and 3.2 we shall see that this combinatorial explosion can sometimes be contained. Section 3.3 presents some empirical results which arise from another case where the explosion can be contained.

2.3 Example Analysis

As an illustration of the above technique we analyse the binary tree network shown in Figure 2. If we were to carry out the method in full we would have to consider several hundred different states but common sense allows us to cut this down to 12 states grouped in various ways to give 7 non-terminal symbols. For instance, from the empty state there are 5 different states which can be reached by inserting a data item into the network. According to our definition of equivalence these 5 states are all different but it is clear that any one of them can be reached from any of the others without consuming any input data items or producing any output. We therefore allow ourselves to consider them equivalent. If we associated non-terminals symbols B, C, D, E & F with these 5 states and used A to represent the empty state then we would have a production rather like $A \rightarrow \ldots |2B|2C|2D|2E|2F| \ldots$ in the grammar. Due to their similarity the non-terminals B, C, D, E and F would all have identical productions and so we can reduce the initial productions to $A \rightarrow \ldots |2B| \ldots$. The algorithm for converting the initial grammar into a regular grammar later on in the process is usually described in terms of finite state machines. One of the characteristic effects of this algorithm is the generation of *super-states* such as that described above. Intuition allows us to foresee their creation in some situations and thus we can make our work easier by using them from the beginning.

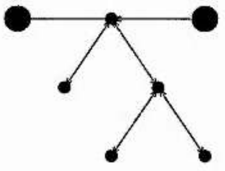


Figure 2: An example network

There is only one state in which the network contains no data items and we shall associate the empty network with the non-terminal symbol A. The 5 possible states we mentioned earlier, which contain one data item, we shall associate with the non-terminal symbol B. The same holds for a network containing 2 data items, any state is reachable from any other state without producing any output so we shall associate all these states with the non-terminal symbol C. There are three essentially distinct states containing 3 data items and these are shown in Figure 3. We shall associate these with the non-terminal D. This grouping is not for the same reasons as the previous ones however, if we keep them separate, the NFA to DFA construction would create a state consisting of the union of all three so we may as well do it here. There is no intuition involved in this step we have merely grouped together all the possible distributions of the three tokens over the network. Similarly there are 6 essentially different states containing 4 data items, shown in Figure 4, and we associate these with the non-terminal E. Having set up some of the required super-states we can begin to think what the grammar looks like. We begin with transitions from the empty state, clearly if a data item with rank 1 is output then the network will remain empty so $A \to 1A$. If a data item of rank 2 is output then a data item of rank 1 must be inserted into the network first and will remain there after the 2 has been output. This single data item can be on any of the five nodes in

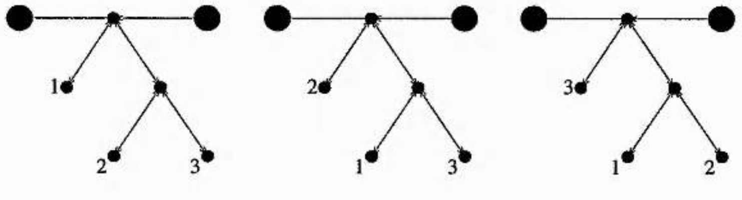


Figure 3: States associated with symbol D

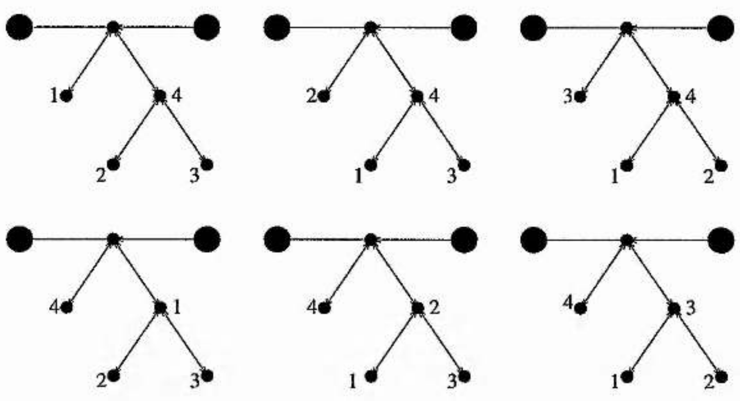


Figure 4: States associated with symbol E

the network and so we have $A \rightarrow 2B$. Similarly if any data item of rank 3 is output there will be two data items left in the network and they can be distributed in any way on the network so $A \rightarrow 3C$. If the data item output is of rank 4 then the three remaining data items can be distributed in three different ways as shown in Figure 3 so, since all three are reachable we can use the super-state non-terminal symbol D instead of three separate non-terminal symbols. This gives $A \rightarrow 4D$ When a data item of rank 5 is output from the empty state there are only 6 ways in which the four remaining data items can be distributed over the nodes of the network, these are the ways shown in Figure 4. This final production gives us an overall production from A which is $A \rightarrow 1A|2B|3C|4D|5E|\lambda$.

In a similar fashion we get productions for B, C and D and the grammar so far is

$$\begin{aligned} A &\rightarrow 1A|2B|3C|4D|5E|\lambda \\ B &\rightarrow 1A|2B|3C|4D|5E \\ C &\rightarrow 1B|2B|3C|4D|5E \\ D &\rightarrow 1C|2C|3C|4D|5E \end{aligned}$$

Things become a little untidy when we consider the super-state corresponding to E. Consider what happens when we output a data item of rank 1 from any of the states in Figure 4. It is immediately obvious that there are only two which allow the 1 to be output, those on the left of the figure. Both of these result in the same state, which is shown in Figure 5 and shall correspond to the non-terminal symbol

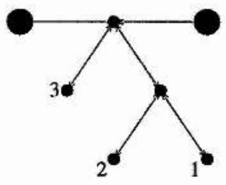


Figure 5: State F

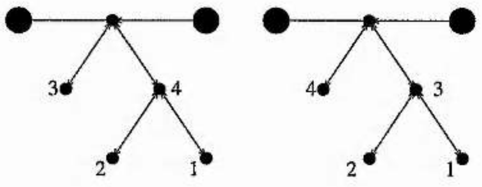


Figure 6: State G

F. Although this state is one of the ones contained in the super-state D it would not be correct to use D because not *all* the states in D are reachable from E. Similarly, when a data item of rank 2 is output the result is state F and when a data item of rank 3 is output the result is state F. If a data item of rank 4 is output the result is any state in Figure 3 so $E \to 4D$ and if a data item of rank 5 is output then network will not change so we have $E \to 1F|2F|3F|4D|5E$.

From state F we see that if any of $1, 2$ or 3 is output the network returns to a state where it contains only two data items and thus any distribution of them is possible so it is in state C. If a data item of rank 4 is output the remaining data items are unchanged so it remains in state F. Finally, if a data item of rank 5 is output the network must enter one of the states shown in Figure 6 and we shall associate this super-state with the non-terminal symbol G. Therefore we have the production $F \to 1C|2C|3C|4F|5G$

Finally we consider the states reachable from G; No data item of rank 1 or 2 can be output from either of the states in G, if 3 or 4 is output then the result is state F and if a data item of rank 5 is output the result is state G so we have the final grammar for this network.

$$\begin{aligned}
A &\to 1A|2B|3C|4D|5E|\lambda \\
B &\to 1A|2B|3C|4D|5E \\
C &\to 1B|2B|3C|4D|5E
\end{aligned}$$

$$\begin{aligned} D &\rightarrow 1C|2C|3C|4D|5E \\ E &\rightarrow 1F|2F|3F|4D|5E \\ F &\rightarrow 1C|2C|3C|4F|5G \\ G &\rightarrow 3F|4F|5G \end{aligned}$$

From this grammar we can derive a set of mutual recurrence equations for calculating the number of words of length n in the language generated by this grammar. The recurrence equations are

$$\begin{aligned} a_0 &= 1, \\ b_0 &= c_0 = d_0 = e_0 = f_0 = g_0 = 0, \\ a_n &= a_{n-1} + b_{n-1} + c_{n-1} + d_{n-1} + e_{n-1}, \\ b_n &= a_{n-1} + b_{n-1} + c_{n-1} + d_{n-1} + e_{n-1}, \\ c_n &= 2b_{n-1} + c_{n-1} + d_{n-1} + e_{n-1}, \\ d_n &= 3c_{n-1} + d_{n-1} + e_{n-1}, \\ e_n &= 3f_{n-1} + d_{n-1} + e_{n-1}, \\ f_n &= 3c_{n-1} + f_{n-1} + g_{n-1}, \\ g_n &= 2f_{n-1} + g_{n-1}. \end{aligned}$$

This recurrence can be represented in matrix notation and the characteristic polynomial arising from this matrix is

$$CP(A) = 6x^3 - 10x^4 + 12x^5 - 7x^6 + x^7$$

From this we find that the largest eigenvalue is approximately 4.9258 and so the number of allowable permutations for our tree network is $O(5^n)$ as $n \rightarrow \infty$.

2.4 Binary Input Sequence

In Section 2.2 we considered permutation inputs. We shall now look at a more restricted scenario where the input sequence is chosen from the binary alphabet. In contrast to the previous case the order of the data items in the input sequence is important and so, instead of considering allowable permutations, we must consider allowable pairs. We shall give a necessary and sufficient condition for a pair of binary sequences (σ, τ) to be $N-$allowable for some network N and from this prove that the permutational power of a network over binary inputs is identical to a single buffer of a certain size.

Definition 2.1 *The* Blocking Capacity, *$\mathcal{B}(N)$, of a transportation network N is the maximum number of data items the network can contain and still be able to move a data item from the input to the output without outputting any other data item.*

For example, in a network with no cycles, containing n internal nodes and with m internal nodes on the shortest path from the input node to the output node we would have $\mathcal{B}(N) = n - m$.

As stated in Section 1.1 we now give formal definitions of irreducibility and standard computations which are valid for a transportation network operating on binary inputs.

Definition 2.2 *A pair of binary sequences (σ, τ) is* irreducible *if it is not possible to split it into two parts (α, β) and (γ, δ) satisfying*

- $\sigma = \alpha\gamma$
- $\tau = \beta\delta$
- $Length(\alpha) = Length(\beta)$
- $Length(\gamma) = Length(\delta)$
- $\sum \alpha_i = \sum \beta_i$
- $\sum \gamma_i = \sum \delta_i$

In characterising the binary allowable pairs of a transportation network we shall only consider an irreducible pair (σ, τ). This is a reasonable approach since any allowable pair is either irreducible or formed from the juxtaposition of several irreducible pairs.

Definition 2.3 *A* standard computation *of an allowable pair* (σ, τ) *is one in which no data item is inserted into the transportation network if it is possible to produce another element of* τ *from the data items already contained in the network.*

It follows from the irreducibility of (σ, τ) and the use of a standard computation that if the first data item in σ is a 1 then the network will contain a 1 for the entire computation (and similarly for a 0). If this were not the case then the network could have become empty at some point during the computation. Another consequence of using a standard computation on an irreducible pair is that if the first data item of the input is a 1 then there will never be more then one 0 in the network at a time. Since a 0 will only be inserted when it is the next data item required on the output. Similarly, if the first data item of the input is a 0 there will only ever be one 1 stored in the network at any time.

To state and prove the following lemmas we introduce an alternative representation of the pair (σ, τ). Let $\underline{a}$ be a vector where a_i represents the position of the i^{th}1 in σ. Similarly let $\underline{b}$ represent the 1's in τ, $\underline{c}$ represent the 0's in σ and $\underline{d}$ represent the 0's in τ.

Lemma 2.7 *An irreducible pair* $(1\alpha0, 0\beta1)$ *is N-allowable if and only if* $c_i - d_i \leq \mathcal{B}(N)$ *for all* i.

Proof: Suppose $c_i - d_i > \mathcal{B}(N)$ for some i. To output the i^{th} 0 it is necessary to process all the data items which precede the 0 in σ. There are $c_i - 1$ such data items, and only $d_i - 1$ of them can be output before the i^{th} 0. Therefore we must store the remaining $c_i - d_i$ data items in the network. To output the i^{th} 0 it would be necessary to store more than $\mathcal{B}(N)$ 1's in the network, input a 0, allow it to pass through the network and output it. However it is only possible to store $\mathcal{B}(N)$ data items and still move a new data item from the input to the output. Therefore the above is a necessary condition for allowability.

To show that the condition is sufficient we shall show how the computation transforming σ into τ would proceed. Initially a number of 1's must be read in and some of these must be stored in the network until the first 0 is reached. All of these 1's can be stored in the network and it is still possible to move a data item from the input node to the output node. The 0 can then pass through the network and be output. A number of the 1's which are currently stored are then output; this is possible since there is a path from every node to the output. The process is then repeated, reading in 1's until a 0 is reached, the 0 is output and then some 1's are output. Every time a 0 passes through the network the number of 1's which must be stored is $c_i - d_i$ and since this is no more than $\mathcal{B}(N)$ it will always be possible to store the 1's and allow a 0 through. Thus the pair is allowable and the condition is sufficient. □

Lemma 2.8 *An irreducible pair* $(0\alpha 1, 1\beta 0)$ *is N-allowable if and only if* $a_i - b_i \leq \mathcal{B}(N)$

Proof: Since 0 and 1 are simply labels and the operation of the transportation network is independent of their value we can exchange them. We then have a pair $(1\bar{\alpha}0, 0\bar{\beta}1)$ where $\bar{\alpha}$ is the complement of α. It also follows that $\underline{a}$ and $\underline{b}$ are the positions of the 0's in $\bar{\sigma}$ and $\bar{\tau}$ respectively. The result then follows immediately from Lemma 2.7 □

Lemma 2.9 *A pair of binary sequences* (σ, τ) *is N-allowable if and only if, when split into its irreducible parts* $(\alpha_1, \beta_1), \ldots (\alpha_r, \beta_r)$*, each irreducible pair satisfies the appropriate one of Lemma 2.7 or Lemma 2.8 depending upon the first data item of* α_i.

Proof: The necessity of the condition is clear. If (σ, τ) is allowable then it can be produced using a standard computation. Since, during a standard computation the network holds the least number of data items possible, the network will become empty at the end of each irreducible section. Therefore each irreducible section is allowable independently of the other irreducible sections.

Sufficiency is also clear. If each irreducible section is allowable then there is a computation C_i which produces β_i from α_i. It is immediate that the computation

$C = C_1C_2 \ldots C_r$ transforms σ into τ and so the pair is allowable. □

We now have a characterisation of the binary allowable pairs for a network N. As noted earlier these networks can be used to model various oblivious abstract container data types. Figures 7, 8, 9, 10 and 11 show some typical examples.

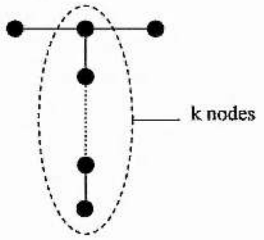


Figure 7: Network representing a stack of capacity k

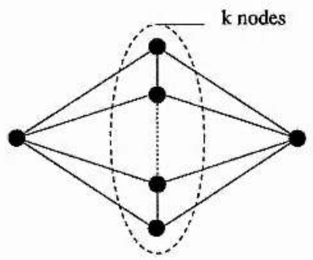


Figure 8: Network representing a buffer of capacity k

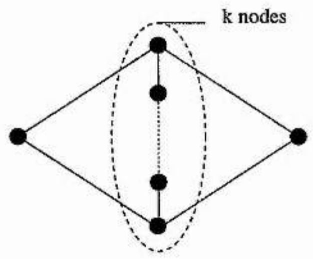


Figure 9: Network representing a deque of capacity k

Lemma 2.10 *Buffers, stacks, deques, input restricted deques and output restricted deques of capacity k and networks with a blocking capacity $\mathcal{B}(N) = k - 1$ have the same set of binary allowable pairs.*

Proof: The proof follows immediately from the inspection of Figures 7, 8, 9, 10 and 11. In each case the abstract container data type represented is one of capacity

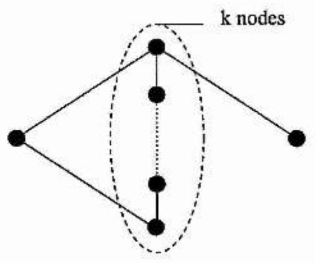


Figure 10: Network representing an input restricted deque of capacity k

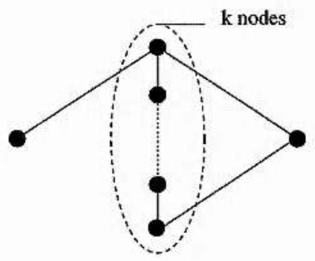


Figure 11: Network representing an output restricted deque of capacity k

k and is represented by a network with blocking capacity $k-1$. □

3 Oblivious Abstract Container Data Types

In this chapter we consider four well known abstract container data types whose behaviour is independent of the value of the data items being processed. These data types are the buffer, the queue, the deque and the stack. They are most easily described by giving the rules for inputting and outputting data items. In three cases the rule for inputting a data item is simply that if the data type is not full the data item is inserted. A deque differs from this since there is some choice about where the data item is inserted; it can insert data items at either the top or bottom of its queue. A buffer can output any data item it contains. A queue can output only the data item which has been in the data structure for longest and a deque can output data items from either the top or bottom of its queue. Finally a stack can only output the data item most recently inserted. There are two variations on a deque which we also consider: the input restricted deque and the output restricted deque. The input restricted deque can only input data items to the top of its queue but can still output from either end, and the output restricted deque can input data items to either end of its queue but can only output data items from the top. The other obvious modification, an input and output restricted deque, we do not consider because, if the input and output end of the queue are the same we have a stack and if they are different we have a standard queue.

There are a variety of existing results for queues and deques. In [Pra73] it is shown that two unbounded queues in parallel can produce $c_n = \binom{2n}{n}/(n+1)$ permutations of $1, 2, \ldots, n$. It is shown in [EI71] that the least number of unbounded queues in parallel required to produce a permutation σ is the minimum number of colours required to colour the permutation graph $\Pi(\sigma)$ and it is stated in [ELP72] that there are very efficient algorithms for solving this problem. Tarjan [Tar72] gives a characterisation of the permutations which can be sorted using m queues in parallel as those which avoid the pattern $m+1, m, \ldots, 1$. He also gives some partial results for networks of the type in Chapter 2 but using unbounded queues instead of unit buffers.

Knuth [Knu73a] shows that the generating function for the number of allowable permutations for an output restricted deque is $\frac{1}{2}\left(3 - z - \sqrt{1 - 6z + z^2}\right)$. He points out that the set of permutations which can be computed by an input restricted deque is in one to one correspondence with the set of permutations which can be computed by an output restricted deque. Therefore any enumeration or characterisation for one holds for the other with only minor modification. Pratt ([Pra73]) answers a question of Knuth's referring to *honest trees*, where every internal node has out degree at least two, by giving a $2 - 1$ correspondence between the allowable permutations of an output restricted deque and honest trees. It is also shown that an output restricted deque's allowable permutations are characterised by avoiding the patterns 4231 and 4132 and that an input restricted deque's allowable permutations are characterised by avoiding the patterns 4231 and 4213. An infinite set of patterns is then given which characterise the allowable permutations of an unbounded, unrestricted deque.

