ASYMPTOTIC COMPLEXITY OF ITERATIVE COMPUTATIONS

Robert C. Melville
Ph.D. Thesis
January 1981

TR 81-451

Department of Computer Science
Cornell University
Ithaca, New York 14853
ASYMPTOTIC COMPLEXITY OF ITERATIVE COMPUTATIONS

A Thesis
Presented to the Faculty of the Graduate School of Cornell University in Partial Fulfillment for the Degree of Doctor of Philosophy

by
Robert Christian Melville
January 1981
Biographical Sketch

Robert Christian Melville was born in Los Angeles on 16 April 1954. He attended the University of Delaware where he was a Dean Scholar. He entered the graduate program at Cornell University in September 1976 where he earned his Master's degree in August 1978 and his Ph.D. in January 1981. Besides his Computer Science work, Mr. Melville pursues interests in experimental mechanical and electronic projects, tennis, Balkan studies and German.
To my parents
Acknowledgements

I would like first to thank my advisor, David Gries, for his continued support. More importantly, his unusually keen insight into the business of Computer Programming has set for me a standard of clarity in the design and presentation of algorithms that will be hard to live up to. My other two committee members, Juris Hartmanis and Tim Teitelbaum, have been particularly encouraging and helpful. Alan Demers has always been willing to listen to my ideas, both good and bad, and the chance to work with him has been one of the more rewarding parts of my graduate study. Special thanks to Nick Pippenger for an enjoyable summer at the IBM Watson Center. Fellow graduate students Steve Fortune, Paul Dietz, Gary Levin and Bob Hood have been kind enough to read and comment on some of my work. The neat proof of the frame-crossing lemma in section 5.4 was suggested by Tat-Hung Chan.

Last, but not least, I would like to thank Leon Levy for my fundamental training in Computer Science at the University of Delaware.
**Table of contents**

1 Fast Iterative Algorithms

1.1 Introduction .................................................. 1
1.2 Organization and contribution of the thesis .................... 2

2 Incremental Updating

2.1 An example of Incremental Updating: Horner's Method .......... 4
2.2 Abstraction of Incremental Updating ............................ 7
2.3 Minimum spanning tree algorithm ................................ 10

3 Deferred Updating

3.1 Introduction .................................................... 14
3.2 Abstraction of Deferred Updating ............................... 17
3.3 Paul's counter implementation ................................. 19
3.4 On-line to off-line conversion ................................. 21

4 Predicate Weakening

4.1 Introduction .................................................... 24
4.2 A data structure example ....................................... 25
4.3 A Turing Machine construction ................................. 28
4.3.1 Quadratic simulation ....................................... 29
4.4 In-place array permutation .................................... 33

5 Specific Algorithms Developed Using the Paradigms

5.1 Writing bounded Turing Machine simulation

5.1.1 Problem statement and preliminary lemmas .................. 37
5.1.2 The simulation ............................................... 40
5.1.3 Reconfiguration .............................................. 44
5.1.4 The general case ............................................. 45
5.1.5 Programming details .......................................... 46
5.1.6 Corollary ...................................................... 47
5.1.7 Extensions and open problems ............................... 48
5.2 The Controlled-Density data structure ....................... 49
5.2.1 Introduction .................................................. 49
5.2.2 Worst-case time analysis .................................... 51
5.2.3 The complete sort algorithm ................................ 56
5.2.4 Extensions of the Controlled-Density Structure .......... 59
5.2.5 Empirical results ............................................. 62
5.3 A Time-Space Tradeoff for In-place Array Permutation .... 64
5.3.1 Problem statement ........................................... 64
5.3.2 Alternative solutions ........................................ 65
5.4 Two-dimensional Turing Machine Problem .................... 70
5.4.1 Problem statement ........................................... 70
5.4.2 The construction ............................................. 72
5.4.3 Running time analysis ........................................ 76
6 Closing Remarks .................................................. 78
6.1 Comparison of the paradigms .................................. 78
6.2 Related work .................................................... 79
6.3 Suggestions for further work ................................. 80
Bibliography ......................................................... 81
List of figures

"Das Bild ist eine Tatsache" - Wittgenstein

Figure 2.1.1 Obvious method of evaluating a polynomial .......... 4
Figure 2.1.1 Improved polynomial evaluation ..................... 5
Figure 2.1.3 Sequence generator ................................. 6
Figure 2.1.4 Improved sequence generator ......................... 7
Figure 2.2.1 .......................................................... 8
Figure 2.2.2 .......................................................... 8
Figure 2.2.3 .......................................................... 9
Figure 2.2.4 .......................................................... 9
Figure 2.3.1 Cubic spanning tree algorithm ....................... 11
Figure 2.3.2 Improved spanning tree algorithm .................... 12
Figure 3.1.1 Straightforward queue in pure LISP ................. 14
Figure 3.1.2 Improved queue representation ....................... 16
Figure 3.1.3 Linear time list reversal ............................. 17
Figure 3.2.1 .......................................................... 19
Figure 3.3.1 Simple counter implementation ....................... 20
Figure 3.3.2 Improved counter implementation ..................... 21
Figure 4.2.1 Process a sequence of N insertions ................... 27
Figure 4.3.1 Tape layout for quadratic simulation ................ 30
Figure 4.3.1 Tape layout for improved simulation ................. 32
Figure 4.4.1 In-place permutation, first version ................. 34
Figure 4.4.2 In-place permutation, second version ............... 36
Figure 5.1.1 Tape layout of simulating machine ................... 41