The stack data type has been widely studied by many people and significant connections between it and other combinatorial objects have been found. It operates a *last in first out* queuing strategy where the input and output operations are traditionally referred to as *push* and *pop* respectively. As with all oblivious abstract container data types, for permutation inputs, we investigate allowable permutations. The allowable permutations are closely related to the valid sequences of push and pop operations. Indeed the number of valid computation sequences of length $2n$ is the number of permutations of length n which the stack can produce and this is in a one to one correspondence with the number of balanced bracket sequences of length $2n$. There are then correspondences with trees ([Knu73a], [Rot75]), triangulations of polygons ([CLR92]), Young Tableaux ([Knu73b]), lattice paths ([Moh79]) and ballot sequences ([Knu73b], [RV78]). These connections are of great interest in the design of efficient algorithms (see [Knu73b], [Pra73], [Tar72]) and all point to the stack's fundamental role in giving precise expression to informal concepts such as "nesting", "structured decomposition" and "hierarchy".

It is known from the many correspondences above that, given an input sequence formed from distinct data items, there are c_n possible output sequences. It is also known that if the input sequence is $1, 2, \ldots, n$ then the allowable output sequences of an unbounded stack are those permutations which avoid the pattern 312 [Knu73a].

3.1 Buffers

The analysis of an unbounded buffer is trivial since any reordering of an input sequence is possible whether it be a sequence of distinct data items, binary data items or chosen from an arbitrary multiset. We need only insert the entire input sequence and output the data items in the order we desire.

We therefore turn our attention to a less trivial case where the buffer has a fixed capacity k.

Theorem 3.1 *For an input sequence $\sigma = 1, 2, \ldots, n$ and a buffer of capacity k there are precisely $k^{n-k}k!$ allowable permutations.*

Proof: At the point where $i - 1$ data items have been output, if $i \leq n - k$ then there remain at least k data items either in the buffer or still in the input stream. Of these there are precisely k which could be the next data item in the output sequence. If $i > n - k$ then there are $n - i + 1$ possibilities for the next output data item. It follows from this that there are a total of $k^{n-k}k!$ different output orders possible. □

Theorem 3.2 *The allowable permutations of a bounded buffer of capacity k with input sequence $1, 2, \ldots, n$ are precisely the sequences which avoid all patterns of length $k + 1$ which begin with their maximal element.*

Proof: Suppose an output permutation τ contains a pattern of length $k + 1$ which begins with its maximal data item. It follows immediately that, to generate the permutation, the k data items less than the maximal one, which form the pattern, must all be stored in the buffer. The maximal data item must then be inserted and removed from the buffer. This involves storing $k + 1$ data items in a buffer of capacity k and so the permutation cannot be generated by the buffer.

Suppose now that a permutation τ does not contain a pattern of length $k + 1$ which begins with its maximal data item and consider how the i^{th} data item can be generated. We assume $\tau_1 \ldots \tau_{i-1}$ have been produced and there are some data items stored in the buffer, possibly including τ_i. If τ_i is in the buffer then it can be produced immediately since a buffer can output any data item it contains. If τ_i is not in the buffer then we must read in some number of data items from the input until we have read in τ_i. This must be possible for, to fail, we would have to be forced to store k or more data items before reading in τ_i. This implies that k data items less than τ_i occur in τ after τ_i which contradicts the assumption that the patterns are avoided. □

This gives us a simple characterisation of the allowable pairs of a buffer of capacity k but we introduce a second characterisation which we shall use in the proof of Lemma 3.2.

Lemma 3.1 *The allowable permutations, τ, of a buffer of capacity k with input sequence $1, 2, \ldots, n$ are precisely those which satisfy $\tau_i \leq i + k - 1$.*

Proof: Suppose τ does not satisfy the condition, then there is a data item $\tau_i > i + k - 1$. Therefore at least $i + k - 1$ data items in τ are smaller than τ_i. Since τ_i appears in the i^{th} position of τ at least k data items smaller than τ_i occur after the i^{th} position. It follows from Theorem 3.2 that τ is not an allowable permutation for a buffer of capacity k.

Alternatively, if τ satisfies the condition, then τ is allowable because it cannot contain a pattern of length $k + 1$ which begins with its maximal data item. Suppose it did contain such a pattern and that τ_i is the left most data item in τ which matches the maximal data item in the pattern. Then all the data items to the left of τ_i are less than τ_i and there are less than $\tau_i - k$ of them because k of them must come after τ_i to form the pattern. So $i < \tau_i - k + 1$ which means $\tau_i > i + k - 1$; a contradiction. □

We can compose two buffers in series by connecting the output of one to the input of the other. We now go on to consider systems formed in this way. We also consider parallel composition, where two or more buffers can all take data items from the input sequence and place them on a shared output sequence. This arrangement is depicted in Figure 12 for two buffers.

Lemma 3.2 *The serial combination of buffers of capacities $k_1, k_2, \ldots, k_r$ is equivalent to a single buffer of capacity $\sum_{i=1}^{r} k_i - (r-1)$*

Proof: The result follows inductively if we show it holds for $r = 2$ and so we consider only this case. Let $L(B_h B_k)$ denote the set of permutations that can be output by the series combination of two buffers of sizes h and k and $L(B_{h+k-1})$ the set of permutations that can be output by a single buffer of size $h+k-1$. Let π be any permutation of $L(B_h B_k)$. Then π is the permutation composition $\pi_1\pi_2$, where π_1 and π_2 are permutations that can be output by buffers of sizes h and k respectively. It follows from Theorem 3.1 that $\pi(i) = \pi_1(\pi_2(i)) \leq \pi_2(i) + h - 1 \leq i + h + k - 2$ and thus, by Theorem 3.1 again, $\pi \in L(B_{h+k-1})$.

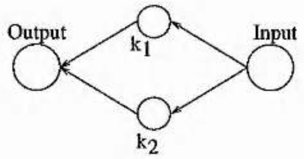


Figure 12: Two buffers in Parallel

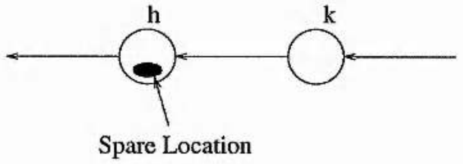


Figure 13: Two buffers in series simulating a single buffer

To show the reverse we use a simulation argument. We shall show how the two buffers in series can behave as one buffer of size $h + k - 1$ by allocating one of the locations in the second buffer as *spare*. The remaining spaces behave as a single buffer of size $h + k - 1$. This arrangement is shown in Figure 13. It is always possible to input a data item into the series arrangement so long as there are fewer than $h+k-1$ data items already in the pair of buffers, although it may first be necessary to transfer a data item from the first buffer to the second one. Further, any data item among the $h + k - 1$ or fewer data items present in the system can be output. If the data item is contained in the later buffer then it can be output immediately, otherwise it can be moved from the earlier buffer into the spare location in the later buffer and then output. This is the only time a data item is placed in the spare location. It follows that the two buffers in series can perform any sequence of input and output operations that the single buffer can perform. □

For parallel composition we have a similar result.

Lemma 3.3 *The parallel combination of buffers of capacities $k_1, k_2, \ldots, k_r$ is equivalent to a single buffer of capacity $\sum_{i=1}^{r} k_i$.*

Proof: As with the proof for Lemma 3.2 we only need to show the result for $r = 2$ because a simple inductive argument extends this to an arbitrary number of buffers.

Suppose we have two buffers of capacities k_1 and k_2, both of which can take a data item from the input sequence and can place a data item on the output sequence. This system is depicted in Figure 12. Suppose a permutation τ is allowable by a buffer of size $k_1 + k_2$ and is computed by some sequence, C, of input and output operations. We show inductively that the parallel system of two buffers can directly simulate the computation to produce the same permutation.

The inductive hypothesis is that the parallel system contains exactly the same data items as the single buffer. The base case is clearly true since both systems start in the empty state. For the inductive step suppose the i^{th} operation is an input operation, then it must be the case that there are less than $k_1 + k_2$ data items stored in the single buffer. Therefore there are less than $k_1 + k_2$ data items stored in the parallel system and so there is space to input another data item into one of the buffers and it does not matter which one is used. Alternatively, if the operation is an output then the data item being output from the single buffer must be contained in one of the two buffers in parallel and so it can be output from that system. Therefore τ can be produced by the two buffers in parallel.

An identical argument shows that the single buffer can simulate the two buffers in parallel and so they have exactly the same allowable permutations. □

It is important to note that these two results do not combine together. The results apply only to isolated systems formed as described. Consider as a counter example the two networks in Figure 14. One might hope that the left hand system could be reduced to the right hand one using Lemma 3.2. However, the permutation $(7, 1, 2, 3, 4, 5, 6)$ can be produced by the left hand system but not the right.

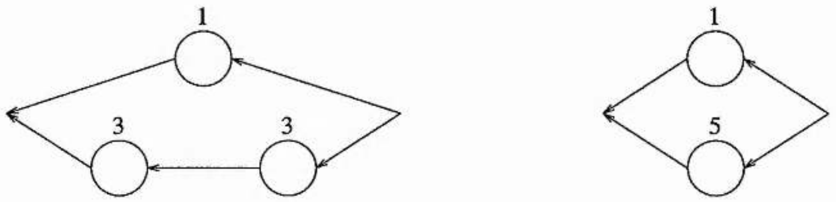


Figure 14: Two inequivalent systems

When we restrict the input to the binary alphabet we can find similar results to those above. First we present a recurrence for the number of binary allowable pairs.

Lemma 3.4 *For a bounded dictionary of capacity k the number $x_{k,n}$ of binary allowable pairs of length n satisfies the recurrence*

$$\begin{aligned} x_{k,n} &= \binom{2n}{n} \textit{ for } n \le k, \\ x_{k,n} &= \sum_{i=1}^{r}(-1)^{i+1} x_{k,n-i} a_{k,i} \textit{ with } r = \left\lceil \tfrac{k}{2} \right\rceil, \\ a_{k,0} &= 1, \\ a_{k,i} &= 2\binom{k-i}{i-1} + \binom{k-i}{i}. \end{aligned}$$

Proof: The proof of the base case is trivial. If there are no more than k data items to be input we can read them all in and output them in any order. Therefore there are

$$\sum_{i=0}^{n} \binom{n}{i}^2 = \binom{2n}{n}$$

allowable pairs.

For the general case fix k and let $w_n^{(i)}$ be the number of allowable pairs of length n of the form $(0^i\alpha, \beta)$, then obviously $x_{k,n} = w_n^{(0)}$, and

$$\begin{aligned} w_{n+1}^{(0)} &= 2w_{n+1}^{(1)}, \text{ for } n \ge 0, \\ w_{n+1}^{(i)} &= w_n^{(i-1)} + \sum_{j=i}^{k-1} w_n^{(j)}, \text{ for } n \ge i > 0. \end{aligned} \tag{1}$$

The first of these two equations holds because the number of pairs of the form $(0\alpha, \beta)$ is the same as the number of pairs of the form $(1\alpha, \beta)$ and all allowable pairs have one of these two forms.

For the second notice that all pairs $(0^i\alpha, \beta)$, of length $n+1$, either have the form $(0^i\alpha, 0\beta')$ or $(0^i\alpha, 1\beta')$. There are $w_n^{(i-1)}$ pairs of the first form because the first 0 is input and immediately output, and then there are $w_n^{(i-1)}$ ways to complete the pair. The second can be split into several further forms, $(0^j 1\gamma, 1\beta')$ for $i \le j < k$. For each of these all j 0's and the 1 must be inserted and the 1 immediately output. There are then $w_n^{(j)}$ ways to complete the pair and thus there are $\sum_{j=i}^{k-1} w_n^{(j)}$ allowable pairs of the second form.

We can represent this recurrence more succinctly using matrix notation. Let

$$w_n = \begin{pmatrix} w_n^{(1)} \\ \vdots \\ w_n^{(k-1)} \end{pmatrix}, \; A_k = \begin{pmatrix} 3 & 1 & \cdots & 1 & 1 \\ 1 & 1 & \cdots & 1 & 1 \\ 0 & 1 & \cdots & 1 & 1 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 1 & 1 \end{pmatrix}$$

then recurrence equation (1) becomes

$$w_{n+1} = A_k w_n$$

Hence, for any constants $d_0, d_1, \ldots d_t$, $\sum_{i=0}^{t} d_i w_{n+i} = \sum_{i=0}^{t} d_i A_k^i w_n$. If we choose the constants so that $\sum_{i=0}^{t} d_i \lambda^i$ is the characteristic polynomial of A then, by the Cayley-Hamilton Theorem ([Lan69], p131), we shall have $\sum_{i=0}^{t} d_i w_{n+i} = 0$. We can derive a recurrence equation for the characteristic polynomial, $u_{k-1}(\lambda) = det(A_k - \lambda I)$, of A_k by subtracting the $(k-2)^{th}$ column of the determinant from the $(k-1)^{th}$ column and expanding it by the $(k-1)^{th}$ column. It is then easily seen that

$$u_m(\lambda) = -\lambda(u_{m-1}(\lambda) + u_{m-2}(\lambda)), \text{ for all } m \ge 3.$$

The initial cases

$$\begin{aligned} u_1(\lambda) &= 3 - \lambda, \\ u_2(\lambda) &= \lambda^2 - 4\lambda + 2. \end{aligned}$$

are calculated directly.

From the recurrence it follows easily by induction on k that there exists polynomials y_{2r} and y_{2r+1} each of degree $r+1$ for which

$$\begin{aligned} u_{2r}(\lambda) &= \lambda^{r-1} y_{2r}(\lambda), \qquad (2)\\ u_{2r+1}(\lambda) &= \lambda^{r} y_{2r+1}(\lambda), \qquad (3)\end{aligned}$$

and that the polynomials satisfy

$$\begin{aligned} y_{2r}(\lambda) &= -\lambda y_{2r-1}(\lambda) - y_{2r-2}(\lambda),\\ y_{2r+1}(\lambda) &= -y_{2r}(\lambda) - y_{2r-1}(\lambda).\end{aligned}$$

Now a standard inductive proof using binomial coefficient identities proves

$$y_{k-1}(\lambda) = (-1)^{k-1}\sum_{i=0}^{r} \lambda^{r-i}(-1)^i a_{k,i}, \text{ where}$$
$$r = \left\lceil \tfrac{k}{2} \right\rceil, a_{k,0} = 1, a_{k,i} = 2\binom{k-i}{i-1} + \binom{k-i}{i}.$$

Combining this with (2) and (3) gives

$$\begin{aligned} u_{k-1}(\lambda) &= \lambda^{\lfloor \frac{k-2}{2} \rfloor} y_{k-1}(\lambda)\\ &= (-1)^{k-1}\sum_{i=0}^{r} \lambda^{r-i+\lfloor \frac{k-2}{2} \rfloor}(-1)^i a_{k,i}\end{aligned}$$

Therefore the sequence (w_n) satisfies

$$\sum_{i=0}^{r} w_{r+\lfloor \frac{k-2}{2} \rfloor + n - i}(-1)^{i+1} a_{k,i} = 0, \text{ where } r = \left\lceil \tfrac{k}{2} \right\rceil.$$

Since $r + \left\lfloor \frac{k-2}{2} \right\rfloor \leq k$, $w_n^{(1)}$ is a component of the vector w_n and $x_{k,n} = w_n^{(0)} = 2w_n^{(1)}$, we have for $n \geq k$

$$x_{k,n} = \sum_{i=1}^{r} x_{k,n-i}(-1)^{i+1} a_{k,i}$$

□

Having counted the allowable pairs we now recall Lemma 2.10. From this we immediately have

Theorem 3.3 *For a buffer, stack, deque, input restricted deque or output restricted deque of capacity k or a network N with blocking capacity $\mathcal{B}(N) = k - 1$ the number of binary allowable pairs is given by*

$$\begin{aligned} x_{k,n} &= \binom{2n}{n}, \text{ for } n \leq k, \\ x_{k,n} &= \sum_{i=1}^{r} (-1)^{i+1} x_{k,n-i} a_{k,i}, \text{ with } r = \left\lceil \tfrac{k}{2} \right\rceil, \\ a_{k,0} &= 1, \\ a_{k,i} &= 2\binom{k-i}{i-1} + \binom{k-i}{i}. \end{aligned}$$

□

Another consequence of Lemma 2.10 is that the characterisation of allowable pairs given in Lemma 2.9 is valid for buffers, stacks, deques, input restricted deques and output restricted deques. This characterisation provides the basis for a linear time algorithm which checks the allowability of a pair of binary sequences.

The operation of the algorithm follows the characterisation of allowable pairs closely. Given a pair of binary sequences of the form $(\sigma, \tau) = (1\alpha 0, 0\beta 1)$ we scan both from the left until the first data items of value 0 are located. If these occur at positions i and j in the input and output sequences we then check if $i - j \leq k$. If i and j do not satisfy this then the pair is not allowable; otherwise we scan for the next 0's and repeat. If the pair has the form $(0\alpha 1, 1\beta 0)$ we scan for 1's instead of 0's. Finally there are some special cases to check for; pairs of the form $(0, 0)$ and $(1, 1)$ and a mismatch in the number of data items of value 1 in the sequences. These can all be catered for in the single pass of the sequences and so the algorithm operates in linear time.

As in the permutation case we now go on to investigate the power of several buffers composed in series and in parallel. The results of this investigation are identical to those for the permutation case. For buffers in series we have a version of Lemma 3.2.

Theorem 3.4 *The serial composition of buffers of capacities $k_1, k_2, \ldots, k_r$ with binary inputs is equivalent to a single buffer of capacity $\sum_{i=1}^{r} k_i - (r - 1)$*

Proof: We show that this result is a direct consequence of Lemma 3.2. Suppose (σ, τ) is a binary allowable pair. It is possible for some fixed data item σ_i to occur in one of various different positions in τ and for any one of these positions the output sequence τ can be produced. For example in the pair $(110, 101)$ the first 1 in the output sequence could be either of the two 1's in the input sequence. By attaching unique labels $1, 2, \ldots, n$ to the data items in the input sequence we can see more clearly what is happening. The input sequence becomes $1_1 1_2 0_3$ and the output could be either $1_1 0_3 1_2$ or $1_2 0_3 1_1$. In this case an abstract container data type can generate 101 from 110 if and only if it can produce one of the permutations 132 or 231. For any binary pair there will always be at least one characterising permutation which is allowable if and only if the binary pair is. We shall denote this permutation $\pi_{(\sigma,\tau)}$.

The result is now immediate since a binary pair (σ, τ) is allowable for a series system of buffers if and only if $\pi_{(\sigma,\tau)}$ is allowable for that system. By Lemma 3.2 this permutation is allowable for the system if and only if it is allowable for a single buffer of capacity $\sum_{i=1}^{r} k_i - (r-1)$. Finally, the permutation is allowable for this single buffer if and only if the binary pair (σ, τ) is allowable for the single buffer. □

For buffers composed in parallel we have a version of Lemma 3.3.