vii
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.2</td>
<td>Transfer operation</td>
<td>43</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Reconfiguration computation</td>
<td>47</td>
</tr>
<tr>
<td>5.2.1</td>
<td></td>
<td>51</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Sort algorithm</td>
<td>58</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Insertion</td>
<td>58</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Configuration</td>
<td>59</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Empirical results</td>
<td>64</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Algorithm permute1</td>
<td>66</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Algorithm permute2</td>
<td>67</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Algorithm permute3</td>
<td>69</td>
</tr>
<tr>
<td>5.4.1</td>
<td></td>
<td>71</td>
</tr>
<tr>
<td>5.4.2</td>
<td></td>
<td>74</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Simulation algorithm</td>
<td>75</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Sub-division upon frame entry</td>
<td>76</td>
</tr>
</tbody>
</table>
Chapter 1

Fast Iterative Algorithms

Section 1.1 Introduction

The study of computational complexity has produced many interesting and elegant algorithms, in some cases with surprisingly fast running times. The linear median algorithm of Blum, et al. [5] and the fast matrix multiplication algorithm of Strassen [29] are good examples. These two algorithms are typical of a wide class of fast algorithms employing the "Divide-and-Conquer" technique - a problem of size n is decomposed into two sub-problems of size roughly n/2, which are then solved recursively. If the sub-solutions can be combined quickly into a solution to the original problem, the result is usually a fast algorithm. The Divide-and-Conquer strategy qualifies as a paradigm in the sense of Floyd [11] -- it is difficult to give a precise formalization of Divide and Conquer -- however most people recognize it as a technique.

Some characteristics of Divide-and-Conquer algorithms are:

(1) they are most naturally expressed as recursive procedures;

(2) they operate off-line[1,p109], since all the input data is needed for the decomposition step;

(3) running times are evaluated by solving a recurrence relation whose structure mimics that of the algorithm.

In this thesis we are also concerned with paradigms and patterns of reasoning but our emphasis is on algorithms with an iterative flavor. Of course, it is impossible to unequivocally classify an algorithm as

---

1paradigm - a model or pattern.
recursive or iterative; since a universal partial recursive function can be written in an iterative form (universal Turing Machine) or in a recursive form (LISP EVAL[31]), these two styles of programming are equally powerful. Nonetheless, the algorithms we study will differ from familiar recursive and Divide-and-Conquer algorithms on several counts:

(1) the algorithms are neatly expressed in an iterative form, with correctness established using loop invariants[8];

(2) many of the algorithms are on-line[1,p109], implying at least the iterative structure of an outer driving loop supplying successive inputs;

(3) running times are evaluated by solving summations, the terms of the summation corresponding to iterations of a loop.

In drawing this distinction we mean no disregard for recursive programming\(^2\), but we can state some advantages of the iterative programming style used in our work:

(1) recursive programming is awkward or grossly inefficient in some imperative programming languages;

(2) the well developed theory of correctness using invariant assertions favors an iterative programming style;

(3) many automata models operate in an inherently iterative fashion.

Section 1.2 Organization and contribution of the thesis

Our research has followed two main lines. First, in chapters two through four, we collect several examples of efficient algorithms that

\(^2\) Indeed, the author learned programming in a Burroughs shop and feels he can recurse with the best of them.
can be developed by application of certain design techniques. We abstract these design techniques and attempt to formalize them. This formalization gives insight into program design and, as such, is a contribution to the Theory of Programming. Second, in chapter five, we demonstrate the use of these design techniques in the invention of several original algorithms. These algorithms are of interest in their own right, as well as exhibiting the paradigms we discuss. The Turing Machine simulation of section 5.1 solves an open problem posed by Hartmanis. The data structure described in section 5.2 is a very simple and practical alternative to balanced trees for in-core table storage. The permutation algorithm of section 5.3 also solves a common programming problem.

Finally, in chapter 6 we mention some related work and make suggestions for further research.
Chapter 2

Incremental Updating

Section 2.1 An example of Incremental Updating: Horner's Method

Consider evaluating a polynomial:

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \]

at a point \( x_0 \). The obvious method of evaluation is shown in the program of figure 2.1.1. If \( x_0^i \) is computed by \( i-1 \) multiplications the entire loop has quadratic cost. Using the fast exponentiation trick of [19, pp399–400] results in an \( O(n \log n) \) algorithm, which is still not linear. Horner's Method for evaluating the polynomial observes the recurrence:

\[ x^{i+1} = x \times x^i \text{ for } i \geq 0 \]

and the corresponding program, figure 2.1.2, clearly has linear running time. Variable \( e \) satisfies invariantly \( e = x_0^i \) and is updated incrementally by the assignment \( e := e \times x_0 \).

```
i, p := 0, 0;
(invariant: 0 ≤ i ≤ n+1 and p = a_0 + \ldots + a_{i-1} x_0^{i-1})
de i := n+1 \rightarrow e := x_0^i; i, p := i+1, p+a_i \times e \od
```