Theorem 3.5 *The parallel composition of buffers of capacities $k_1, k_2, \ldots, k_r$ with binary inputs is equivalent to a single buffer of capacity $\sum_{i=1}^{r} k_i$.*

Proof: This proof is identical to that of Theorem 3.4. We apply the same labelling technique and see that a binary pair (σ, τ) is allowable for a system of buffers in parallel if and only if $\pi_{(\sigma,\tau)}$ is allowable for that system. By Lemma 3.3 this is if and only if $\pi_{(\sigma,\tau)}$ is allowable for a single buffer of capacity $\sum_{i=1}^{r} k_i$. Finally the permutation is allowable for the single buffer if and only if the binary pair (σ, τ) is allowable for the single buffer. □

3.2 Deques

Many of the problems relating to unbounded deques have already been solved; the allowable permutations for input restricted, output restricted and unrestricted deques have all been characterised by pattern avoidance, a generating function has been given for the number of allowable permutations for an output restricted deque and there is a two to one correspondence between allowable permutations of an output restricted deque and honest trees which permits the derivation of a recurrence for the number of allowable permutations.

We can, however, still contribute to the area by applying the technique developed in Chapter 2 to some specific bounded deques. We present the grammars for unrestricted deques, as shown in Figure 9, for $k = 3, 4$ and 5. We also give the final result of the analysis for $k = 6$ where the detailed derivation would be of little interest. We then give some similar results for input and output restricted deques.

The grammar for a deque of capacity 3 is given in (4) and for capacity 4 in (5). Both of these are very concise grammars with k non terminal symbols and a regular structure. This leads to an easy analysis of the corresponding matrix resulting in integral maximal eigenvalues which are 3 and 4 respectively.

$$\begin{array}{rcl} A & \rightarrow & 1A|2B|3C|\lambda \\ B & \rightarrow & 1A|2B|3C \\ C & \rightarrow & 1B|2B|3C \end{array} \tag{4}$$

$$\begin{array}{rcl} A & \rightarrow & 1A|2B|3C|4D|\lambda \\ B & \rightarrow & 1A|2B|3C|4D \\ C & \rightarrow & 1B|2B|3C|4D \\ D & \rightarrow & 1C|2C|3C|4D \end{array} \tag{5}$$

We may hope, after seeing the first two cases, that a deque of capacity 5 would give rise to a grammar with 5 non terminal symbols and a corresponding largest eigenvalue of 5 but unfortunately this is not the case. The grammar, shown in (6), has 9 non terminal symbols and the largest eigenvalue is approximately 4.85577 (The largest root of $x^3 - 7x^2 + 10x + 2$)

$$\begin{aligned} A &\rightarrow 1A|2B|3C|4D|5E|\lambda \\ B &\rightarrow 1A|2B|3C|4D|5E \\ C &\rightarrow 1B|2B|3C|4D|5E \\ D &\rightarrow 1C|2C|3C|4D|5E \\ E &\rightarrow 1F|2F|3D|4D|5E \\ F &\rightarrow 1C|3C|4F|5G \\ G &\rightarrow 1F|3H|4F|5G \\ H &\rightarrow 2C|3C|4H|5I \\ I &\rightarrow 2F|3F|4H|5I \end{aligned} \tag{6}$$

The next case, where the capacity is 6, results in a grammar containing 25 non terminal symbols and the largest corresponding eigenvalue is approximately 5.483 (The largest root of $x^{17} - 20x^{16} + 169x^{15} - 788x^{14} + 2179x^{13} - 3390x^{12} + 1698x^{11} + 3988x^{10} - 8985x^9 + 6836x^8 + 1335x^7 - 6592x^6 + 3697x^5 + 1594x^4 - 2066x^3 - 168x^2 + 460x + 100$). It is clear that the rate of increase of the sequence of eigenvalues is decreasing but nothing is known of the asymptotic behaviour.

We can carry out similar analysis for an output restricted deque and obtain the following results. The grammar for an output restricted deque of capacity 3 is identical to that for an unrestricted deque, given in (4). The grammar for output restricted deques of capacity 4 and 5 are given in (7) and (8).

$$\begin{aligned} A &\rightarrow 1A|2B|3C|4D|\lambda \\ B &\rightarrow 1A|2B|3C|4D \\ C &\rightarrow 1B|2B|3C|4D \\ D &\rightarrow 1E|2E|3C|4D \\ E &\rightarrow 1B|3E|4F \end{aligned} \tag{7}$$

$$F \rightarrow 1E|3E|4F$$

$$\begin{aligned}
A &\rightarrow 1A|2B|3C|4D|5E|\lambda \qquad (8)\\
B &\rightarrow 1A|2B|3C|4D|5E\\
C &\rightarrow 1B|2B|3C|4D|5E\\
D &\rightarrow 1F|2F|3C|4D|5E\\
E &\rightarrow 1G|2G|3H|4D|5E\\
F &\rightarrow 1B|3F|4I|5J\\
G &\rightarrow 1F|4G|5K\\
H &\rightarrow 1F|2F|4H|5L\\
I &\rightarrow 1F|2F|4I|5J\\
J &\rightarrow 1G|3G|4I|5J\\
K &\rightarrow 1G|4G|5K\\
L &\rightarrow 1G|2G|4H|5L
\end{aligned}$$

The largest eigenvalues corresponding to these grammars are 3, $2+\sqrt{3}$ (approximately 3.732) and $2+\sqrt{5}$ (approximately 4.236) respectively. For an output restricted deque of capacity 6 we find a grammar which contains 24 non terminal symbols and gives a largest eigenvalue of approximately 4.586 (the largest root of $x^4-6x^3+6x^2+2x+1$). As we would expect these are slightly lower than the values for the unrestricted deque. Again, the rate of increase appears to be decreasing. Knuth shows in [Knu73a] that in the unbounded capacity, output restricted case the number of allowable permutations is $O\left((3+\sqrt{8})^n/\sqrt{n^3}\right)$. It follows that the sequence of eigenvalues is bounded above by $3+\sqrt{8}$, (approximately 5.8).

When we look at input restricted deques something quite surprising happens. The direct correspondence between allowable permutations for an input restricted deque and allowable permutations of an output restricted deque would lead us to expect the grammars to be similar and possibly even to share the same structure. However it turns out that the grammars for input restricted deques are very regular and for capacity k the grammar has k non terminal symbols. Once again, for $k = 3$, the grammar is identical to that given in (4). For $k = 4$ and 5 the grammars are given in (9) and (10).

$$\begin{aligned} A &\rightarrow 1A|2B|3C|4D|\lambda \\ B &\rightarrow 1A|2B|3C|4D \\ C &\rightarrow 1B|2B|3C|4D \\ D &\rightarrow 1C|3C|4D \end{aligned} \tag{9}$$

$$\begin{aligned} A &\rightarrow 1A|2B|3C|4D|5E|\lambda \\ B &\rightarrow 1A|2B|3C|4D|5E \\ C &\rightarrow 1B|2B|3C|4D|5E \\ D &\rightarrow 1C|3C|4D|5E \\ E &\rightarrow 1D|4D|5E \end{aligned} \tag{10}$$

The following two lemmas allow us to write down the grammar for an input restricted deque of any fixed capacity k. This grammar is shown in (11).

Lemma 3.5 *If an input restricted deque with input sequence $1, 2, \dots$ contains i data items then those data items form an increasing sequence in the deque with the largest data item being at the end attached to the input edge.*

Proof: For simplicity we assume the input edge goes to the top of the deque, as depicted in Figure 10, Section 2.4. We proceed by induction; the base case being clear since if the deque is empty all the data items are indeed in increasing order towards the top. Suppose the deque contains $i - 1$ data items and the i^{th} is just about to be inserted. Certainly the new data item is larger than all the others since the input sequence is $1, 2, \dots, n$ and since the existing data items are in increasing order towards the top end the new set of data items are in order as required. The only other way in which the contents of the deque can change is if a data item is removed. Whether the data item is removed from the top or the bottom the remaining data items are clearly still in increasing order towards the top of the deque. □

Lemma 3.6 *An input restricted deque of capacity k with input sequence $1, 2, \ldots$ and which contains i data items can output only the data items of rank $1, i, \ldots, k$ amongst those not yet output.*

Proof: Lemma 3.5 tells us that there is essentially only one state that an input restricted deque containing i data items can be in. Although there could be many dispositions of the data items over the nodes of the deque they all have the data items in increasing order from the bottom to the top.

A consequence of this is that the top and bottom data items of this sequence are the only ones currently in the deque which can be output. These are the data items of rank 1 and i among those which have not yet been output. The only other way to output a data item is to read it in from the input and then output it. To output a data item of rank $j > i$ it is necessary to store j data items in the deque, being those of rank $1, 2, \ldots, j$ in the set of data items not yet output. It follows that this is possible for $j = i \ldots k$. □

We now construct the grammar using the above two lemmas. We shall use non terminal symbols $S_0, S_1, \ldots S_k$ where S_i corresponds to the input restricted deque containing i data items. Lemma 3.5 tells us we only need one non terminal for each i and Lemma 3.6 tells us that the productions for S_i are $S_i \rightarrow 1S_{i-1}$, $S_i \rightarrow iS_{i-1}$ and $S_i \rightarrow jS_{j-1}$ for $j = i + 1 \ldots k$. We add one extra production, $S_0 \rightarrow \lambda$, and the resulting grammar is shown in (11).

$$\begin{aligned} S_0 &\rightarrow 1S_0|2S_1|3S_2\ldots|kS_{k-1}|\lambda \\ S_1 &\rightarrow 1S_0|2S_1|3S_2\ldots|kS_{k-1} \\ S_2 &\rightarrow 1S_1|2S_1|3S_2\ldots|kS_{k-1} \\ &\vdots \end{aligned} \tag{11}$$

$$\begin{aligned} S_i &\rightarrow 1S_{i-1}|iS_{i-1}|i+1S_i|\ldots|kS_{k-1} \\ &\vdots \\ S_{k-1} &\rightarrow 1S_{k-2}|k-1S_{k-2}|kS_{k-1} \end{aligned}$$

The matrix associated with this grammar is

$$\begin{pmatrix} 1 & 1 & 1 & 1 & \ldots & 1 & 1 \\ 1 & 1 & 1 & 1 & \ldots & 1 & 1 \\ 0 & 2 & 1 & 1 & \ldots & 1 & 1 \\ 0 & 0 & 2 & 1 & \ldots & 1 & 1 \\ \vdots & & & \ddots & & & \vdots \\ \vdots & & & & \ddots & & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 2 & 1 \end{pmatrix}$$

Having a general form for the matrix allows us to automate the calculation of the eigenvalues and so we can generate a significantly longer list than was possible in the previous cases. The list is presented in Figure 15.

n	Largest Eigenvalue	n	Largest Eigenvalue
3	3	11	5.345
4	3.732	12	5.411
5	4.236	13	5.464
6	4.586	14	5.508
7	4.836	15	5.545
8	5.019	16	5.575
9	5.156	17	5.601
10	5.262	18	5.623

Figure 15: Eigenvalues for an input restricted deque

3.3 Stacks

We can apply the technique of Chapter 2 to bounded capacity stacks on permutation inputs and we find that the grammars which are generated have a regular structure, as was the case for an input restricted deque.

Figure 16 tabulates the value of the greatest eigenvalue for stacks of capacity 1 to 5. It is known that the Catalan numbers, $c_n = \binom{2n}{n}/(n+1)$, are asymptotic to 4^n. Since there are c_n allowable permutations for an unbounded stack on input length n we know that $\alpha_k \to 4$ as $k \to \infty$.

Stack Capacity	α_k	Numerical Estimate
1	1	1
2	2	2
3	$\frac{3+\sqrt{5}}{2}$	2.61
4	3	3
5	largest root of $x^3 - 5x^2 + 6x - 1$	3.247

Figure 16: α_k for small values of k

In [dBKR72] some work was done on finding the average height of planted plane trees with a fixed number of nodes. This corresponds to the average capacity required by a stack to compute a randomly chosen permutation of length n. The result is that, on average,

$$k = \sqrt{\pi n} - \frac{1}{2} + O\left(\frac{1}{\sqrt{n}} \log n\right)$$

stack locations are required.

4 Priority Queues and Double Ended Priority Queues

The study of priority queues warrants a separate chapter for two reasons; the first is simply the quantity of results from this area, both existing and new. The second reason is the more important; there is a fundamental difference between a priority queue and any of the other container data types considered so far in this thesis. This difference is that the behaviour of the priority queue is dependent on the *value* of the priority attached to the data items it is processing as well their order in the input sequence. Usually this priority is taken to be the value of the data item but it may differ as we shall see in Section 4.2. The dependency on priority is clearly illustrated by considering the behaviour of a priority queue on two input sequences, $1, 2, \ldots, n$ and its reverse $n, n-1, \ldots, 1$. With the first input sequence the possible output permutations are those that an ordinary queue could produce, namely just $1, 2, \ldots, n$. From the second input sequence a priority queue can produce precisely the permutations which a stack can produce.

There are a great many existing results in this field, mostly based on binary and permutation input sequences and exclusively based on unbounded priority queues. For permutation inputs and an unbounded priority queue it is shown in [AT93] that there are $(n+1)^{n-1}$ allowable pairs of length n. A combinatorial proof of this is then presented in [AB94]. This paper also presents an $O(n)$ time algorithm for finding $s(\tau)$ and an $O(n^4)$ time algorithm for $t(\sigma)$. The language associated with a priority queue can be thought of as a relation on sequences. With this viewpoint the transitive closure of the allowability relation for a priority queue is found and a one-to-one correspondence between the allowable pairs and labelled trees is given. Independently and subsequently [GZ94] developed a different direct correspondence between the allowable pairs of priority queue and labelled trees and an alternative linear time algorithm for finding $s(\tau)$.

In [Atk93] a characterisation, the *partial sum criterion*, of the allowable pairs of an unbounded priority queue with a binary input sequence is given. It is also shown that there are c_{n+1} allowable pairs of length n, this being the same as the number of allowable permutations of length $n+1$ for a stack with permutation inputs.

Algorithms are presented which find $s(\tau)$ and $t(\sigma)$ in $O(n^2)$ time and some symmetry results are presented: a pair (σ, τ) is allowable if and only if (τ^r, σ^r) is allowable if and only if $(\bar{\tau}, \bar{\sigma})$ is allowable.

In [ALW95] some progress has been made in the case when the input is chosen from a multiset. If the input and output sequences are reorderings of a multiset $S = \{1^{a_1}, 2^{a_2}, \ldots, k^{a_k}\}$ then the number of allowable pairs is

$$\frac{1}{n+1}\prod_{i=1}^{k}\binom{n+1}{a_i}$$

This is proved by giving a one to one correspondence between the allowable pairs and k-way trees. Algorithms are also presented which find $t(\sigma)$ in $O(n^4)$ time and $s(\tau)$ in $O(n^2)$ time.

Double ended priority queues are studied in [Thi93] where various results are given and some conjectures are stated. No closed formula or recurrence is found for the number of allowable pairs of a double ended priority queue but a linear time allowability test is presented. In [Lin94] an alternative characterisation of the allowable pairs of a double ended priority queue is given in terms of the avoidance of pairs of patterns.

In this chapter we further investigate priority queues for permutation, binary and multiset inputs. The focus here is on bounded capacity priority queues although we do present some new results in the unbounded case. We also present a few small results relating to double ended priority queues with permutation inputs. Some of this work is also presented in [ATar].

4.1 Priority Queue with Permutation Inputs

A bounded priority queue on permutation inputs is a very interesting case which has turned out to be particularly difficult to solve. There is no general formula for the number of allowable pairs of length n for a priority queue of size k given here, nor is there a characterisation of the allowable pairs. We do, however, make some progress for the special case $k = 2$.

Lemma 4.1 *If σ and τ are permutations of $1, 2, \ldots, n$ then (σ, τ) is 2-allowable if and only if there exist decompositions of σ and τ into substrings such that*

$$\begin{aligned}\sigma &= \alpha n \beta \gamma \\ \tau &= \delta \beta n \epsilon\end{aligned}$$

where each of (α, δ) and (γ, ϵ) are also 2-allowable

Proof: Suppose first that there is a sequence of Insert and Delete-Minimum operations that, using a capacity 2 priority queue, transforms σ into τ. At the point that the symbol n is inserted in the priority queue all the symbols of σ which precede n (the segment α) will have been inserted and all except possibly one will have been output; the remaining symbol of α, if any, will then be output immediately since it is smaller than n and no Insert operation is possible if the priority queue contains two items. Thus τ will have an initial segment δ with (α, δ) 2-allowable.

After δ has been generated the priority queue will contain only the symbol n. Since the priority queue has capacity 2 there must then be a number (possibly zero) of pairs of Insert, Delete-Minimum operations followed by a Delete-Minimum which copies a segment β of σ into the output τ and then outputs n. The priority queue will now be empty and the remaining segment γ of σ will be transformed into a final segment ϵ of τ so that (γ, ϵ) will be 2-allowable.

To prove the implication in the other direction suppose we have a pair of sequences (σ, τ) which have the required decomposition. It is clear from this decomposition that the initial segment α of σ can be transformed into δ. The priority queue is then empty and it is clearly possible to transform $n\beta$ into βn. Finally γ can be transformed into ϵ and so (σ, τ) is allowable. □

Corollary 4.2 *If σ and τ are permutations of $1, 2, \ldots, n$ with $\sigma = \alpha n \beta$ and $\beta = \beta_1 \beta_2 \ldots \beta_r$ then*

$$T_2(\alpha n \beta) = T_2(\alpha) T_2(n \beta) \tag{12}$$

$$T_2(n \beta) = \{n\} T_2(\beta) \cup \{\beta_1\} T_2(n \beta_2 \ldots \beta_r) \tag{13}$$

where juxtaposition XY of sets denotes the set $\{xy | x \in X, y \in Y\}$

Proof: Equation (12) follows from an application of Lemma 4.1, since we have

$$\begin{aligned} T_2(\alpha n \beta) &= T_2(\alpha)\{\gamma n T_2(\delta) \mid \gamma\delta = \beta\} \\ &= T_2(\alpha) T_2(n\beta) \end{aligned}$$

For equation (13) notice that $T_2(n\beta)$ consists of those outputs arising from inserting and immediately deleting n from the priority queue together with those that are obtained by inserting n, inserting β_1, and then deleting β_1. The first of these sets is $\{n\}T_2(\beta)$ and the second is $\{\beta_1\}T_2(n\beta_2 \ldots \beta_r)$. □

Since the union in (13) is a disjoint union we have the further corollary.