Figure 2.1.1 Obvious method of evaluating polynomial at a point
i, p := 0, 0;
e := 1;
\text{invariant: } 0 \leq i \leq n+1 \text{ and } e = x_0^i \text{ and } p = a_0 + \cdots + a_{i-1} x^{i-1}
\text{do } i := i + 1 \rightarrow i, p := i + 1, p + a_i e; \text{ e := e + x}_0 \text{ od}

Figure 2.1.2 Improved polynomial evaluation

This is a simple example of a very important algorithmic paradigm. Like many important programming techniques it has been discovered and named by several different groups. Researchers developing optimizing compilers have coined the term "strength reduction" in order to describe the above transformation because the "strong" operation of exponentiation has been replaced by the "weaker" operation of multiplication. Similarly, if a subscript of a multidimensional array takes on values from an arithmetic progression (a very common case) a multiplication in the addressing polynomial can be replaced by an addition. The SETL compiler[18] can apply this transformation to data objects much more complex than integers such as sets, sequences and relations. Reference[27] uses the term "set-theoretic strength reduction". The SETL group also describes this transformation as "differentiation" -- in the above example the "derivative" of the operation "raise x to the i-th power" with respect to the operation "i := i+1" is to multiply by x.

Dijkstra has used the phrase "taking an invariant relation out of a loop" and we summarize an example from [9] to illustrate this terminology. Suppose we want to generate the first thousand members, in
sorted order of the sequence of integers defined by the following conditions:

(1) The integer 1 is in the sequence;

(2) If x is in the sequence, then so are 2x, 3x and 5x.

Let \( A = (A_1, A_2, \ldots, A_n) \) be the sequence generated so far (so \( n \) is the length of the sequence). We introduce variables \( i_2, i_3 \) and \( i_5 \) with meaning defined by the predicate \( P \):

\[
P(A; i_2, i_3, i_5) \equiv \\
i_2 \text{ is the least integer in } (1, \ldots, n) \text{ such that } 2 \cdot A_{i_2} \geq A_n \text{ and} \\
i_3 \text{ is the least integer in } (1, \ldots, n) \text{ such that } 3 \cdot A_{i_3} \geq A_n \text{ and} \\
i_5 \text{ is the least integer in } (1, \ldots, n) \text{ such that } 5 \cdot A_{i_5} \geq A_n.
\]

The program of figure 2.1.3 generates the sequence using the variables defined by \( P \). A value for \( i_2 \) that satisfies \( P \) could be determined by the following loop:

\[
i_2 := 1; \text{ do } 2 \cdot A_{i_2} \geq A_n \rightarrow i_2 := i_2 + 1 \text{ od}
\]

and similarly for \( i_3 \) and \( i_5 \). The resulting algorithm is correct but inefficient. To construct a more efficient algorithm, we "move the predicate \( P \) out of the loop" as shown in figure 2.1.4. In this version, the predicate \( P \) is first established outside the loop by the

\[
A := (1); \\
\text{do } n \leq 1000+ \\
\text{Determine } i_2, i_3 \text{ and } i_5 \text{ to satisfy } P(A; i_2, i_3, i_5); \\
\text{Extend } A \text{ with } \min(2 \cdot A_{i_2}, 3 \cdot A_{i_3}, 5 \cdot A_{i_5}) \\
\text{od}
\]

Figure 2.1.3: Sequence generator
initialization code.

A := (1);
i2, i3, i5 := 1,1,1;
(P(A; i2, i3, i5) established)
do n = 1000;
  Extend A with \min(2*A_{i2}, 3*A_{i3}, 5*A_{i5});
  Update i2, i3 and i5 to re-establish P
od

Figure 2.1.4 Improved sequence generator

The computation to re-establish P after adding a new entry to A is straightforward and requires only constant time:

\begin{align*}
do & 2*A_{i2} \leq A_n + i2 := i2 + 1 \od; \\
do & 3*A_{i3} \leq A_n + i3 := i3 + 1 \od; \\
do & 5*A_{i5} \leq A_n + i5 := i5 + 1 \od.
\end{align*}

In the next section we develop our own algebraic formalization of this technique; the algebraic formalization using commutative diagrams seems to be novel and is also applied to the "deferred updating" technique of chapter 3.

Section 2.2 Abstraction of Incremental Updating

The program of Figure 2.2.1 shows a loop computing some quantity \( x \) at each iteration. We collect in an abstract variable \( Q \) all state information for the loop other than the value of \( x \). The computation of \( x \) from this other state information is modeled by a function "F". Let the loop iterate \( n \) times generating a sequence of states \( Q_0, Q_1, Q_2, \)
$Q_n$ (where $Q_0$ is the initial state) with state transition $\sigma$. This implies a sequence of values for variable $x$: $x_i = F(Q_i)$, $1 \leq i \leq n$. See figure 2.2.2.

The cost of $n$ iterations is easily expressed in terms of the cost of application of $F$ and $\sigma$:

$$\text{(cost of } Q_0) + \sum_{i=0}^{n-1} \text{(cost of } \sigma Q_i \rightarrow Q_{i+1}) + \sum_{i=1}^{n} \text{(cost of } F Q_i \rightarrow x_i).$$

More simply, if the costs of application of $F$ and $\sigma$ do not change much between iterations:

initialization: $i := 0$
do $i := i + 1$ + $i := i + 1$; $Q := \sigma(Q)$; $x := F(Q)$ ned

Figure 2.2.1

Figure 2.2.2
(cost to determine $Q_0$) + $n*(cost to apply $\sigma$) + $n*(cost to apply $F$).

From the diagram of figure 2.2.2, we read that $x_{i+1} = F(\sigma(Q_i))$. If the successive values of $x$ satisfy a recurrence $x_{i+1} = T(Q_i x_i)$, there is a possibility for asymptotic improvement. Figure 2.2.3 shows the modified program and figure 2.2.4 the corresponding commutative diagram. An initial value $x_0$ is provided for $x$ and subsequent values are computed by the updating function $T$. We call $x$ the incremental variable and $T$ the incremental function.

initialization including $x_0 = x_0$; $i = 0$;
do $i = i+1$ + $i = i+1$; $Q_i = \sigma(Q_i)$; $x_i = T(Q_i x_i)$ end

Figure 2.2.3

Figure 2.2.4
Assuming that the cost of T-application does not change drastically between iterations, the cost of the modified loop is:

\[(\text{cost to initialize}) + n^2(\text{cost to apply } \sigma) + n^2(\text{cost to apply } T).\]

So a savings is possible if T is significantly cheaper than F (i.e. asymptotically cheaper). The designation of x as the incremental variable can only be done relative to all the other state information, since Q itself is "incrementally updated" by \( \sigma \).

In the example of the next section, the incremental variable is a set, thus illustrating incremental updating of a more complex data structure.

**Section 2.1 Minimum spanning tree algorithm**

Let \( V \) be a set of \( n \) nodes and \( E \subseteq V \times V \) the complete set of undirected edges. That is, the ordered pair \((a, b) \in V \times V\) is considered the same as the pair \((b, a) \in V \times V\). We write \( a \rightarrow b \) for the undirected edge between nodes \( a \) and \( b \). Let \( \text{len}(a, b) \) be the length of the edge \( a \rightarrow b \). A spanning tree on the nodes is a set of \( n-1 \) edges, \( T \subseteq E \), that forms an acyclic connected graph including each of the \( n \) nodes. The length of \( T \) is the sum of the length of the edges of \( T \), and \( T \) is a minimum spanning tree if the length of \( T \) is as small as the length of any other spanning tree on the same set of nodes.

As a practical example, suppose \( n \) distinguished pins of a wire wrap panel are to be made electrically common. A minimum spanning tree indicates an interconnection using the least amount of wire.

The following theorem [1, p173] is the basis for many minimum spanning tree algorithms.
Theorem 2.3.1: Let $\text{AGV} \subseteq \text{A-V}$ and define the set of links:

$$L = \{ a--b \mid a \in A \text{ and } b \in B \}.$$ 

Let $T$ be a minimum spanning tree for the set of nodes $A$ such that $T$ is a sub-tree of some minimum spanning tree of the whole set $V$. Augment $T$ by adding the link of shortest length, $a_{\min}--b_{\min}$. This augmented tree is still a sub-tree of some minimum spanning tree of $V$.

Figure 2.3.1 shows a program implementing the construction of the above theorem. The FindMin operation is very expensive -- for $O(n)$ iterations of the outer loop the sizes of both $A$ and $B$ may be $O(n)$. This means that exhaustive search for the closest pair has quadratic cost and the whole algorithm cubic cost.

---

$A := \{\text{any node}\}; B := V-A;$

do $A = V +$

FindMin:

$L := \{a--b \mid a \in A \text{ and } b \in B\};$

Let $a_{\min}--b_{\min}$ be the shortest link in $L$;

Transfer $b_{\min}$ from $B$ to $A$

end

Figure 2.3.1 Cubic spanning tree algorithm

---

A dramatic improvement is possible by introducing an incremental set variable. Define $LL \subseteq L$:
$a \rightarrow b \in L \iff a \in A \text{ and } b \in B \text{ and } \quad \text{dist}(a,b) = \min_{c \in A} \text{dist}(c,b)$ \hfill (2.3.1)

In other words, for each $b \in B$ the set $L$ includes all edges from $b$ to an element of $A$; but the set $LL$ includes only one edge from $b$ to an element of $A$, namely the one of shortest length (ties may be broken arbitrarily and this is why the minimum spanning tree may not be unique). The size of $LL$ is now equal to the size of $B$, which is never more than $O(n)$. Moreover, the shortest edge in $L$ will also be in $LL$.