Corollary 4.3

$$\begin{aligned} t_2(\alpha n \beta) &= t_2(\alpha) t_2(n\beta) \\ t_2(n\beta) &= t_2(\beta) + t_2(n\beta_2 \ldots \beta_r) \\ &= 1 + \sum_{i=0}^{r-1} t_2(\beta_{i+1} \ldots \beta_r) \end{aligned}$$

The total number of 2-allowable pairs of permutations is the sum of $t_2(\sigma)$ over all possible input sequences σ. We therefore write

$$x_n = \sum_{\sigma} t_2(\sigma) \tag{14}$$

Although we have no closed form for x_n we can find both a recurrence for the values and the exponential generating function for the sequence (x_n).

Theorem 4.1 *The exponential generating function of the sequence (x_n) is*

$$\begin{aligned} U(t) &= \sum \frac{x_n t^n}{n!} \\ &= \frac{1}{1 + \log(1-t)} \end{aligned}$$

Proof: We shall express each permutation σ in the form $\alpha n \beta$ in (14) and sum over all possibles sizes of α. For each size we then consider the number of ways we can choose the elements of α and sum over all permutations of those elements. Finally we sum over all permutations of the remaining elements, β, giving

$$\begin{aligned} x_n &= \sum_{i=0}^{n-1} \binom{n-1}{i} \sum_{|\alpha|=i} \sum_{|\beta|=n-i-1} t_2(\alpha) \times t_2(n\beta) \\ &= \sum_{i=0}^{n-1} \binom{n-1}{i} \sum_{|\alpha|=i} t_2(\alpha) \sum_{|\beta|=n-i-1} t_2(n\beta) \\ &= \sum_{i=0}^{n-1} \binom{n-1}{i} x_i \sum_{|\beta|=n-i-1} t_2(n\beta) \end{aligned}$$

To handle the inner summation put

$$y_m = \sum_{|\beta|=m} t_2(n\beta_1 \ldots \beta_m)$$

where β runs over all permutations of a fixed set of size m. Then

$$\begin{aligned} y_m &= \sum_{|\beta|=m} t_2(\beta_1 \ldots \beta_m) + t_2(n\beta_2 \ldots \beta_m) \\ &= \sum_{i=0}^{m} \sum_{|\beta|=m} t_2(\beta_{i+1} \ldots \beta_m) \\ &= \sum_{i=0}^{m} \binom{m}{m-i} i! \sum_{|\gamma|=m-i} t_2(\gamma) \\ &= \sum_{i=0}^{m} \frac{m! x_{m-i}}{(m-i)!} \end{aligned} \tag{15}$$

So,

$$x_n = \sum_{i=0}^{n-1} \binom{n-1}{i} x_i y_{n-i-1} \tag{16}$$

Equation (15) can be rewritten as

$$\frac{y_m}{m!} = \sum_{i=0}^{m} \frac{x_i}{i!} \tag{17}$$

Using (17) we see that

$$\frac{y_n}{n!} - \frac{y_{n-1}}{(n-1)!} = \frac{x_n}{n!}$$

and thus

$$\sum_1^\infty \frac{y_n t^n}{n!} - t \sum_1^\infty \frac{y_{n-1} t^{n-1}}{(n-1)!} = \sum_1^\infty \frac{x_n t^n}{n!}$$

We now introduce a second generating function for clarity.

$$V(t) = \sum \frac{y_n t^n}{n!}$$

With this we can then write the above as

$$V(t) - y_0 - tV(t) = U(t) - x_0$$

which, since $x_0 = y_0 = 1$, means

$$V(t) = \frac{U(t)}{(1-t)}$$

The recurrence for x_n in (16) can be re-expressed as

$$\begin{aligned} \frac{x_n}{n!} &= \frac{1}{n!}\sum_{i=0}^{n-1} \frac{(n-1)!}{(n-i-1)!i!} x_i y_{n-i-1} \\ &= \frac{1}{n}\sum_{i=0}^{n-1}\sum_{j=0}^{n-i-1} \frac{x_i x_j}{i!j!} \qquad (18) \end{aligned}$$

The left hand side of (18) is the coefficient of t^{n-1} in $U'(t)$ and the right hand side is the coefficient of t^{n-1} in $U(t)V(t)$. It follows that

$$\begin{aligned} U'(t) &= U(t)V(t) \\ &= \frac{U^2(t)}{1-t} \end{aligned}$$

which can be rewritten as

$$\frac{1}{U^2} dU = \frac{1}{1-t} dt$$

This equation is easily integrated, the constant of integration being determined from the condition $u_0 = 1$, and the proof is now complete. □

The generating function then allows us to find the asymptotic behaviour of the sequence.

Lemma 4.4

$$x_n = O\left(n!\left(\frac{e}{e-1}\right)^n\right)$$

Proof: The only singularity of the analytic function $U(t)$ is where $1+\log(1-t) = 0$, which is when $t = (e-1)/e$. Hence the radius of convergence of $\sum x_n t^n/n!$ is $(e-1)/e$. The radius of convergence, R, of a power series $\sum a_n x^n$ is given by

$$\frac{1}{R} = \lim_{n\to\infty} \sqrt[n]{|a_n|}$$

(See e.g. [Apo63], Theorem $13-21$) We therefore have

$$\lim_{n\to\infty} \sqrt[n]{\left|\frac{x_n}{n!}\right|} = \frac{e}{e-1}$$

and thus

$$\frac{x_n}{n!} = O\left(\left(\frac{e}{e-1}\right)^n\right)$$

□

The next corollary was pointed out by George Phillips and gives a somewhat simpler recurrence for x_n than occurs in the proof of Theorem 4.1. It also provides the basis for an $O(n^2)$ dynamic programming algorithm for finding x_n.

Corollary 4.5 *Let $z_n = x_n/n!$ then we have*

$$\begin{aligned} z_0 &= 1, \text{ and} \\ z_n &= \sum_{i=1}^{n} \frac{z_{n-i}}{i}, \text{ for } n > 0. \end{aligned}$$

Proof: First note that

$$1 + \log(1-t) = 1 - t - \frac{t^2}{2} - \frac{t^3}{3}\dots$$

Then the result follows from equating coefficients of t^n in

$$\left(\sum_{i=0}^{\infty} z_i t^i\right)\left(1 - t - \frac{t^2}{2} - \frac{t^3}{3}\dots\right) = 1$$

□

Lemma 4.6 *If the pair (σ, τ) is 2-allowable then the pair (τ^r, σ^r) is also 2-allowable.*

Proof: Since σ and τ have decompositions as given by Lemma 4.1 we have

$$\begin{aligned}\tau^r &= \epsilon^r n \beta^r \delta^r \\ \sigma^r &= \gamma^r \beta^r n \alpha^r\end{aligned}$$

By induction on the length of the permutations we may presume that (ϵ^r, γ^r) and (δ^r, α^r) are each 2-allowable and another application of Lemma 4.1 completes the proof. □

Corollary 4.3 gives a recursive method for computing $t_2(\sigma)$. If applied directly the execution time of the resulting method would be exponential in n. However by using the same dynamic programming technique used in [AB94] for the unbounded case the calculation can be carried out in $O(n^4)$ steps and, since $s_2(\tau) = t_2(\tau^r)$, by Corollary 4.6 we obtain

Corollary 4.7 *If σ and τ are permutations of length n then both $t_2(\sigma)$ and $s_2(\tau)$ can be computed in time $O(n^4)$*

For a priority queue P_k the language, $L(P_k)$, associated with it can be thought of as an allowability relation. With this view we consider the composition of bounded priority queues. The results generally involve the composition of relations: given two relations A, B the relation AB is the set of pairs $\{(x,y) | \text{there exists } z \text{ with } (x,z) \in A \text{ and } (z,y) \in B\}$. The following results are all related to the serial composition of priority queues. We have no corresponding results for parallel composition. We also work under the assumption that all the priority queues in the systems being considered have capacity 2 or more. Any of capacity 1 can be removed without altering the power of the system.

The next lemma refers to the *weak order relation*, W_n. An ordered pair, (σ, τ), of permutations on n symbols are in W_n if and only if every pair of symbols of σ which are in increasing order are also in increasing order in τ. The weak order relation is an important tool for the study of geometric and combinatorial properties of the symmetric group and it is discussed at length in [Bjö83].

Lemma 4.8

$$L_n(P_i)^{n-2} \neq L_n(P_i)^{n-1} = L_n(P_i)^*, \text{ for all } i \geq 2,$$
$$L_n(P_i)^* = L_n(P_j)^* = W_n, \text{ for all } i, j \geq 2.$$

*where * denotes the transitive closure of a relation.*

Proof: The proof which follows is given in [AB94] to show that $L_n(P_\infty)^{n-2} \neq L_n(P_\infty)^{n-1} = L_n(P_\infty)^* = W_n$ and, as noted in its original form, it is valid for any bounded priority queue of capacity greater than 1.

For $(\sigma, \tau) \in W_n$ let $\rho(\sigma, \tau)$ be the smallest integer i such that all but the left most i symbols of σ and τ agree. Observe that ρ is never equal to 1. In order to show that $L_n(P_i)^{n-1} = W_n$ it suffices to show that for any $(\sigma, \tau) \in W_n$ with $\sigma \neq \tau$, there exists a σ' such that the following hold:

- $(\sigma, \sigma') \in L_n(P_i)$
- $(\sigma', \tau) \in W_n$
- $\rho(\sigma', \tau) < \rho(\sigma, \tau)$

To see this, write σ as $\alpha x \beta \gamma$ and τ as $\delta x \gamma$ such that $|\gamma| = n - \rho(\sigma, \tau)$. Since $(\sigma, \tau) \in W_n$, it follows that x must be larger than all symbols in β. Therefore, $\sigma' = \alpha \beta x \gamma$ satisfies the first of the three conditions above. The second condition is also satisfied by σ' since any pair of symbols in σ' appear either in the same order as in σ, or the same order as in τ. Finally, since σ' and τ both end with $x\gamma$, the third condition is satisfied. Since ρ cannot take the value 1, the sequence $\sigma', \sigma'', \ldots$ must reach τ in at most $n-1$ steps, proving $L_n(P_i)^{n-1} = W_n$.

To complete the proof, we show that the pair $((n, n-1, \ldots, 1), (1, n, n-1, \ldots, 2))$, which clearly lies in W_n, does not belong to $L_n(P_i)^{n-2}$. Suppose it were possible to transform $n, n-1, \ldots, 1$ into $1, n, n-1, \ldots, 2$ by a system of $n-2$ priority queues of capacity i in series. When the final element of the input has been placed in the first priority queue the other $n-1$ symbols must all be in the $n-2$ priority queues since none can be output yet. One of the priority queues must therefore contain two of the symbols $\{2, 3, \ldots, n\}$. This is clearly impossible since the two symbols it contains would then have to be output in increasing order. $\square$

Lemma 4.9

$$\text{If } i < j \text{ then } L(P_i) \subset L(P_j) \tag{19}$$

$$\text{For } i > 1, \text{ if } k < l \text{ then } L(P_i)^k \subset L(P_i)^l \tag{20}$$

$$L(P_{r+s-1}) \subset L(P_r)L(P_s) \tag{21}$$

Proof: In the proof of Lemma 3.2 we used the notion of simulation to show two systems were equivalent. We make use of that approach again here.

The inclusion in (19) is clear since P_j can simulate the operation of P_i and so any allowable pair of P_i is an allowable pair of P_j. Also the pair $((j, j-1, \ldots, 1), (1, 2, \ldots, j))$ is an allowable pair of P_j but not of P_i so the containment is strict.

To prove (20) we first note that P_i^l can clearly simulate P_i^k by making no use of the last $l - k$ priority queues in the system. Under the assumption that $i > 1$, strict containment is then given by the pair $((l, l-1, \ldots, 1), (1, l, l-1, \ldots, 2))$ which is an allowable pair of P_i^l but not of P_i^k.

Finally, for (21), we begin by showing $P_r P_s$ can simulate P_{r+s-1}. This argument uses the same method as the proof of Lemmas 3.2 and 3.4 for showing simulation is possible. A single storage location in the later of the two priority queues in series is reserved to allow data items from the earlier priority queue to be output. Consider an arbitrary allowable pair of P_{r+s-1} and the computation which transforms the input into the output. Whenever this computation inserts a data item into the priority queue we shall insert a data item into P_r. This is always possible although we may first have to move a data item from P_r to P_s. The total capacity of the priority queue P_{r+s-1} is $r + s - 1$ and so there is always at least one free location in $P_r P_s$, and by only moving data items into P_s when there is no free location in P_r we can guarantee that the free location is always in P_s. When a data item is removed from P_{r+s-1} we know that the same data item is in $P_r P_s$ somewhere and there is no smaller data item in the system. If the data item is in P_s we can simply delete it and continue with the simulation. If it is in P_r we can move it into the free location in P_s and then delete it. So we see that P_rP_s can simulate P_{r+s-1}. Finally, assuming $r, s > 1$, the pair $((3,2,1),(1,3,2))$ is an allowable pair of P_rP_s but not of P_{r+s-1} and so containment is strict. □

Lemma 4.10 *$L(P_{i_1})L(P_{i_2})\ldots L(P_{i_k}) = L(P_{j_1})L(P_{j_2})\ldots L(P_{j_l})$ if and only if $k = l$ and $i_a = j_a$ for $a = 1, 2, \ldots, k$*

Proof: Suppose $l > k$, then the pair $((l, l-1, \ldots, 1), (1, l, l-1, \ldots, 2))$ is an allowable pair of the right hand system but not the left so it must be the case that $l = k$.

Now suppose $i_a > j_a$ and let $s = k - 1 + i_a$, $p = k - a - 1$ then the following pair is an allowable pair of the left hand system but not of the right; $((s, s-1, \ldots, 1), (1, s, s-1, \ldots, s-p, s-p-i_a+1, s-p-i_a+2, \ldots, s-p-1, s-p-i_a, s-p-i_a-1, \ldots, 2))$.

This concludes the proof of the implication in one direction. It is clear that the opposite direction holds since the two conditions imply that the systems are identical. □

We now move from permutation inputs to inputs chosen from an arbitrary multiset. In this case we present a one to one correspondence between the pairs allowable by a capacity 2 priority queue and *multi segment trees*, described in Definition 4.1. We then count these trees to find a recurrence for the number of 2-allowable pairs of length n.

Definition 4.1 *$\mathcal{M}(n, m)$ is the set of all multi segment trees accounting for n data items chosen from $\{1, 2, \ldots, m\}$. A multi segment tree in $\mathcal{M}(n, m)$ satisfies the following:*

If $n = 0$ then the tree is empty, otherwise the root node consists of a maximum element $m' < m$ and r segments, for some $r \leq n$. Each segment contains a sequence, β_i, of symbols chosen from the alphabet $\{1, 2, \ldots, m' - 1\}$. The node has $r + 1$

children, as shown in Figure 17, each of which is a multi segment tree in $\mathcal{M}(n_i, m_i)$ *with* $m_i < m'$ *and* $\sum n_i + \sum |\beta_i| + r = n$.

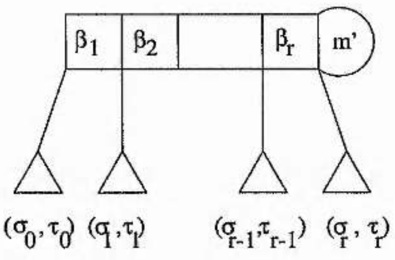


Figure 17: A Multi Segment Tree

Lemma 4.11 *For a priority queue of capacity* 2 *and input of length* n *chosen from the alphabet* $\{1, 2, \ldots, m\}$ *there is a one to one correspondence between the allowable pairs and the multi segment trees in* $\mathcal{M}(n, m)$.

Proof: Suppose (σ, τ) is a 2-allowable pair with symbols chosen from $\{1, 2, \ldots, m\}$. Then for some $m' \leq m$ it must have the form

$$(\sigma_0 \quad m'\beta_1\ \sigma_1 \quad m'\beta_2 \quad \ldots \quad m'\beta_r\ \sigma_r \quad , \quad \tau_0\ \beta_1 m' \quad \tau_1\ \beta_2 m' \quad \ldots \quad \beta_r m' \quad \tau_r)$$

with each (σ_i, τ_i) being 2-allowable and only containing data items in $\{1 \ldots m' - 1\}$.

We define the root node of a multi segment tree from this structure as follows. The sequences $\beta_1, \beta_2, \ldots, \beta_r$ are stored, in order, in the segments of the node and the node also contains the maximum value m'. The i^{th} child of this node corresponds to $(\sigma_{i-1}, \tau_{i-1})$ which is formed from the alphabet $\{1, 2, \ldots, m' - 1\}$ and has length $n_{i-1} \geq 0$. It follows inductively that there is a multi segment tree corresponding to each (σ_i, τ_i) and it is in $\mathcal{M}(n_{i-1}, m' - 1)$. Clearly we have $\sum n_i + \sum |\beta_i| + r = n$. Therefore we have indeed generated a multi segment tree in $\mathcal{M}(n, m)$.

The construction is quite clearly reversible since the β_i sequences are completely defined, the maximum element, which lies between each of those sequences, is given and the (σ_i, τ_i) pairs are all completely defined by the subtrees. The correspondence is therefore bijective. $\square$

From Lemma 4.11 it is clear that if we can count the number of trees in $\mathcal{M}(n,m)$ then we know how many 2-allowable pairs there are of length n over the alphabet $\{1,2,\ldots,m\}$ and so we have

Theorem 4.2 *The number of allowable pairs of length n for a priority queue of capacity 2 and input alphabet of size m is given by*

$$\begin{aligned} x(n,1) &= x(0,m) = 1, \\ x(n,m) &= \sum_{r=1}^{n}\sum_{i=r}^{n}\sum_{l=2}^{m}(l-1)^{(i-r)}\binom{i-1}{r-1}y(n-i,l-1,r+1). \end{aligned}$$

where

$$\begin{aligned} y(n,m,1) &= x(n,m), \\ y(n,m,r) &= \sum_{i=0}^{n}\left(x(i,m)y(n-i,m,r-1)\right). \end{aligned}$$

Proof: The base cases are trivial; $x(n,1)=1$ because the only tree in $\mathcal{M}(n,1)$ is the single node tree with n empty β sequences and 1 as the maximum element. Similarly $x(0,m)=1$ because the only tree in $\mathcal{M}(0,m)$ is the empty tree.