---

Select any node $a_0$ to start;

$A, B := \{a_0\}, V - \{a_0\}$;

$LL := \{a_0 \rightarrow b \mid b \in B\}$;

(!!(2.3.1)!!)

do $A \times V +$

FindMin:

Let $a_{\min} \rightarrow b_{\min}$ be the shortest link in $LL$;

Transfer $b_{\min}$ from $B$ to $A$;

Update:

$(\forall b \in B)$:

Say $a_{\text{curr}} \rightarrow b \in LL$;

if $
\text{len}(a_{\text{curr}} \rightarrow b) \leq \text{len}(a_{\min} \rightarrow b) \rightarrow
$
replace $a_{\text{curr}} \rightarrow b$ with $b_{\min} \rightarrow b$ in $LL$

elseif \ $\text{len}(a_{\text{curr}} \rightarrow b) > \text{len}(a_{\min} \rightarrow b)$ then skip

!!!(2.3.1) re-established after transfer!!!

end

Figure 2.3.2 Improved spanning tree algorithm
The transformation from figure 2.3.1 to figure 2.3.2 follows the pattern outlined above. Set LL is initialized outside the loop and maintained incrementally thereafter. When a node b_{min} is transferred from B to A, set LL must be updated. For any b still in B, if the distance from b to b_{min} is at most the distance from b to its current partner in A assigned by LL then the assignment should be changed to b_{min}, which is now a member of A. Initialization costs O(n) to create LL. The search FindMin costs O(size of B), which is O(n) at most. Updating LL upon transfer of a node into A also takes time proportional to the size of B. This is much cheaper than the O(size of A \times size of B) time needed to compute L in the program of figure 2.3.1. The total time is reckoned as O(n) + n \times O(n) = O(n^2). Observe that for a complete graph with independent edge weights, quadratic time is optimal since the algorithm must inspect each of the O(n^2) edge weights to be sure of constructing the minimum tree. The algorithm of figure 2.3.2 is attributed to Prim[25]; essentially the same idea is described in Dijkstra[7].
Chapter 3
Deferred Updating

Section 3.1 Introduction.

A straightforward implementation of a queue (first in, first out list) in pure LISP is shown in figure 3.1.1. The entries in the queue appear in reverse order of insertion as a list Q -- i.e. CAR(Q) is the oldest entry. Because of this convention the INSERT operation uses APPEND and, therefore, takes time proportional to the length of the list at the time the INSERT operation is performed. A sequence of n INSERT operations, with no intervening DELETES, would take time $O(1+2+...+n) = O(n^2)$.

Why is this so bad? Suppose inputs a, b and c are inserted into a queue with initial value Q. The first append operation must traverse Q

\[ Q = \text{LIST}[v_1, v_2, ..., v_n] \]
\[ v_1 \text{ is the oldest entry, } v_n \text{ is the most recent entry} \]

\[
\begin{align*}
\text{QUERY}(Q) &= \text{CAR}(Q)=v_1 \quad (Q=\text{NIL}) \\
\text{DELETE}(Q) &= \text{CDR}(Q)=\text{LIST}[v_2, ..., v_n] \quad (Q=\text{NIL}) \\
\text{INSERT}(Q,v) &= \text{APPEND}[Q,\text{LIST}(v)]=\text{LIST}[v_1, ..., v_n,v]
\end{align*}
\]

Figure 3.1.1 Straightforward queue in pure LISP
and results in \([Q.a]\). The second insert must traverse \([Q.a]\) and results in \([Q.a.b]\). The third insert traverses \([Q.a.b]\) and results in \([Q.a.b.c]\) etc. Traversal of \(Q\) is common to all three insertions, and this seems wasteful because the common portion of the queue is unchanged by each insertion. It is possible to "factor out" some of this common portion and store it in a separate list so that insertions are spared the expense of traversing this common portion. However, after some number of deletions or insertions, the common portion may become out of date and must be updated. This updating will be expensive but does not happen very often.

We make these ideas precise with the improved representation given in figure 3.1.2. \(H\) is a list of entries in reverse order of insertion; \(T\) is a list of entries in order of insertion. -- i.e. \(\text{CAR}(T)\) is the most recent entry. The concatenation of \(H\) with the reverse of \(T\) would give the queue as represented in figure 3.1.1: \(Q = \text{APPEND}(H, \text{REVERSE}(T))\). We impose the additional condition that \(\text{NULL}(H)\) implies that the entire queue is null. In other words, the situation \(T = \text{NIL}\) and \(H = \text{NIL}\) is disallowed. This means that a query operation can always be done by inspecting \(\text{CAR}(H)\) (unless, of course, the whole queue is empty but then the query operation is undefined anyway). The \(\text{INSERT}\) operation is always \(O(1)\) and \(\text{DELETE}\) is \(O(1)\) unless deletion would result in \(H = \text{NIL}\). In this case, the program replaces \(H\) with \(\text{REVERSE}(T)\) and replaces \(T\) with \(\text{NIL}\).

It is not difficult to argue that this new representation can be maintained in time linear in the number of queue operations. Let a \(\text{DELETE}\) operation requiring a reversal of \(T\) occur at steps \(t_1, t_2, \ldots\) (initially \(H=T=\text{NIL}\) at step \(t_0\)). Reversal of a list can be done in
The queue is a pair \((H,T)\) where
\[ H = \text{LIST}[v_1,v_2,\ldots,v_{i-1}] \quad \text{and} \quad T = \text{LIST}[v_n,v_{n-1},\ldots,v_i] \]
for some \(i\).

**QUERY**((H,T)) == CAR(H)
**INSERT**((H,T),v) == (H,CONS[v,T])

**DELETE**((H,T)) == if LENGTH[H] \(\geq\) 1  
  then (CDR[H],T)  
  else (REVERSE[T],NIL)  
fi

Figure 3.1.2 Improved queue representation

linear time by the program of figure 3.1.3. The cost of step \(t_i\) is
\(O(\text{"length of } T \text{ at step } t_i\) \text{"")}, but \(T = \text{NIL}\) after step \(t_{i-1}\) so the length
of \(T\) at step \(t_i\) cannot be more than \(t_i - t_{i-1}\). Suppose a total of \(n\) queue
operations are performed and \(m\) of them require a reversal of \(T\). The
total cost is

\[
m = O(1) + \sum_{i=1}^{m} O(t_i - t_{i-1}) = O(n).
\]

This linear time solution should be compared with a Turing Machine
construction due to Stoss[28]. A real-time LISP solution is possible
[15] and should be compared with a Turing Machine construction due to
Leong and Seiferes[22].

In section 3.2 we attempt to formalize the above transformation in
algebraic terms. Sections 3.3 and 3.4 give more examples of this
technique for efficient manipulation of data structures.
REVERSE[T] = REV[T,WIL] where
REV[X,Y] = if NULL[X] -> Y
□ mot NULL[X] -> REV[CDR[X],CONS[CAR[X],Y]]
fi

Figure 3.1.3 Linear time list reversal

Section 3.2 Abstraction of Deferred Updating

We model deferred updating with a mapping diagram as in figures 2.2.2 and 2.2.4. Let A be the domain of values of a data structure and \( \mathcal{\Theta} \) a set of operations on the data structure:

\[
\begin{array}{c}
A \\
\downarrow \Theta \\
A
\end{array}
\]

all \( \Theta \in \mathcal{\Theta} \).

In the queue example of section 3.1:

\[
A = \{(v_1, v_2, ..., v_n) \mid n \geq 0, v_i \in \text{QueueValues}\}
\]

\( \mathcal{\Theta} = \{\text{query, delete} \cup \text{insert}_v \mid v \in \text{QueueValues}\}. \)

The pair \((A, \mathcal{\Theta})\) represents the "obvious" implementation of the data structure and associated operations. However a sequence of \( n \) operations

\[
A_0 \rightarrow A_1 \rightarrow A_2 \cdots \rightarrow A_n
\]

may be expensive (e.g. \( O(n^2) \) for the queue example). To reduce the cost we introduce a homomorphically related system \((A, \Pi)\). The sets \( \mathcal{\Theta} \) and \( \Pi \)
will provide the same operations, however we cannot simply say $\mathcal{O} = \mathcal{I}$ since the domains of the operations are different sets. Again for the queue example:

$$\mathcal{A} = \{ ((v_1, v_2, \ldots, v_n) \cdot (u_1, u_2, \ldots, u_m)) \mid n, m \geq 0 \text{ and } v_i, u_j \in \text{QueueValues} \}$$

$$\mathcal{I} = \mathcal{O}, \text{ but programmed as in figure 3.1.2.}$$

Each $A \in \mathcal{A}$ may be represented by several $B \in \mathcal{B}$. Following our example, a list $(v_1, v_2, \ldots, v_n) \in \mathcal{B}$ corresponds to any of the following $m+1$ elements of $\mathcal{A}$

$$((\ast) \cdot (v_1, v_2, \ldots, v_n)) \cdot ((v_1) \cdot (v_2, \ldots, v_n)) \cdot$$

$$((v_1, v_2) \cdot (v_2, \ldots, v_n)) \cdot \ldots \cdot ((v_1, v_2, \ldots, v_n).() \cdot (\ast))$$

Let $R: \mathcal{A} \longrightarrow \mathcal{A}$ be the surjective map that takes any element of $\mathcal{A}$ to the element of $\mathcal{A}$ representing the same information. So all the $m+1$ pairs above would map to $(v_1, \ldots, v_n) \in \mathcal{A}$. Let $S: \mathcal{A} \longrightarrow \mathcal{B}$ be the injective map taking $A \in \mathcal{A}$ to some canonical representative in $\mathcal{B}$. For the queue example, $S(L) = (\text{REVERSE}(L).()) \in \mathcal{B}$.