The general case is as follows. We shall count how many ways we can construct the root node of a tree in $\mathcal{M}(n,m)$ and how many ways the subtrees can be arranged from each such root. The root node can have between 1 and n sequences in it, we shall assume it contains r sequences. With r fixed the root node can account for between r and n of the n elements in the entire tree, we shall assume it accounts for i of them. The maximum element can take any value between 2 and m and we shall assume it takes value l. With all these values fixed we see that the sequence of elements in the root node can be chosen in $(l-1)^{i-r}$ ways and this sequence can be split into the r sequences in $\binom{i-1}{r-1}$ ways. The root has $r+1$ children, some of which may be empty, and between them they account for $n-i$ elements chosen from the alphabet $\{1,2,\ldots,l-1\}$. Suppose there are $y(n-i,l-1,r+1)$ ways to choose children which satisfy this, then we see that

$$x(n,m) = \sum_{r=1}^{n}\sum_{i=r}^{n}\sum_{l=2}^{m}(l-1)^{(i-r)}\binom{i-1}{r-1}y(n-i,l-1,r+1)$$

It remains to count the number of ways we can create the $r+1$ children using $n-i$ elements over $\{1,2,\ldots,l-1\}$. The base case is again easy; if there is only one child then we have

$$y(n-i, l-1, 1) = x(n-i, l-1)$$

For the general case, note that the first of the $r+1$ trees can account for $0 \leq j \leq n-i$ elements, and the tree can take $x(j, l-1)$ different forms. The remaining r trees can then be constructed in $y(n-i-j, l-1, r)$ different ways giving us

$$y(n, m, r) = \sum_{j=0}^{n} x(j, m) y(n-j, m, r-1)$$

□

4.2 Permutation Inputs with Binary Priorities

Up to this point in the chapter we have worked under the assumption that the priority of a data item, i, is simply its value. In general it is more likely that the data items being processed by a priority queue will consist of a data part and a separate priority. This allows a much more complicated situation where data items can share the same priority, as in the binary case, but they remain distinguishable.

To model this we consider three sequences rather than a pair, although we shall shortly see this is not entirely necessary. The first two sequences are the input, σ, and the output, τ. These are formed from distinct data items $1, 2, \ldots, n$. The third sequence, π, is the *priority assignment* and contains the priorities of the data items in σ. The interpretation of the priority assignment is that σ_i has priority π_i.

There are various quantities we could investigate with this model. We shall focus on the set of pairs which are allowable for at least one priority assignment. That is, the set $\{(\sigma, \tau, \pi) | (\sigma, \tau)$ is allowable for some priority assignment $\pi\}$.

Since the values of the data items in σ are completely independent of their behaviour in the priority queue we assume $\sigma = 1, 2, \ldots, n$. The total number of "allowable tuples" will then be $n!$ times the number found with this restriction. We must also introduce some relatively severe restrictions on the form of π to make progress. First we insist that the elements of π are chosen from the binary alphabet so all data items in σ either have priority 1 or priority 0. Then we look at the special cases when there are only 1, 2 or 3 data items of priority 0 in σ. In these special cases we shall consider the quantity $x_{n,z}$ which is the number of allowable tuples of length n containing z data items with priority 0.

The labels $1, 2, \ldots, n$ serve only one purpose in this situation. That is to allow us to distinguish between different data items which have the same priority. It is therefore sensible to change the notation to emphasise the attribute of each data item which is of greatest significance: the priority. Rather than consider the i^{th} symbol of the input to be a data item with value σ_i and priority π_i we shall consider it to be the symbol π_i with a unique subscript. As an example the tuple $((1,2,3),(3,2,1),(1,0,0))$ would be represented as $((1_1, 0_1, 0_2), (0_2, 0_1, 1_1))$. Further, as we shall see in the analysis which follows, it is not necessary to keep a close track of the data items of value 1. We therefore only require unique subscripts on the 0's.

Suppose there is only one data item which has priority 0 in the input sequence. Then the distribution of priorities in (σ, τ) will be $(1^{i-1}0_1 1^{n-i}, 1^a 0_1 1^{n-a-1})$, where $0 \leq a < i \leq n$. It is clear from this that we can write down an expression for the number of such pairs.

$$\begin{aligned} x_{n,1} &= \sum_{i=1}^{n}\sum_{a=0}^{i-1}\binom{i-1}{a}a!(n-a-1)! \\ &= \sum_{i=1}^{n}\sum_{a=0}^{i-1}\frac{(i-1)!a!(n-a-1)!(n-i)!}{a!(i-a-1)!(n-i)!} \\ &= \sum_{i=1}^{n}(i-1)!(n-i)!\sum_{a=0}^{i-1}\binom{n-a-1}{i-a-1} \\ &= \sum_{i=1}^{n}(i-1)!(n-i)!\binom{n}{i-1} \\ &= n!\sum_{i=1}^{n}\frac{1}{n-i+1} \\ &= n!\sum_{i=1}^{n}\frac{1}{i} \end{aligned} \tag{22}$$

The third step in this derivation uses the identity

$$\sum_{k=0}^{n}\binom{r+k}{k} = \binom{r+n+1}{n}$$

which is given in [GK82]. We note this identity since it is used a great many times in the following analysis.

From the final expression for $x_{n,1}$ we obtain a recurrence.

Lemma 4.12

$$\begin{aligned} x_{1,1} &= 1, \\ x_{n,1} &= nx_{n-1,1} + (n-1)!. \end{aligned}$$

Proof: We prove this by directly substituting (22) into the recurrence.

It is clear that in the summation $x_{1,1} = 1$ and in the general case we have

$$\begin{aligned} x_{n,1} &= n! \sum_{i=1}^{n} \frac{1}{i} \\ &= n! \sum_{i=1}^{n-1} \frac{1}{i} + \frac{n!}{n} \\ &= nx_{n-1,1} + (n-1)! \end{aligned}$$

as required. □

It is interesting to note that the numbers generated by this recurrence are the *Stirling numbers of the second kind, $S(n,k)$*, with the second parameter equal to 2. $S(n,k)$ is the number of ways of partitioning a set $\{1, 2, \ldots, n\}$ into k non-empty sets, see e.g. [Bry93]. The final thing to note in this case comes immediately from (22).

Corollary 4.13 $x_{n,1} = \Theta(n! \log n)$

When there are two data items of priority 0 we can take the same approach as before. The input sequence has the priorities distributed over it as $1^{i-1}0_1 1^{j-i-1} 0_2 1^{n-j}$ and the output sequence can have the priorities distributed in one of two ways; either as $1^a 0_1 1^b 0_2 1^{n-a-b-2}$ or as $1^a 0_2 0_1 1^{n-a-2}$. We can write down a summation to count all the allowable pairs of this form as follows. It is clear that i varies between 1 and $n-1$, j varies between $i+1$ and n, a varies between 0 and $i-1$ and b varies between 0 and $j-a-2$. For any fixed values i, j, a and b the first a elements of τ are chosen from the first $i-1$ of σ and can appear in any order so we can have $\binom{i-1}{a}a!$ different initial segments of τ. If $b > 0$ then the elements in the middle segment can be chosen from $j-a-2$ elements and can appear in any order so we have $\binom{j-a-2}{b}b!$

middle segments. Then, the remaining elements can appear in any order in the final segment giving a total of $(n-a-b-2)!$ different final segments. If $b=0$ then we have no middle segment and the final segment contains $n-a-2$ elements but the two data items of priority 0 can appear in either order in the output. This leads us to the first summation expression for $x_{n,2}$

$$\begin{aligned} x_{n,2} &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{a=0}^{i-1} \left(\sum_{b=1}^{j-a-2} \binom{i-1}{a} a! \binom{j-a-2}{b} b!(n-a-b-2)! \right. \\ &\qquad \left. + 2\binom{i-1}{a} a!(n-a-2)! \right) \end{aligned}$$

We can substantially rewrite this formula and eliminate two of the summations within it.

$$\begin{aligned} x_{n,2} &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{a=0}^{i-1} \binom{i-1}{a} a! \left(\sum_{b=0}^{j-a-2} (j-a-2)!(n-j)! \binom{n-a-b-2}{j-a-b-2} + (n-a-2)! \right) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{a=0}^{i-1} \binom{i-1}{a} a! \left((j-a-2)!(n-j)! \binom{n-a-1}{j-a-2} + (n-a-2)! \right) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{a=0}^{i-1} \binom{i-1}{a} a! \left(\frac{(n-a-1)!}{n-j+1} + (n-a-2)! \right) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(\sum_{a=0}^{i-1} \binom{i-1}{a} a! \frac{(n-a-1)!}{n-j+1} + \sum_{a=0}^{i-1} \binom{i-1}{a} a!(n-a-2)! \right) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(\frac{(i-1)!(n-i)!}{n-j+1} \sum_{a=0}^{i-1} \binom{n-a-1}{i-a-1} + (i-1)!(n-i-1)! \sum_{a=0}^{i-1} \binom{n-a-2}{i-a-1} \right) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(\frac{(i-1)!(n-i)!}{n-j+1} \binom{n}{i-1} + (i-1)!(n-i-1)! \binom{n-1}{i-1} \right) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(\frac{n!}{(n-i+1)(n-j+1)} + \frac{(n-1)!}{n-i} \right) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} n! \left(\frac{1}{(n-i+1)(n-j+1)} + \frac{1}{n(n-i)} \right) \\ &= n! \sum_{i=2}^{n} \sum_{j=1}^{i-1} \left(\frac{1}{ij} + \frac{1}{n(i-1)} \right) \\ &= n! \left(\sum_{i=2}^{n} \sum_{j=1}^{i-1} \frac{1}{n(i-1)} + \sum_{i=2}^{n} \sum_{j=1}^{i-1} \frac{1}{ij} \right) \\ &= n! \left(\frac{n-1}{n} + \sum_{i=2}^{n} \sum_{j=1}^{i-1} \frac{1}{ij} \right) \end{aligned}$$

$$= n!\left(1-\frac{1}{n}+\sum_{i=2}^{n}\sum_{j=1}^{i-1}\frac{1}{ij}\right) \tag{23}$$

As before, from this expression we obtain a recurrence

Lemma 4.14

$$\begin{aligned} x_{2,2} &= 2, \\ x_{n,2} &= nx_{n-1,2}+x_{n-1,1}+(n-2)!. \end{aligned}$$

Proof: We prove this by direct substitution of the expressions for $x_{n,2}$ and $x_{n,1}$ into the recurrence.

$$\begin{aligned} nx_{n-1,2}+x_{n-1,1}+(n-2)! &= n\left((n-1)!\left(1-\frac{1}{n-1}+\sum_{i=2}^{n-1}\sum_{j=1}^{i-1}\frac{1}{ij}\right)\right) \\ &\quad +(n-1)!\sum_{i=1}^{n-1}\frac{1}{i}+(n-2)! \\ &= n!\left(1-\frac{1}{n-1}+\sum_{i=2}^{n-1}\sum_{j=1}^{i-1}\frac{1}{ij}+\sum_{i=1}^{n-1}\frac{1}{ni}+\frac{1}{n(n-1)}\right) \\ &= n!\left(1-\frac{1}{n}+\sum_{i=2}^{n}\sum_{j=1}^{i-1}\frac{1}{ij}\right) \end{aligned}$$

□

Once again, from (23) we can find the asymptotic behaviour of the sequence $(x_{n,2})$.

Corollary 4.15

$$x_{n,2}=\Theta\left(n!\log n\log n\right)$$

Proof: We begin by noting that in

$$n!\left(1-\frac{1}{n}+\sum_{i=2}^{n}\sum_{j=1}^{i-1}\frac{1}{ij}\right)$$

the dominant expression is the double summation. We then write

$$\sum_{1\le j<i\le n}\frac{n!}{ij}$$

$$
\begin{aligned}
&= \frac{n!}{2} \sum_{1 \le j \ne i \le n} \frac{1}{ij} \\
&= \frac{n!}{2} \left(\sum_{1 \le i,j \le n} \frac{1}{ij} - \sum_{i=1}^{n} \frac{1}{i^2} \right) \\
&= \frac{n!}{2} \left(\sum_{1 \le i,j \le n} \frac{1}{ij} - C_n \right) \\
&= \frac{n!}{2} \left(\left(\sum_{i=1}^{n} \frac{1}{i} \right)^2 - C_n \right) \\
&= \Theta(n! \log n \log n)
\end{aligned}
$$

since

$$0 \le C_n \le \frac{\pi^2}{6}$$

□

When we allow three data items in the input sequence to have priority 0 we can still apply the same technique as before although the results are significantly more complex. The priorities are distributed over the input sequence in the form $1^{i-1}0_1 1^{j-i-1}0_2$ $1^{k-j-1}0_3 1^{n-k}$ and over the output sequence they can have any of the following six distributions:

$$
\begin{array}{lll}
1^a 0_1 1^b 0_2 1^c 0_3 1^{n-a-b-c-3} & 1^a 0_1 1^b 0_3 0_2 1^{n-a-b-3} & 1^a 0_2 0_1 1^c 0_3 1^{n-a-c-3} \\
1^a 0_2 0_3 0_1 1^{n-a-3} & 1^a 0_3 0_1 0_2 1^{n-a-3} & 1^a 0_3 0_2 0_1 1^{n-a-3}
\end{array}
$$

In the input sequence we clearly have $0 < i < j < k \le n$. We also have $0 \le a \le i-1$, $0 \le b \le j-a-2$ and $0 \le c \le k-a-b-3$ in the output sequence. For fixed i, j, k and a there are some special cases; if $b \ne 0, c = 0$ then the last two data items of priority 0 can occur in either order, if $b = 0, c \ne 0$ then the first two data items of priority 0 can occur in either order and if $b = c = 0$ then the three data items of priority 0 can occur in any of 6 different orders. This gives us the following, with some annotation for clarity.

$$
\begin{aligned}
x_{n,3} = & \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} \\
& \sum_{a=0}^{i-1} \binom{i-1}{a} a! \left[\sum_{b=1}^{j-a-2} \binom{j-a-2}{b} b! \right.
\end{aligned}
$$

$$\left(\sum_{c=1}^{k-a-b-3}\binom{k-a-b-3}{c}c!(n-a-b-c-3)! +_{(c=0)} 2(n-a-b-3)!\right)$$
$$+_{(b=0)}2\sum_{c=1}^{k-a-3}\binom{k-a-3}{c}c!(n-a-c-3)! +_{(b,c=0)} 6(n-a-3)!\Bigg]$$

We can greatly simplify this as before and, for readability, we begin by looking at a small subexpression.

$$\begin{aligned}
& \sum_{c=1}^{k-a-b-3}\binom{k-a-b-3}{c}c!(n-a-b-c-3)! + (n-a-b-3)! \\
= & \sum_{c=0}^{k-a-b-3}\binom{k-a-b-3}{c}c!(n-a-b-c-3)! \\
= & \sum_{c=0}^{k-a-b-3}(k-a-b-3)!(n-k)!\binom{n-a-b-c-3}{k-a-b-c-3} \\
= & (k-a-b-3)!(n-k)!\binom{n-a-b-2}{k-a-b-3} \\
= & \frac{(n-a-b-2)!}{n-k+1}
\end{aligned}$$

We now move out one summation and continue rewriting

$$\begin{aligned}
& \sum_{b=1}^{j-a-2}\binom{j-a-2}{b}b!\left(\frac{(n-a-b-2)!}{n-k+1} + (n-a-b-3)!\right) + \\
& \frac{(n-a-2)!}{n-k+1} + (n-a-3)! \\
= & \sum_{b=0}^{j-a-2}\binom{j-a-2}{b}b!\left(\frac{(n-a-b-2)!}{n-k+1} + (n-a-b-3)!\right) \\
= & \frac{(j-a-2)!(n-j)!}{n-k+1}\sum_{b=0}^{j-a-2}\binom{n-a-b-2}{j-a-b-2} + \\
& (j-a-2)!(n-j-1)!\sum_{b=0}^{j-a-2}\binom{n-a-b-3}{j-a-b-2} \\
= & (j-a-2)!(n-j-1)!\left(\frac{n-j}{n-k+1}\binom{n-a-1}{j-a-2} + \binom{n-a-2}{j-a-2}\right) \\
= & (n-a-2)!\left(\frac{n-a-1}{(n-k+1)(n-j+1)} + \frac{1}{n-j}\right)
\end{aligned}$$

Moving out another summation we then have

$$\begin{aligned}
= & \sum_{a=0}^{i-1}\binom{i-1}{a}a!\left((n-a-2)!\left(\frac{n-a-1}{(n-k+1)(n-j+1)} + \frac{1}{n-j}\right)\right. \\
& \left. + \frac{(n-a-2)!}{n-k+1} + 3(n-a-3)!\right)
\end{aligned}$$

$$\begin{aligned}
&= \sum_{a=0}^{i-1} \binom{i-1}{a} a!(n-a-1)! \frac{1}{(n-k+1)(n-j+1)} \\
&+ \sum_{a=0}^{i-1} \binom{i-1}{a} a!(n-a-2)! \frac{1}{n-j} \\
&+ \sum_{a=0}^{i-1} \binom{i-1}{a} a!(n-a-2)! \frac{1}{n-k+1} \\
&+ \sum_{a=0}^{i-1} \binom{i-1}{a} a! 3(n-a-3)! \\
&= (i-1)! \left(\frac{(n-i)!}{(n-k+1)(n-j+1)} \binom{n}{i-1} + \frac{(n-i-1)!}{n-j} \binom{n-1}{i-1} \right. \\
&\qquad \left. + \frac{(n-i-1)!}{n-k+1} \binom{n-1}{i-1} + 3(n-i-2)! \binom{n-2}{i-1} \right) \\
&= n! \left(\frac{1}{(n-i+1)(n-j+1)(n-k+1)} + \frac{1}{n(n-i)(n-j)} \right. \\
&\qquad \left. + \frac{1}{n(n-i)(n-k+1)} + \frac{3}{n(n-1)(n-i-1)} \right)
\end{aligned}$$

The expression for $x_{n,3}$ is now a triple summation of four parts and, to proceed further, we consider the triple summation of each of these parts in turn. For the first part we simply change the summation indices

$$\begin{aligned}
& \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} \frac{n!}{(n-i+1)(n-j+1)(n-k+1)} \\
&= \sum_{i=3}^{n} \sum_{j=2}^{i-1} \sum_{k=1}^{j-1} \frac{n!}{ijk}
\end{aligned}$$

For the second we have

$$\begin{aligned}
& \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} \frac{n!}{n(n-i)(n-j)} \\
&= \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \frac{n!(n-j)}{n(n-i)(n-j)} \\
&= \sum_{i=1}^{n-2} \frac{n!(n-i-1)}{n(n-i)} \\
&= (n-1)! \sum_{i=1}^{n-2} \frac{n-i-1}{n-i} \qquad (24)
\end{aligned}$$

For the third part we write

$$\begin{aligned}
&\sum_{i=1}^{n-2}\sum_{j=i+1}^{n-1}\sum_{k=j+1}^{n}\frac{n!}{n(n-i)(n-k+1)} \\
=\ &(n-1)!\sum_{i=1}^{n-2}\frac{1}{n-i}\left(\sum_{j=i+1}^{n-1}\sum_{k=j+1}^{n}\frac{1}{n-k+1}\right) \\
=\ &(n-1)!\sum_{i=1}^{n-2}\frac{1}{n-i}\left(\sum_{k=1}^{n-i-1}\frac{n-i-k}{k}\right) \\
=\ &(n-1)!\sum_{i=1}^{n-2}\sum_{k=1}^{n-i-1}\left(\frac{1}{k}-\frac{1}{n-i}\right) \\
=\ &(n-1)!\sum_{i=1}^{n-2}\left(\frac{(n-i-1)}{i}-\frac{(n-i-1)}{n-i}\right)
\end{aligned}$$