The mapping diagram of figure 3.2.1 portrays a sequence of $n$ operations performed simultaneously on both representations of the data structure. Since the diagram commutes, any path gives the same answer. However some paths are cheaper than others. The quadratic program for queue maintenance follows a path of the form $(\mathcal{O}_i)^n$ whereas the linear program follows a path $((\mathcal{O}_i)^n \mathcal{RS})^n$. Because of the different representation of the data structure the updating operation, "RS", can be deferred for some number of $n$ operations.
Section 3.3 Paul's counter implementation

Deferred updating, or something very similar, is often used in efficient Turing Machine constructions; the Hennie-Stearns simulation [14] is perhaps the best example. In this section we present an implementation technique due to Paul [23] which is representative of the use of deferred updating in Turing Machine constructions.

Suppose a multi-tape machine $M$ needs to time itself so that it performs some computation for a bounded number of steps, say $N$. This would be necessary if $M$ wants to diagonalize away from all machines running in time bound $N$. The obvious solution is for $M$ to maintain a count of the number of steps of the computation performed so far as a binary integer at a fixed location on one of the machines tapes. The head for the tape storing the counter might move a distance $O(N)$ away from the location of the counter yet must return to this vicinity for each incrementation. The result is a miserable $O(N^2)$ bound to count to
A much better implementation is to keep the counter right under the head of some tape, say on a separate track (see figure 3.3.1). The counter has length $O(\log N)$ and can be incremented or moved one position in time $O(\log N)$. Now $N$ steps can be counted at a cost of $O(N \log N)$. Observe that when counting in binary most of the activity takes place in the low order bits.

Paul's improved implementation is shown in figure 3.3.2. The counter is split into a low order portion (LOP) of length $\log \log N$ and a high order portion (HOP) of length $\log N - \log \log N = O(\log N)$. Initially, LOP and HOP are separated by a distance of $\log N$. Each step is counted in LOP, which can be incremented and shifted in time $O(\log \log N)$. After $2\log \log N$ steps LOP may generate a carry into HOP. At this point LOP and HOP cannot be separated by a distance of more than $2\log N$. The machine first spends time $O(\log N)$ to reset LOP a distance of $\log N$ from HOP, then spends time $O(\log N)$ to increment HOP (notice that this requires a second tape). This resetting operation happens every $\log N$ steps, so it can happen a total of $N/\log N$ times. The total cost to count to $N$ can be reckoned as: $N=O(\log \log N) +$
\(O(\log N) \times \frac{N}{\log N} = O(N \log \log N)\). The reference cited actually carries this process to the limit, storing the counter in \(\log N\) pieces to achieve a bound of \(O(N \log^* N)\).

In the notation of the previous section, \(\mathcal{O}\) and \(\Pi\) contain the operations \textsc{increment}, \textsc{shift left} and \textsc{shift right}. The domain \(A\) would be binary strings of length \(\log N\). The expanded domain \(\overline{\mathcal{A}}\) is strings of the form \(xyz\) where \(x\) is a binary string of length \(\log N - \log \log N\), \(z\) is a binary string of length \(\log \log N\) and \(y\) is a string of blanks of length \(2 \log N\). A binary integer has several representations in the \(xyz\) form, each differing in the number of blanks between \(x\) and \(y\). The \(R\) map is simply \(xyz \rightarrow xz\). The \(S\) map converts a binary string \(w\) of length \(\log N\) to \(xyz\) where \(z\) is the \(\log \log N\) low order bits of \(w\), \(x\) is the remaining high order bits, and \(y\) is a string of \(\log N\) blanks.

Section 3.4 On-line to off-line conversion

A simple but useful application of deferred updating is a device to convert an algorithm from on-line form to off-line form. Suppose a data structure is being modified by a sequence of operations processed on-line and that the size of the data structure is proportional to \(n\), the number of operations processed so far. It is often useful to know the

| \(\log N -\) | \(\log N...\) | \(\log \log N\) |
| \(\log \log N\) | 2 \(\log N\) | \(\ldots\) |
| H | O | P | bbbbbbbbb L | O | P |

^ head

Figure 3.3.2 Improved counter implementation
maximum value of \( n \) in advance, say in order to allocate memory or to compute an index into the data structure but the maximum value of \( n \) cannot be known until the input sequence terminates.

A solution is to predict the maximum value of \( n \) as far as the next power of two. That is, for \( 2^k \leq n < 2^{k+1} \) let the data structure have size \( 2^{k+1} \). If the input terminates before the \( 2^{k+1} \)-st operation then fine — the size of the data structure has been over-estimated by at most a factor of two. If \( 2^{k+1} \) inputs are received then, before processing input \( 2^{k+1} \), the current data structure is rebuilt with size \( 2^{k+2} \) and brought up to date by repeating operations \( 1, 2, \ldots, 2^{k+1}-1 \) on the new structure. Now operations \( 2^{k+1}, 2^{k+1}+1, \ldots, 2^{k+2}-1 \) can be processed on the resized structure.

By this strategy the operations occurring at steps \( 2^k, 2^{k+1}, 2^{k+2}, \ldots, 2^{k+1}-1 \) are repeated at most \( \log n-k \) times during the resizings at steps \( 2^{k+1}, 2^{k+2}, \ldots, 2 \log n \). The cost of each operation is multiplied by at most \( \log n \) so if the cost of \( n \) operations without resizing is \( f(n) \) then the cost with resizing is at most \( O(f(n) \log n) \).