We could simplify this further but notice that (24) is the same as part of the above expression. Finally, for the fourth part, we have

$$\begin{aligned}
&\sum_{i=1}^{n-2}\sum_{j=i+1}^{n-1}\sum_{k=j+1}^{n}\frac{3n!}{n(n-1)(n-i-1)} \\
=\ &\sum_{i=1}^{n-2}\sum_{j=i+1}^{n-1}\frac{3n!(n-j)}{n(n-1)(n-i-1)} \\
=\ &\sum_{i=1}^{n-2}\frac{3n!}{n(n-1)(n-i-1)}\sum_{j=i+1}^{n-1}(n-j) \\
=\ &\sum_{i=1}^{n-2}\frac{3n!}{n(n-1)(n-i-1)}\frac{1}{2}(n-i-1)(n-i) \\
=\ &\frac{3}{2}(n-2)!\sum_{i=1}^{n-2}(n-i) \\
=\ &\frac{3}{4}(n-2)!(n+1)(n-2)
\end{aligned}$$

Combining these back into one expression we obtain

$$\begin{aligned}
x_{n,3} &= \sum_{i=3}^{n}\sum_{j=2}^{i-1}\sum_{k=1}^{j-1}\frac{n!}{ijk} \\
&+ (n-1)!\sum_{i=1}^{n-2}\frac{n-i-1}{i} \\
&+ \frac{3}{4}(n-2)!(n+1)(n-2)
\end{aligned}$$

Once again, from this expression we can find a recurrence

Lemma 4.16

$$\begin{aligned} x_{3,3} &= 6, \\ x_{n,3} &= n x_{n-1,3} + x_{n-1,2} + x_{n-2,1} + 3(n-3)!. \end{aligned}$$

Proof: The base case is clear and the general case is proved by direct substitution.

$$\begin{aligned}
& n x_{n-1,3} + x_{n-1,2} + x_{n-2,1} + 3(n-3)! \\
= \; & \sum_{i=3}^{n-1}\sum_{j=2}^{i-1}\sum_{k=1}^{j-1} \frac{n(n-1)!}{ijk} \\
& + n(n-2)! \sum_{i=1}^{n-3} \frac{n-i-2}{i} \\
& + \frac{3}{4}(n-3)!(n-3)n^2 \\
& + (n-1)! \left(1 - \frac{1}{n-1} + \sum_{j=2}^{n-1}\sum_{k=1}^{j-1} \frac{1}{jk}\right) \\
& + (n-2)! \sum_{i=1}^{n-2} \frac{1}{i} \\
& + 3(n-3)! \\
= \; & \left\{ \sum_{i=3}^{n-1}\sum_{j=2}^{i-1}\sum_{k=1}^{j-1} \frac{n(n-1)!}{ijk} + (n-1)! \sum_{j=2}^{n-1}\sum_{k=1}^{j-1} \frac{1}{jk} \right\} \\
& + \left\{ n(n-2)! \sum_{i=1}^{n-3} \frac{n-i-2}{i} + (n-2)!(n-2) + (n-2)! \sum_{i=1}^{n-2} \frac{1}{i} \right\} \\
& + \left\{ \frac{3}{4}(n-3)!(n-3)n^2 + 3(n-3)! \right\} \\
= \; & \sum_{i=3}^{n}\sum_{j=2}^{i-1}\sum_{k=1}^{j-1} \frac{n!}{ijk} \\
& + n(n-2)! \left\{ \sum_{i=1}^{n-3} \frac{n-i-2}{i} + \sum_{i=1}^{n-2} \frac{1}{ni} + \frac{n-2}{n} \right\} \\
& + \frac{3}{4}(n-3)! \left(n^3 - 3n^2 + 4\right) \\
= \; & \sum_{i=3}^{n}\sum_{j=2}^{i-1}\sum_{k=1}^{j-1} \frac{n!}{ijk} \\
& + (n-1)! \left\{ \sum_{i=1}^{n-3} \frac{n(n-i-2)}{i(n-1)} + \sum_{i=1}^{n-2} \frac{1}{i(n-1)} + \frac{n-2}{n-1} \right\} \\
& + \frac{3}{4}(n-3)!(n-2)(n^2-n-2) \\
= \; & \sum_{i=3}^{n}\sum_{j=2}^{i-1}\sum_{k=1}^{j-1} \frac{n!}{ijk}
\end{aligned}$$

$$
\begin{aligned}
& +(n-1)!\left\{\sum_{i=1}^{n-2}\frac{n(n-i-2)+1}{i(n-1)}+\sum_{i=1}^{n-2}\frac{i}{i(n-1)}\right\} \\
& +\frac{3}{4}(n-3)!(n-2)(n^2-n-2) \\
= & \sum_{i=3}^{n}\sum_{j=2}^{i-1}\sum_{k=1}^{j-1}\frac{n!}{ijk} \\
& +(n-1)!\sum_{i=1}^{n-2}\frac{n^2-ni-2n+1+i}{i(n-1)} \\
& +\frac{3}{4}(n-3)!(n-2)(n^2-n-2) \\
= & \sum_{i=3}^{n}\sum_{j=2}^{i-1}\sum_{k=1}^{j-1}\frac{n!}{ijk} \\
& +(n-1)!\sum_{i=1}^{n-2}\frac{n-i-1}{i} \\
& +\frac{3}{4}(n-3)!(n-2)(n^2-n-2)
\end{aligned}
$$

as required. □

The expression for $x_{n,3}$ also allows us to deduce its asymptotic behaviour.

Corollary 4.17 $x_{n,3} = \Theta\left(n! \log n \log n \log n\right)$

Proof: The dominant term in $x_{n,3}$ is the triple summation for which we have

$$
\begin{aligned}
& \sum_{1\le k<j<i\le n}\frac{n!}{ijk} \\
= & \frac{n!}{6}\sum_{1\le k\ne j\ne i\le n}\frac{1}{ijk} \\
= & \frac{n!}{6}\left(\sum_{1\le i,j,k\le n}\frac{1}{ijk}-3\sum_{1\le i,j\le n}\frac{1}{i^2 j}+2\sum_{i=1}^{n}\frac{1}{i^3}\right) \\
= & \frac{n!}{6}\left(\left(\sum_{i=1}^{n}\frac{1}{i}\right)^3-3\sum_{1\le i,j\le n}\frac{1}{i^2 j}+2C_n\right) \\
= & \Theta(n!\log n\log n\log n - n!\log n) \\
= & \Theta(n!\log n\log n\log n)
\end{aligned}
$$

since

$$0 \le C_n \le \zeta(3)$$

where ζ is *the Riemann zeta function* ([THB86]) and $\zeta(3) \sim 1.2$. □

We can continue this style of analysis for larger numbers of zeros and the only difficulty which occurs is the exponential increase in formula complexity as we go on. The initial summation expressions, final recurrences and asymptotic behaviour for each case exhibit a very definite pattern. The next few recurrence equations are conjectured to be

$$\begin{aligned} x_{4,4} &= 24, \\ x_{n,4} &= nx_{n-1,4} + x_{n-1,3} + x_{n-2,2} + 3x_{n-3,1} + 13(n-4)!. \end{aligned}$$

$$\begin{aligned} x_{5,5} &= 120, \\ x_{n,5} &= nx_{n-1,5} + x_{n-1,4} + x_{n-2,3} + 3x_{n-3,2} + 13x_{n-4,1} + 71(n-5)!. \end{aligned}$$

$$\begin{aligned} x_{6,6} &= 720, \\ x_{n,6} &= nx_{n-1,6} + x_{n-1,5} + x_{n-2,4} + 3x_{n-3,3} + 13x_{n-4,2} + 71x_{n-5,1} + 461(n-6)!. \end{aligned}$$

$$\begin{aligned} x_{7,7} &= 5040, \\ x_{n,7} &= nx_{n-1,7} + x_{n-1,6} + x_{n-2,5} + 3x_{n-3,4} + 13x_{n-4,3} + 71x_{n-5,2} + 461x_{n-6,1} \\ &\quad +3447(n-7)!. \end{aligned}$$

$$\begin{aligned} x_{8,8} &= 40320, \\ x_{n,8} &= nx_{n-1,8} + x_{n-1,7} + x_{n-2,6} + 3x_{n-3,5} + 13x_{n-4,4} + 71x_{n-5,3} + 461x_{n-6,2} \\ &\quad +3447x_{n-7,1} + 29093(n-8)!. \end{aligned}$$

and it is conjectured that in general

Conjecture 1

$$x_{z,z} = x_{z,0} = z!,$$

$$x_{n,z} = nx_{n-1,z} + \sum_{i=1}^{z} a_i x_{n-i,z-i}.$$

where

$$\begin{aligned} a_1 &= 1, \\ a_m &= m! - \sum_{i=1}^{m-1} i!a_{m-i}. \end{aligned}$$

□

We have a second conjecture related to this problem. Below we present a method for constructing a mathematical expression dependent on z and n. It is conjectured that the expression constructed is a closed formula for $x_{n,z}$.

Construction: We describe how to construct a mathematical expression $w_{n,z}$. The basic form of the expression is a series of z nested summations with indices $p_1, p_2, \ldots, p_z$. The summand is an expression $X(p_1, p_2, \ldots, p_z)$. We therefore have

$$w_{n,z} = \sum_{p_1=1}^{n-z} \sum_{p_2=p_1+1}^{n-z+1} \cdots \sum_{p_z=p_{z-1}+1}^{n} X(p_1, p_2, \ldots, p_z)$$

The summand consists of 2^{z-1} separate terms which can be gathered into z groups. The g^{th} such group contains $\binom{z-1}{g-1}$ terms each of which is a fraction which has $(n-g+1)!$ as a factor of its numerator. If we let r_g be the number of terms in the g^{th} group then we write the group as

$$(n-g+1)!\left(\frac{n_1}{d_1} + \ldots + \frac{n_{r_g}}{d_{r_g}}\right)$$

The denominator of each fraction is a product of $z-g+1$ factors each corresponding to one of $p_1, p_2, \ldots p_z$. Each denominator must have the factor corresponding to p_1 in it; there are r_g ways to choose the other p_i's and thus r_g fractions in the group. For any fixed sequence of p_i's, say $p_1 = p_{i_1}, p_{i_2}, \ldots, p_{i_{z-g+1}}$, we define q_i to be the number of elements in $p_1, p_2, \ldots p_i$ which are not in the chosen sequence. The denominator of the fraction is then

$$(n - p_1 + 1 + q_1 - g)(n - p_{i_2} + 1 + q_2 - g) \ldots (n - p_{i_{z-g+1}} + 1 + q_{z-g+1} - g)$$

The numerator of each fraction is also dependent on the sequence of p_i's it corresponds to. In the sequence there are generally several continuous blocks of p_i's missing. We define $l_1, l_2, \ldots, l_m$ to be the lengths of these blocks. The numerator is then

$$\prod_{i-1}^{m} a_{l_i}$$

where the sequence a_n is defined in Conjecture 1.

This completes the description of each fraction, the summand $X(p_1, p_2, \ldots, p_z)$ and thus the entire expression $w_{n,z}$. □

Conjecture 2 *The expression $w_{n,z}$ described in the above construction evaluates to $x_{n,z}$, the number of allowable pairs, with binary priorities, of length n with z data items having priority* 0.

Although the construction is reasonably complex there is some very convincing numerical data to support the conjecture. Table 18 contains the number of allowable pairs for fixed n and z as computed using the conjectured expressions and as found by direct enumeration.

$n\rightarrow$ $z\downarrow$	10	9	8	7	6	5	4	3	2	1
1	10628640	1026576	109584	13068	1764	274	50	11	3	1
2	16019496	1495260	153404	17452	2224	321	53	10	2	
3	17043884	1542392	152640	16633	2009	270	40	6		
4	14886110	1314175	126359	13296	1534	192	24			
5	11783123	1023250	96480	9892	1092	120				
6	9056938	779800	72612	7248	720					
7	7048888	604008	55296	5040						
8	5646744	478080	40320							
9	4625280	362880								
10	3628800									

Figure 18: Numerical Evidence for Conjectures 1 and 2

A consequence of Conjecture 2 is:

Conjecture 3

$$x_{n,z} = \Theta(n!(\log n)^z)$$

4.3 Binary Inputs

We now turn our attention to binary input sequences and a bounded capacity priority queue. Here we give some results analogous to those in [Atk93]. We extend the *partial sum criterion* of that paper to bounded capacity priority queues. Using this we investigate the case $k = 2$ and find a rather unexpected recurrence for the number of allowable pairs. We then go on to give a one to one correspondence between k-allowable binary pairs and ordered forests. Work in [dBKR72] then gives a closed formula for the number of such forests for fixed k and n. We continue by giving results relating to the composition and closure of the allowability relations of bounded capacity priority queues. This is followed by proofs of the existence of quadratic time algorithms for finding $s(\tau)$ and $t(\sigma)$. Finally we show that one of the symmetry results, mentioned at the start of this chapter, holds in the bounded capacity case.

When a priority queue is producing τ from some σ we can assume the priority queue never contains more than one data item of priority 0. This is clear since the notion of standard computation in Definition 2.3 is valid here. This definition combined with the fact that 0 has a higher priority than 1 tells us that a 0 must be output from the priority queue immediately after being inserted. We assume throughout this section that all computations are standard.

We begin by extending the partial sum criterion of [Atk93] to characterise k-allowable pairs. Any pair of binary sequences (σ, τ) can be written as

$$(1^{s_0}01^{s_1}\ldots01^{s_r}, 1^{t_0}01^{t_1}\ldots01^{t_r})$$

With this notation the *extended partial sum criterion* is

Lemma 4.18 *A pair (σ, τ) is k-allowable if and only if*

$$0 \leq \sum_{i=0}^{j} s_i - t_i \leq k-1, \quad 0 \leq j \leq r$$

with equality on the left when $j = r$.

Proof: Suppose (σ, τ) fails to satisfy the criterion in some way. There are two ways this can occur, either we do not have equality when $j = r$ or there exists a j for which $\sum_{i=0}^{j} s_i - t_i < 0$ or $\sum_{i=0}^{j} s_i - t_i \geq k$.

In the first case is is clear that, if we do not have equality when $j = r$, then τ is not a reordering of σ and the pair cannot be allowable.

For the second case, suppose j is the least value at which the pair fails the criterion. If $\sum_{i=0}^{j} s_i - t_i < 0$ then, at the point where the $j + 1^{th}$ 0 is about to be input and output the total number of 1's output exceeds the total number input. This clearly is not possible and so the pair is not allowable. The final case is if $\sum_{i=0}^{j} s_i - t_i \geq k$. At the point where the $j + 1^{th}$ 0 is about to be input and output the number of 1's which must be stored is the difference between the total number which precede the 0 in σ and the total number which precede the 0 in τ. The space required to store these, plus one for the 0 which must pass through, is $1 + \sum_{i=0}^{j} s_i - t_i$. This value exceeds the capacity of the priority queue and thus the pair is not allowable. This concludes the proof of necessity.

It remains to show the criterion is sufficient and we shall do this by showing how to construct τ from σ given that (σ, τ) satisfies the criterion. Suppose we are carrying out the computation transforming σ into τ and that we have just output the j^{th} 0. We know there are $a = \sum_{i=0}^{j} s_i - t_i$ 1's currently stored in the priority queue, there are a further s_{j+1} to be read in and a further t_{j+1} to be output. Since $b = \sum_{i=0}^{j+1} s_i - t_i \geq 0$ there are enough 1's available to allow us to output t_{j+1} of them. Having done this there will be b 1's which we must store. Since $b \leq k - 1$ we can store them and have space to insert and delete a 0. The pair is therefore allowable and the proof is complete. □

For the special case of a priority queue of capacity 2 we have the following unexpected result; the numbers (x_n) are every second Fibonacci number.

Theorem 4.3 *The number of binary allowable pairs of length n for a priority queue* *of capacity* 2 *is* x_n*, given by*

$$\begin{aligned} x_0 &= 1, \\ x_1 &= 2, \\ x_n &= 3x_{n-1} - x_{n-2}. \end{aligned}$$

Proof: We partition the set of allowable pairs of length n into 3 subsets according to the first data item of the input sequence σ and the first data item of the output sequence τ. Each pair of binary sequences has one of three possible structures according to this partitioning

- $(0\sigma', 0\tau')$
- $(1\sigma', 1\tau')$
- $(1\sigma', 0\tau')$

The first two classes can be counted easily; the number of pairs $(w\sigma', w\tau')$ for some $w \in \{0, 1\}$ is precisely the number of allowable pairs (σ', τ') of length $n-1$. Therefore there are x_{n-1} 2-allowable pairs in each of these classes. The third class does not yield so easily and we must split it into further subclasses.

Any pair $(\sigma, \tau) = (1\sigma', 0\tau')$ can be split into two unique pairs $(1\alpha 0, 0\beta 1)$ and (γ, δ) where $\sigma = 1\alpha 0\gamma$, $\tau = 0\beta 1\delta$. The point where this split occurs is defined by the extended partial sum criterion; we evaluate the sum for each $0 \leq j \leq r$ and split the pair at the i^{th} 0, where i is the least value of j for which the sum evaluates to 0. To count the total number of pairs we must fix the position of the split as occuring just after the i^{th} data item in the input and output sequences. Then we must count how many possibilities there are for $(1\alpha 0, 0\beta 1)$ and (γ, δ). There is no restriction on the second of these pairs except that it is 2-allowable and so there are x_{n-i} possibilities. For the first pair things are even easier since we are dealing with a priority queue of capacity 2. Throughout the entire computation transforming $1\alpha 0$ into $0\beta 1$ there is always a 1 held in the priority queue. If this were not the case then we would have a contradiction for the extended partial sum would evaluate to 0 at the first point where no 1 was stored in the priority queue. Therefore the rest of the computation must be a 1-allowable pair for a priority queue. This means $(\alpha 0, 0\beta)$ is a 1-allowable pair, so $\alpha 0 = 0\beta = 0\pi 0$. Further, if π contains any 1's at all we will again have a contradiction since the partial sum would evaluate to 0 at the point where the 1 occurred in π. We can therefore conclude that π takes the form 0^{i-3} and there is only one choice for each i. The general form of a pair from the third class is now $(10^{i-1}\gamma, 0^{i-1}1\delta)$ and so the total number of pairs is the sum

$$\sum_{i=2}^{n} x_{n-i}$$

Recombining the three classes we get

$$\begin{aligned} x_n &= 2x_{n-1} + \sum_{i=2}^{n} x_{n-i} \\ &= x_{n-1} + \sum_{i=1}^{n} x_{n-i} \end{aligned}$$

and from this we easily have

$$\begin{aligned} x_n - x_{n-1} &= x_{n-1} + \sum_{i=1}^{n} x_{n-i} - x_{n-2} - \sum_{i=1}^{n-1} x_{n-i-1} \\ &= 2x_{n-1} - x_{n-2} \end{aligned}$$

which gives us

$$x_n = 3x_{n-1} - x_{n-2}$$

□

We now introduce the main result of this section.

Theorem 4.4 *If $k \geq 1$ there is a one-to-one correspondence between the set of k-allowable pairs of binary sequences of length n and the set of ordered forests of height at most $k+1$ on $n+1$ nodes.*

The theorem is proved by studying in greater detail the process by which an input sequence σ is transformed into an output sequence τ. In general there is more than one such transformation and the central idea of the proof is to identify a canonical such transformation. This canonical transformation is the *standard computation* referred to in Chapter 1. However, in order to do this we need to introduce, in addition to the operations Insert and Delete-Minimum, an operation that we call *Transfer*. A Transfer operation can only be used when the priority queue is empty and the next input is a 1; it moves this next symbol directly from the input to the output. Clearly, permitting Transfer operations does not affect the definition of k-allowability (at least, if $k \geq 1$) since a Transfer operation can be simulated by an Insert and Delete-Minimum.

Not every sequence of Insert, Delete-Minimum and Transfer operations makes sense. As always, it is necessary that the i^{th} Delete-Minimum operation is preceded by at least i Insert operations and there must be, in all, equal numbers of Delete-Minimum operations and Insert operations. In addition a Transfer operation is only permitted if there are equal numbers of Insert and Delete-Minimum operations preceding it. A sequence of Insert, Delete-Minimum and Transfer operations which satisfies these two conditions will be called an *extended computation*. An extended computation containing a total of n Insert and Transfer operations is said to have size n (on the grounds that it would be applied to an input sequence of length n).

Lemma 4.19 *The number of extended computations of size n is c_{n+1}, the $(n+1)^{th}$ Catalan number.*

Proof: Let e_n be the number in question. The number of extended computations of size n which have no transfer operation is known to be c_n since they are simply the valid stack computations mentioned in the introduction to Chapter 3. All other extended computations of size n can be expressed uniquely in the form $\Theta T \Phi$, where Θ, Φ are extended computations of sizes j and $n-j-1$ for some j, Θ has no Transfer operation, and T is a Transfer operation. Therefore

$$e_n = c_n + \sum_{j=0}^{n-1} c_j e_{n-j-1}$$

The inductive hypothesis $e_m = c_{m+1}$ for all $m < n$ (which is true for $m = 0$) then gives, since $c_0 = 1$,

$$e_n = \sum_{i=0}^{n} c_j c_{n-j}$$

Hence $e_n = c_{n+1}$ by one of the standard identities satisfied by Catalan numbers. □

An extended computation is said to be ***standard for*** (σ, τ) if it transforms σ into τ and never performs an Insert operation when it is possible to generate a further symbol of τ (by either a Delete-Minimum or Transfer operation). To clarify this definition consider how $\sigma = 100$ might be transformed into $\tau = 001$. An extended computation necessarily begins by inserting 1 and then inserting 0 into the priority queue. It could continue either with another Insert and then three Delete-Minimum operations or it could have a Delete-Minimum, an Insert and then two Delete-Minimum operations. Only the latter would be standard for $(100, 001)$ since it generates the output symbols as soon as possible.

Notice that if an extended computation is standard for the pair (σ, τ) and is applied to σ then there is never more than one 0 in the priority queue at a time and as soon as a 0 is inserted it must be removed. This is because once a 0 is inserted it is necessarily the next symbol to be output and therefore, by the definition of standard, must be output immediately.

For ease of exposition we shall express the fact that an extended computation C is standard for the pair (σ, τ) by writing $C \sim (\sigma, \tau)$. Then we have

Lemma 4.20 $\sim$ *defines a one-to-one correspondence between the set of allowable pairs of binary sequences and the set of extended computations.*

Proof: First we show that for every binary allowable pair (σ, τ) there exists a unique extended computation C with $C \sim (\sigma, \tau)$.

Since (σ, τ) is allowable there exists some sequence S of Insert and Delete-minimum operations that transforms σ into τ. S itself may not be standard for (σ, τ) because at some point data items are inserted into the priority queue even though further elements of τ could have been generated by Delete-Minimum or Transfer operations instead. However, we can change S into an extended computation, C, that is standard for (σ, τ). To do this we first systematically defer Insert operations until Delete-Minimum operations have generated whatever further elements of τ are possible. Then we replace every Insert, Delete-Minimum pair of operations which inserts and deletes a 1 from an empty priority queue by a Transfer operation. Since each operation performed by an extended computation that is standard for (σ, τ) is determined entirely by the next output symbol to be generated and the contents of the priority queue, C is the unique extended computation with $(\sigma, \tau) \sim C$.

We may now define a map Δ from the set of allowable pairs of length n to the set of extended computations of size n by setting $\Delta(\sigma, \tau) = C$ where C is the unique extended computation with $(\sigma, \tau) \sim C$.

This map is injective for suppose there were two different allowable pairs (σ, τ) and (α, β) for which $(\sigma, \tau) \sim C$ and $(\alpha, \beta) \sim C$. Then σ and α are different since τ and β are determined by σ, C and α, C. Let α differ from σ first in the i^{th} position. The operation of C which removes σ_i and α_i from the input cannot be a Transfer operation because Transfer operations are applied only to the symbol 1. Thus the operation of C which removes σ_i and α_i is an Insert operation; let it be the j^{th} operation of C

- If C_{j+1} is an Insert operation then $\sigma_i = 1$ and so $\alpha_i = 0$, but then C is not standard for (α, β).
- If C_{j+1} is a Delete-Minimum operation then consider how many 1's are stored in the priority queue at this time. Under the assumption that C is standard for both pairs, both computations are storing only 1's and this number is fixed by C. If there are no 1's stored then $\sigma_i = 0$ so $\alpha_i = 1$ and therefore C is not standard for (α, β). If there are $q > 0$ 1's being stored, then $\sigma_i = 0$ again, so $\alpha_i = 1$ and C is not standard for (α, β).

According to Lemma 2 of [Atk93] the number of allowable pairs of length n for an unbounded priority queue is equal to c_{n+1}, the number of extended computations of size n. Therefore Δ is an invertible map and there is a bijection between the set of allowable pairs of length n and extended computations of size n. Therefore $\sim$ is a one-to-one correspondence. □

Lemma 4.21 *There is a one-to-one correspondence between extended computations* *and ordered forests*

Proof: There is a well known correspondence between valid bracket sequences and trees, see e.g. [vLW92]. This gives us a correspondence between computations and trees as shown in Figure 19 since, clearly, we can let "(" correspond to the

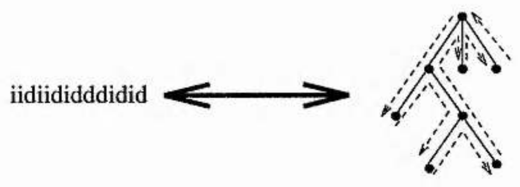


Figure 19: The computation-tree correspondence

Insert operation i and ")" correspond to the Delete-Minimum operation d. This can be further modified by starting a new tree every time a Transfer operation is encountered. For example, in Figure 20, the computation U corresponds to the tree W and V corresponds to X. This gives us the final correspondence between extended

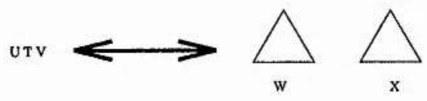


Figure 20: The full correspondence

computations and ordered forests and it is clear that the correspondence is bijective.
□

It remains to show that both the correspondences have the correct properties with respect to k-allowability and height of trees. Suppose (σ, τ) is k-allowable. From the construction of the function Δ in Lemma 4.20 the extended computation associated with (σ, τ) can be carried out using a priority queue of capacity k. From the construction of Lemma 4.21 the corresponding forest has height at most $k+1$. Conversely, if $k \geq 1$, every forest of height $k+1$ is associated with a pair that is k-allowable. This is clear from the construction of the tree since, at a point l levels below the root, the priority queue contains l data items. This can be proved inductively since an Insert operation goes a level further down in the tree and increases the number of data items stored in the priority queue by one and a Delete-Minimum operation goes up a level and reduces the number of data items stored in the priority queue by one. It is important to note that this argument does not hold for $k = 0$ since the forest of n single node trees has height 1 and corresponds to a succession of n Transfer operations (which can be carried out by a 0-capacity priority queue); yet the corresponding allowable pair is $((1,\ldots,1),(1,\ldots,1))$ which needs a priority queue of capacity 1.

This discussion completes the proof of Theorem 4.4 $\square$

We now note that ordered forests of height $k+1$ on $n+1$ nodes are in one-to-one correspondence with ordered trees of height $k+2$ on $n+2$ nodes and so we can appeal to the theory developed in [dBKR72]. A generating function for the number of trees with n nodes and height h or less is given there and from this the following closed form for the number of such trees $T_{n,h}$ is derived.

$$T_{n,h} = \frac{1}{h+1} \sum_{1\leq j\leq h/2} 4^n \sin^2(j\pi(h+1)) \cos^{2n-2}(j\pi(h+1)), \quad n \geq 2 \qquad (25)$$

From Theorem 4.4 and (25) we immediately have

Corollary 4.22 *The number of allowable pairs of binary sequences of length n for a priority queue of capacity k is $T_{n+2,k+2}$.*

Having counted the number of k-allowable pairs we now consider the behaviour of systems of priority queues formed by serial composition. A system of this kind can perform more types of operations than a single priority queue since it must move data items internally. However, we shall ignore these extra operations and simply assume they work in an intuitive manner. If a data item stored in the system is required at the output we shall assume that it is moved there using internal operations. We shall also assume data items are kept as near the input side of the system as is possible without interfering with the operation of the system. We shall consider an Insert operation to be possible if there is spare capacity in the earliest priority queue of the system or if there is spare capacity in a later priority queue and the system performs some internal operations to move a data item out of the earliest priority queue. Similarly a Delete-Minimum operation removes the minimum data item from the entire system of priority queues. If it is not possible to remove the least data item, say because an intermediate priority queue is full, then we consider the computation to have failed and not to have computed the pair.

In this case we have an analogous, but slightly weaker, notion of a standard computation to that used for a single priority queue. We shall say a computation is standard for a pair (σ, τ) if it transforms σ into τ, never performs an Insert operation when it is possible to generate a further symbol of τ by a Delete-Minimum and never moves a data item from an earlier priority queue to a later priority queue unless it must.

It then turns out that the binary case is significantly easier than the permutation case, for we have

Theorem 4.5 *Over a binary alphabet* $L(P_k)L(P_j) = L(P_{j+k-1})$

Proof: P_kP_j can be shown to simulate P_{j+k-1} using the same "spare location" technique used to prove Lemma 3.2 and Theorem 3.4. It is therefore only necessary to show that P_{j+k-1} can simulate P_kP_j. We prove by induction that at all steps during a computation P_{j+k-1} can contain the same set of data items as P_kP_j and can input or output whichever data item P_kP_j can. The base case is clear, with both systems empty they are capable of precisely the same action - read in the first data item. For the inductive step we assume that P_kP_j is performing a computation which is standard for the pair it is processing. During the simulation both systems always contain exactly the same set of data items, since P_{j+k-1} will mimic the inputs and outputs of P_kP_j. Now we consider how the simulation could fail. If P_kP_j outputs a 0 then it is clear that P_{j+k-1} can also do so, since it must contain a 0 and there is nothing with a higher priority to prevent it being output. Suppose now that P_kP_j outputs a 1 but P_{j+k-1} is unable to. Then it must be the case that P_{j+k-1} has a 0 in it and so P_kP_j has a 0 in it also. Further, this 0 must be in the earlier of the two priority queues. For this to occur, the 1 must have been in the later priority queue before the 0 was read from the input, since there is no way for the 1 to bypass the 0 otherwise. We have assumed that the operations are performed according to a standard computation but this is a contradiction since the 1 could have been output without reading in the 0. Therefore this condition cannot arise and the simulation is possible. □

When forming these systems there is a point where, for any fixed input length n, increasing the size of the system has no effect on the language it generates. At this point any pair of length n or less which is allowable by an unbounded priority queue is allowable by the system. Lemma 4.5 tells us when this maximum permutational power is achieved.

Corollary 4.23 *In a system $P_{k_1} \dots P_{k_r}$ the maximum permutation power over inputs of length n is achieved when*

$$\sum_{i=1}^{r} k_i = n + r - 1$$

In [Atk93] algorithms for calculating $s(\tau)$ and $t(\sigma)$ in quadratic time are given but these do not extend to the bounded capacity case. However we can find an alternative algorithm to calculate $s_k(\tau)$ and $t_k(\sigma)$ in quadratic time. We begin by considering how to calculate $s_k(\tau)$ and we shall then derive a method of calculating $t_k(\sigma)$ from this.

As we saw in Chapter 2, a binary sequence α can be represented by a vector $\underline{a}$ where a_i is the position of the i^{th} 0 in α. This vector represents infinitely many binary sequences since there can be an arbitrary number of trailing 1's at the end of the sequence which are not described. However, for any fixed length, there can be at most one sequence which corresponds to any given vector.

A vector $\underline{a}$ corresponds to a binary sequence of length n if and only if it satisfies (26). We then say a pair of these vectors, $(\underline{a}, \underline{b})$, is allowable (corresponding to the allowability of the sequences they represent) if and only if it satisfies (26), (27) and (28).

$$i \le a_i < a_{i+1} \le n, \quad \text{for all } i, \tag{26}$$

$$i \le b_i < b_{i+1} \le n, \quad \text{for all } i, \tag{27}$$

$$b_i \le a_i, \quad \text{for all } i. \tag{28}$$

Condition (28) is a consequence of the partial sum criterion given in [Atk93]. We can extend this condition, as we extended the partial sum criterion in Lemma 4.18, to encapsulate k-allowability. Condition (29) is a direct translation of the extended partial sum criterion into the notation we use here and so we have that $(\underline{a}, \underline{b})$ is k-allowable if and only if it satisfies (26), (27) and (29).

$$b_i \leq a_i \leq b_i + k - 1 \quad \text{for all } i \tag{29}$$

If we now introduce a new vector $\underline{a}'$ with $a'_i = a_i - i$ then condition 26 becomes

$$\begin{aligned} &a'_i \leq a'_{i+1}, \quad \text{for all } i \quad \text{and} \\ &0 \leq a'_i \leq n - i, \quad \text{for all } i. \end{aligned} \tag{30}$$

and condition (29) becomes

$$0 \leq b_i - i \leq a'_i \leq b_i + k - 1 - i, \quad \text{for all } i. \tag{31}$$

The problem of calculating $s_k(\tau)$ is now the problem of finding, for a fixed vector $\underline{b}$, the number of vectors $\underline{a}'$ satisfying (27), (30) and (31). We can define a partially ordered set, $\mathcal{P}(\underline{b})$, of width two from $\underline{b}$ which encapsulates (27), (30) and (31) as follows.

We shall create two independent chains (totally ordered subsets) and then place some constraints between them. The first chain contains the same number of elements as $\underline{b}$ and they are labelled $a'_1, a'_2 \ldots, a'_r$. These elements are placed in order as shown in the left hand chain of Figure 21. The second chain is formed from the sequence $-1, 0, 1, 2 \ldots n + k - 1$ with the b_i's marked at their respective values, as shown in the right hand chain of Figure 21. The additional constraints which we then add are those which reflect condition (31), one set of these is shown in the figure.

Definition 4.2 *A linear extension of a partial ordering $<_p$ is a total ordering $<_t$ satisfying if $x <_p y$ then $x <_t y$*

A total ordering can be represented graphically in the same manner as a partially ordered set, as shown in Figure 22. Using the construction for the partially ordered set and Definition 4.2 we have the following;

Lemma 4.24 *The linear extensions of $\mathcal{P}(\underline{b})$ are in one to one correspondence with the vectors $\underline{a}'$ satisfying (27), (30) and (31).*

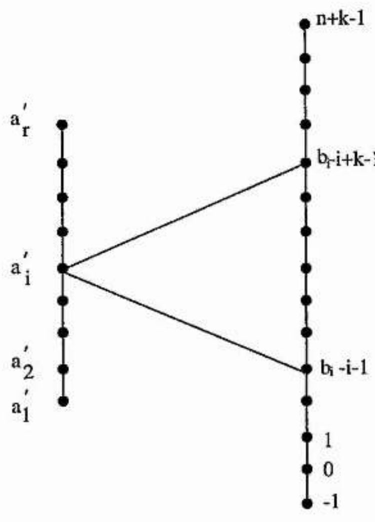


Figure 21: The partially ordered set

Proof: We begin by showing that for any linear extension of the partially ordered set we can find a unique corresponding vector $\underline{a}'$ which satisfies (27), (30) and (31).

Figure 22 shows a typical linear extension. From this we see that we can give each a'_i the value of the integer directly above it in the chain. In this way every a'_i will be given a value between 0 and $n - r$ and all the orderings in the linear extension of the partially ordered set will be honoured. It follows immediately from this that (27), (30) and (31) are satisfied by the values assigned to the a_i's.

To show this is unique, suppose there were two different linear extensions which gave rise to two equal vectors a'_i and c'_i. The linear extensions must differ in the positioning of at least one of the a'_i's or c'_i's and suppose i is the least value where they differ. Since they are in different positions the integer immediately above a'_i must differ from that above c'_i. This is a contradiction since the vectors are equal.

It remains to show that for any vector, $\underline{a}'$, which satisfies (27), (30) and (31) there is a unique corresponding linear extension of the partially ordered set.

It is easy to see that there is a linear extension corresponding to the vector. We begin with a chain constructed from the integers $-1, 0, 1, 2 \ldots n + k - 1$ and place the elements of $\underline{b}$ on it as described in the construction. We then place each a'_i immediately below the integer corresponding to its value. In the case that two elements of $\underline{a}'$ have the same value we place them in order of their subscripts, least at the bottom. To see this is a linear extension of the partially ordered set first note that the elements of $\underline{a}'$ appear in the correct order relative to each other. Then note that the elements of $\underline{b}$ appear in the correct order relative to each other. Since the values in the two vectors satisfy condition (31) their placement satisfies the orderings which appear between the two chains in the partially ordered set.

Finally, we must show that the linear extension is unique. Suppose two different vectors $\underline{a}'$ and $\underline{c}'$ both gave rise to the same linear extension. Further suppose that the first position they differ in is the i^{th} position. Then, when inserting the i^{th} element into the chain of integers, they will be placed in different positions. This is a contradiction since the linear extensions are the same. □

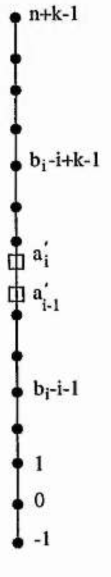


Figure 22: A linear extension of the partially ordered set

Since the vectors $\underline{b}$ are in one to one correspondence with the sequence τ and the vectors $\underline{a}'$ are in one to one correspondence with the vectors $\underline{a}$ and thus the sequences σ we immediately have.

Corollary 4.25 *For any binary sequence τ of length n which is represented by the vector $\underline{b}$ the number of linear extensions of $\mathcal{P}(\underline{b})$ is equal to $s_k(\tau)$*

Before showing the existence of an $O(n^2)$ algorithm for finding $s_k(\tau)$ we present a symmetry result.

Lemma 4.26 *(σ, τ) is k-allowable if and only if (τ^r, σ^r) is k-allowable.*

Proof: We need to show that the pair $(1^{t_r}0\dots01^{t_0}, 1^{s_r}0\dots01^{s_0})$ satisfies the extended partial sum criterion of Lemma 4.18, given that $(1^{s_0}0\dots01^{s_r}, 1^{t_0}0\dots01^{t_r})$ does. So we consider the sum $\sum_{i=j}^{r} t_i - s_i$. We know that

$$0 \leq \sum_{i=0}^{j-1} s_i - t_i \leq k-1$$

and

$$\begin{aligned} & \sum_{i=0}^{j-1} s_i - t_i + \sum_{i=j}^{r} s_i - t_i \\ = & \sum_{i=0}^{j-1} s_i - t_i - \sum_{i=j}^{r} t_i - s_i \\ = & \ 0 \end{aligned}$$

From this it follows that

$$0 \leq \sum_{i=j}^{r} t_i - s_i \leq k-1$$

as required. □

The second symmetry result mentioned at the start of the chapter for the unbounded case is that (σ, τ) is allowable if and only if $(\bar{\tau}, \bar{\sigma})$ is allowable. This does not hold in the bounded case. A simple counter example is $\sigma = 10000$, $\tau = 00001$ with a priority queue of capacity 2.

[AC87] presents an algorithm which calculates the number of linear extensions of a width two partially ordered set with constraints of the form used here. This algorithm, combined with the correspondence given in Lemma 4.25 and Lemma 4.26 gives us

Theorem 4.6 *There are algorithms which can find $s_k(\tau)$ and $t_k(\sigma)$ in $O(n^2)$ time.*

4.4 Double Ended Priority Queues

Double ended priority queues are a more complex variety of priority queues since they support one extra operation. In addition to *Insert* and *Delete-Minimum* they allow *Delete-Maximum* which, as the name suggests, removes the largest data item.

The study of double ended priority queues is barely begun in this thesis but we present some initial results; all of which are based on finding permutations with certain properties. In addition to our choice of input type and capacity constraint for the double ended priority queue we can fix the type of delete operations which occur. As an example we could fix the *delete sequence* as $\mathcal{D} = ddD$ say. This would mean that any computation adhering to this delete sequence would carry out a Delete-Minimum operation the first two times it encountered a Delete instruction. The third time it encountered a Delete instruction it would carry out a Delete-Maximum. With this restriction any computation can be described by a simple sequence of Insert and Delete operations and the delete sequence then describes which type of Delete is to be performed at each stage. With this additional constraint we introduce two new quantities, analogous to $s(\tau)$ and $t(\sigma)$.

- $s_{\mathcal{D}}(\tau) = |\{\sigma | (\sigma, \tau) \text{ is allowable with delete sequence } \mathcal{D}\}|$
- $t_{\mathcal{D}}(\sigma) = |\{\tau | (\sigma, \tau) \text{ is allowable with delete sequence } \mathcal{D}\}|$

We now consider these quantities and ask for which σ and τ are they maximal and minimal. First we require some definitions;

Definition 4.3 *$\alpha(\mathcal{D})$ is the sequence generated by inserting $\alpha = a_1, a_2, \ldots a_n$ into a priority queue and then applying the delete sequence $\mathcal{D}$ to it. The special sequence $1, 2, \ldots, n$ is denoted I.*

As an example consider $I(\mathcal{D})$:

Example 4.1

$$I(\mathcal{D})_i = \begin{cases} n-j & \text{If } \mathcal{D}_i = D \\ k & \text{If } \mathcal{D}_i = d \end{cases}$$

Where j is the number of $D's$ in $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{i-1}$ and k is the number of $d's$ in $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{i-1}$. So $I(dDdd) = 1423$ and $I(ddDDd) = 12543$.

Definition 4.4 *$I(\bar{\mathcal{D}})$ is defined as the sequence generated as above but from the complement of $\mathcal{D}$, where all Delete-Minimum instructions are replaced with Delete-Maximum instructions and vice versa.*

Lemma 4.27 *$s_{\mathcal{D}}(\tau)$ attains its unique maximal value, $n!$, with the sequence $\tau = I(\mathcal{D})$.*

Proof: It is clear that any input can be transformed into $I(\mathcal{D})$ simply by inputting all n data items and then applying the delete sequence. Therefore it gives the maximum value for $s_{\mathcal{D}}(\tau)$. To show it is the unique maximum suppose we have another output $\beta \neq I(\mathcal{D})$ and suppose it first differs from τ in the i^{th} position. If $\mathcal{D}_i = D$ then $\tau_i > \beta_i$ and since this is the first position in which the two outputs differ, τ_i must appear later in β. However, there exists a σ in which τ_i precedes β_i and so it is not possible to produce β from σ. The case $\mathcal{D}_i = d$ is almost identical. □

Lemma 4.28 *$s_{\mathcal{D}}(\tau)$ attains its not necessarily unique minimal value with the sequence $\tau = I(\bar{\mathcal{D}})$*

Proof: We show that the only input for which (σ, τ) is $\mathcal{D}$-allowable is the case $\sigma = \tau$. Suppose $\sigma \neq I(\bar{\mathcal{D}})$ and it first differs at position i. Consider if $\bar{\mathcal{D}}_i = D$ then $\tau_i =$ maximum of $1 \ldots n$ not in $\tau_1 \ldots \tau_{i-1}$ and $\sigma_i < \tau_i$. Clearly σ_i must be input before the i^{th} output of any computation involving σ can be produced. Therefore it is not possible to produce τ as an output from σ. A similar argument holds for the case $\bar{\mathcal{D}}_i = d$. To see that this minimum is not necessarily unique consider $\mathcal{D} = dDd$. It is clear that both 213 and 312 have exactly one input which forms an allowable pair. □

Lemma 4.29 *$t_{\mathcal{D}}(\sigma)$ attains its unique minimal value, 1, with the sequence $\sigma = I(\mathcal{D})$*

Proof: We show that the only possible output is σ. If $\mathcal{D}_i = D$ then, by the definition of $I(\mathcal{D})$, $\sigma_i =$maximum of all data items in $1 \ldots n$ not in $\tau_1 \ldots \tau_{i-1}$. Therefore

σ_i will be the next data item output. Similarly if $\mathcal{D}_i = d$ then σ_i will be the next data item output. Therefore there is only one possible output for the input σ. To show this is a unique minimum we simply note that for any σ, the pairs (σ, σ) and $(\sigma, I(\mathcal{D}))$ are allowable. □

5 Conclusions

In this thesis we have investigated the behaviour of several well known abstract container data types under a variety of conditions. We considered cases where the data type has finite capacity and cases where it has infinite capacity. In addition we investigated their behaviour on various forms of data: binary sequences, permutation sequences and arbitrary sequences. We have drawn a distinction between oblivious abstract container data types such as stacks, queues and deques and non-oblivious abstract container data types such as priority queues and double ended priority queues. The distinction is simply that the operation of an oblivious abstract container data type is independent of the relative values of the data items it is processing and the operation of a non-oblivious abstract container data type is not. We also introduced transportation networks, a graph theoretic model of data movement from a source node to a destination node on a graph. These can be used to represent data transmission from one machine to another on a packet switched network. More importantly, in the context of this thesis, an infinite number of finite capacity oblivious abstract container data types can be described using this model.

In analysing the behaviour of both the abstract container data types and the transportation networks we have viewed them as abstract machines. These machines execute programs formed from Insert and Delete instructions. An Insert instruction causes the machine to read a data item from the input sequence. Similarly a Delete operation causes the machine to delete a data item and place it at the end of the output sequence. To a great extent the rules governing which data items can and cannot be deleted from the machine completely characterise its operation. As the machine executes instructions the input sequence is gradually consumed and, if it is finite, will eventually become empty. At this point the only possible action for the machine is to execute Delete instructions until it is empty. The output sequence then contains every data item that was in the input sequence, but not necessarily in the same order. Any pair of sequences (σ, τ) in which σ can be transformed into τ in this way is called an allowable pair.

In the study of abstract machines it is usual to define and study the language of the machine. In keeping with this we define the language of an abstract container data type to be its set of allowable pairs. The aim of this thesis has been to count, characterise and find algorithms for recognising the members of a given language.

The investigation of transportation networks in Chapter 2 introduced a method of encoding permutations of $1, 2, \ldots, n$. For permutation inputs we noted that since the network is oblivious we need only consider the input sequence $1, 2, \ldots, n$ and count the number of permutations which form an allowable pair with this sequence. We call these allowable permutations and the total number of allowable pairs is $n!$ times the number of allowable permutations. We then showed that for any fixed network the set of encodings of allowable permutations is a regular set. The proof of this was constructive and gave rise to a method for constructing an unambiguous regular grammar. This grammar generates a set of strings which are in a one to one correspondence with the allowable permutations of the network. From this grammar we can also derive a recurrence relation for the number of allowable permutations and find the asymptotic behaviour of the number of allowable permutations as the length tends to infinity.

It was also shown in Chapter 2 that for binary inputs a network can be simulated by a finite buffer. That is, for any network N, there is a buffer of some fixed capacity k with $L(N) = L(B_k)$. We also showed that the network was equivalent to a stack, deque, input restricted deque or output restricted deque. In Chapter 3 we gave a recurrence for the number of binary allowable pairs of a bounded buffer. This combined with the equivalence result of Section 2.4 to give us a method of calculating the number of binary allowable pairs for finite capacity buffers, stacks, deques, input restricted deques, output restricted deques and transportation networks. The counting result for buffers on binary inputs was then extended by showing that systems formed by serial or parallel composition of buffers are equivalent to a single buffer of some greater capacity. This also applied to the other abstract container data types mentioned above due to their equivalence with buffers.

Chapter 4 was devoted to non-oblivious abstract container data types. Here we have investigated a priority queue of capacity 2 with permutation inputs. We found a recurrence and generating function for the number of allowable pairs and the asymptotic behaviour of these numbers as the input length tends to infinity was given. We also gave some consideration to a priority queue of capacity 2 whose input sequence is chosen from an arbitrary multiset. A multi segment tree was defined and it was shown that these trees, with certain restrictions, are in one to one correspondence with the allowable pairs. From the structure of these trees we then derived a recurrence for the total number of allowable pairs. When the input sequence was chosen from the binary alphabet we gave a more general result. We showed there is a one to one correspondence between the binary allowable pairs of length n for a priority queue of capacity k and ordered forests of height no more than $k+1$ on $n+1$ nodes. A closed formula for the number of such forests is given in [dBKR72].

We then went on to consider the situation where the priority queue has permutation inputs but the priority of each data item is independent of its value. We restricted the priorities to the binary alphabet and looked at the cases when there were 1,2 or 3 data items of priority 0 in the input. Expressions for the number of allowable pairs in these cases were derived and several more were stated without proof. We then presented two conjectures, one of significant complexity, stating how to construct the above expressions for any fixed number of data items of priority 0.

5.1 Open Problems

There is a wealth of problems still to be investigated in this area. One of the original and still unsolved problems is the case of a priority queue of capacity k and permutation inputs. In this thesis we presented the solution for $k = 2$ but the general case, and indeed the case $k = 3$, appears significantly more complex.

Conjectures 1 and 2 in Section 4.2 appear hard to prove and so far there has been no progress in that direction. Indeed it is hard even to find any intuition for the result. The only structural link between the problem and conjectured solution is the sequence a_n defined there. This sequence is the number of connected permutations of length n; those which have no prefix $\sigma_1, \sigma_2, \ldots, \sigma_i$ which is a permutation of $1, 2, \ldots, i$ for $i < n$. The analysis presented in this thesis categorises the allowable pairs by their structure, according to how many 0's are "bunched" together in the output part of the allowable pair. The numbers a_n appear to correspond to the number of ways in which the 0's can bunch together. The numerical evidence presented in Figure 18, however, is by far the most convincing justification of the conjectures to date.

There are several areas within this work which have been mentioned but require more work. The first is the unbounded, unrestricted deque. Some empirical data has been found using the technique in Chapter 2 for the bounded case but no limit has been found for the eigenvalues. The allowable permutations have been characterised by pattern avoidance in [Pra73] but no method of counting them, other than direct enumeration, has been found.

The study of double ended priority queues as a whole is only touched upon in this thesis. It seems likely that some results should come with relatively little effort. However, a completely general solution would involve an extra degree of complexity, since it must encompass all possible choices of delete sequence.

Another significant area which this thesis has touched upon is that of composition of data structures. There are some results in the serial and parallel composition of buffers and, in the binary case, priority queues, stacks, deques and networks. Unfortunately the ultimate goal, a method for describing the permutational power of systems formed by the arbitrary composition of data structures, seems unattainable. Lemma 4.10, for instance, shows that a system of priority queues in series is not permutationally equivalent to any other system of priority queues in series. It seems likely that parallel composition would prove no better. Intuitively, it seems likely that if there were any concise method for describing the permutational power of such a system it would be close to the methods used for reasoning about transportation networks. Unfortunately, even here such results seem improbable. Consider two networks, one with three nodes in series and one with two nodes in series. These two networks are clearly permutationally equivalent since they cannot permute the input sequence at all. Suppose now we have a third network, even something as simple as a single node, and place it in parallel with each of the previous two networks. Immediately we see that they are now longer equivalent since the pair $(1234, 4123)$ is allowable by one but not the other.

The problems involving multiset data have received little attention both in this thesis and in the existing literature. Initially these problems appear significantly more complex than the equivalent cases with permutation and binary data. One might expect them to inherit the difficulties associated with both the other types of data considered. By restricting the problem some progress has been made in the priority queue case. This could be furthered by relaxing the restrictions, perhaps by considering capacity 3 and 4 priority queues. Hopefully this would lead to a more general result. Buffers, stacks and deques appear less complex than priority queues overall. The corresponding problems should then yield solutions more readily.

References

[Apo63] T M Apostol. *Mathematical Analysis.* Addison Wesley, 1963.

[Atk93] M D Atkinson. Transforming binary sequences with priority queues. *Order*, 10:31–36, 1993.

[AB94] M D Atkinson and R Beals. Priority queues and permutations. *Siam J. Comput.*, 23:1225–1230, 1994.

[AC87] M D Atkinson and H W Chang. Computing the number of mergings with constraints. *Information Processing Letters*, 24:289–292, 1987.

[ALW95] M D Atkinson, S A Linton, and L A Walker. Priority queues and multi-sets. *Electronic Journal Combinatorics*, 2(Paper R24), 1995.

[ALTar] M D Atkinson, M J Livesey, and D Tulley. Permutations generated by token passing in graphs. *Theoretical Computer Science*, to appear.

[AT93] M D Atkinson and M Thiyagarajah. The permutational power of a priority queue. *BIT*, 33:2–6, 1993.

[AT94] M D Atkinson and D Tulley. The combinatorics of some abstract data types. In *Proc. IMA Conference on Applications of Combinatorics*, 1994.

[ATar] M D Atkinson and D Tulley. Bounded capacity priority queues. *Theoretical Computer Science*, to appear.

[Bjö83] A Björner. Orderings on Coxeter groups. In *Proceedings of Conference on Combinatorics and Algebra*, Providence, RI, 1983. American Mathematical Society.

[BBL93] P Bose, J F Buss, and A Lubiw. Pattern matching for permutations. *Lecture Notes in Computer Science*, 709:200–209, 1993.

[Bry93] V Bryant. *Aspects of Combinatorics.* Cambridge University Press, 1993.

[CS63] N Chomsky and M P Schutzenberger. The algebraic theory of context-free languages. *Computer Programming and Formal Systems*, pages 118–161, 1963.

[CLR92] T H Cormen, C E Leiserson, and R L Rivest. *Introduction to Algorithms.* McGraw-Hill, Cambridge, Mass., 1992.

[dBKR72] N G de Bruijn, D E Knuth, and S O Rice. The average height of planted plane trees. In R C Read, editor, *Graph Theory and Computing*, pages 15–22. Academic Press, 1972.

[EMCO92a] H Ehrig, B Mahr, I Classen, and F Orejas. Introduction to algebraic specification. part 1:formal methods for software development. *Computer Journal*, 35:451–459, 1992.

[EMCO92b] H Ehrig, B Mahr, I Classen, and F Orejas. Introduction to algebraic specification. part 2:from classical view to foundations of systems specifications. *Computer Journal*, 35:460–467, 1992.

[EI71] S Even and A Itai. Queues, stacks and graphs. In Z Kohavi and A Paz, editors, *Theory of Machines and Computations*, 1971.

[ELP72] S Even, A Lempel, and A Pnueli. Permutation graphs and transitive graphs. *Journal of the Association for Computing Machinery*, 19(3):400–410, 1972.

[GZ94] M Golin and S Zaks. Labelled trees and pairs of input-output permutations in priority queues. In *Proc. 20th International conference on Graph-Theoretic Concepts in Computer Science (WG)*, Munich, Germany, June 1994.

[GK82] D H Greene and D E Knuth. *Mathematics for the Analysis of Algorithms.* Birkhauser, Boston - Basel - Stuttgart, 2nd edition, 1982.

[HU79] J E Hopcroft and J D Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[KSW96] André E Kézdy, Hunter S Snevily, and Chi Wang. Partitioning permutations into increasing and decreasing subsequences. *Journal of Combinatorial Theory (A)*, 73(2):353–359, 1996.

[Knu73a] D E Knuth. *Fundamental Algorithms, The Art of Computer Programming.* Addison-Wesley, Reading, Mass, 1973.

[Knu73b] D E Knuth. *Sorting and Searching, The Art of Computer Programming.* Addison-Wesley, Reading, Mass, 1973.

[Lan69] P Lancaster. *Theory of Matrices.* Academic Press, 1969.

[Lin94] S Linton. Private communication, May 1994.

[Moh79] S G Mohanty. *Lattice path counting and applications.* Academic Press, New York - London, 1979.

[Pra73] V R Pratt. Computing permutations with double-ended queues, parallel stacks and parallel queues. *5th ACM Symposium on Theory of Computing*, pages 268–277, 1973.

[Rob91] D J S Robinson. *A Course in Linear Algebra with Applications.* World Scientific, 1991.

[Rot75] D Rotem. On a correspondencebetween binary trees and a certain type of permutation. *IPL*, 4:58–61, 1975.

[RV78] D Rotem and Y L Varol. Generation of binary trees from ballot sequences. *JACM*, 25:396–404, 1978.

[SS85] R Simion and F W Schmidt. Restricted permutations. *European Journal of Combinatorics*, 6:383–406, 1985.

[Tar72] R E Tarjan. Sorting using networks of queues and stacks. *JACM*, 19:341–346, 1972.

[Thi93] M Thiyagarajah. Permutational power of priority queues. Master's thesis, School of Computer Science, Carleton University, 1993.

[THB86] E C Titchmarsh and D R Heath-Brown. *The Theory of the Riemann zeta-function.* Clarendon Press, 2nd edition, 1986.

[vLW92] J H van Lint and R M Wilson. *A Course in Combinatorics.* Cambridge University Press, 1992.