In the special case that each operation has cost less than or equal to some constant \( c \), we claim that the above strategy incurs only a linear slowdown. We will "charge" the cost of the re-sizings to various operations in such a way that accounts for all the re-sizing cost but charges no operation for more than one re-sizing. A re-sizing at step \( 2^k \) costs \( c2^k \) since the \( 2^k \) preceding operations must be repeated at a cost of \( c \) each. To absorb this cost, add \( 2c \) to the cost of the operations at steps \( 2^{k-1}, 2^{k-1}+1, 2^{k-1}+2, \ldots, 2^k-1 \). This accounts for all re-sizing and triples, at most, the cost of any operation. The total cost with re-sizing is \( \leq 3cn \).
This deferred re-sizing technique is essential to the Turing Machine constructions of sections 5.1 and 5.4. In both of these constructions the simulating machine needs an estimate of the amount of memory allocated by the simulated machine so it can construct its own simulating data structure of an appropriate size. Using the schedule for re-sizing discussed above, re-sizing is not a dominant cost of either simulation.
Chapter 4

Predicate Weakening

Section 4.1 Introduction

In this chapter we demonstrate the development of efficient algorithms and data structures by manipulation of invariant predicates.

In the first example below we specify a data structure by an invariant predicate that characterizes the data structure exactly. That is, given the values to be stored, only one arrangement of data values that will satisfy the invariant. The "strict" nature of the invariant is responsible for the poor asymptotic performance of the data structure. Modifying the invariant so that several arrangements of the data values can satisfy it results in an invariant that is easier to maintain and a more efficient data structure.

The second example is a Turing Machine simulation. In the straightforward (quadratic) simulation, the simulating machine follows exactly the movements of the machine to be simulated and consequently may run inefficiently. Relaxing the requirement of such close simulation leads to a faster construction.

In our third example, we start with a familiar algorithm for in-place array permutation and modify its invariant to derive a new algorithm that provides a continuous time/space tradeoff, with the old algorithm as a limiting case.
The theme of this chapter then, is that considerations of correctness and asymptotic efficiency can be combined in the notion of an invariant predicate for a loop or a data structure.

Section 4.2 A data structure example

Let X be a domain of data values with a linear ordering ≤. We design a data structure to contain a subset of X, say S ≤ X, and support two operations: search(x) is true just if x ∈ S and insert(x) implements S := S ∪ {x}. It is convenient to assume a value ∞ ∈ X larger than any real data value. Also we assume an a priori bound, N, on the size of S.

A simple data structure for this problem is a sorted array. Let S = \{x_1, x_2, \ldots, x_n\}. These values are contained in sorted order in the first n locations of an array A[1:N+1] followed by ∞ at location A[n+1].

The data structure satisfies the following invariant:

\[ n \in \mathbb{N} \quad \text{and} \quad A[n+1] = \infty; \]
\[ (\forall i: 1 \leq i \leq n: A[i] \leq A[i+1]). \quad (4.2.1) \]

The search operation can be performed easily in logarithmic time by Binary Search on locations 1 through n+1. To insert a new value x, first determine by Binary Search the index j such that x should go before the value in A[j] (If x = A[j], then nothing more need be done). Now the contents of locations A[j], A[j+1], A[j+2], \ldots, A[n+1] are shifted up one place to make room for the new entry. Because O(n) steps may be needed for this shifting, this implementation of insert is too slow for many applications.

Predicate (4.2.1) is very "strict" in the sense described above. We use more space and introduce gaps into the array so that we can define a weaker invariant. A gap is an array location that holds a data
value but is marked in some way. For example, if the data values are positive integers then gaps can be negative integers. Let \( V(A[i]) \) be the value of \( A[i] \), whether or not \( A[i] \) is marked as a gap. For the example of positive and negative integers \( V(A[i]) \) would be the absolute value of \( A[i] \).

Predicate (4.2.2) below states that \( n \) data values alternate with \( n \) gaps in the first \( m = 2n \) locations of the array.

\[
\text{nsN and ms2n;} \\
A[m+1] \text{ is a gap and } V(A[m+1])=\infty; \\
(\forall i: 1 \leq i \leq m:\n\text{even}(i) \Rightarrow A[i] \text{ is data} ; \\
\text{odd}(i) \Rightarrow A[i] \text{ is a gap}; \\
V(A[i]) \leq V(A[i+1])
\]

Invariant (4.2.2) is no easier to maintain than (4.2.1); however, (4.2.3) given below, will be easier to maintain.

\[
\text{nsN and ms2n and even}(m); \\
A[m+1] \text{ is a gap and } V(A[m+1])=\infty; \\
(\forall i: 1 \leq i \leq m:\n\text{even}(i) \Rightarrow A[i] \text{ is data} ; \\
\text{odd}(i) \Rightarrow A[i] \text{ is data or a gap}; \\
V(A[i]) \leq V(A[i+1])
\]

Note that (4.2.2) \( \Rightarrow \) (4.2.3). In other words, if the data structure is in a state satisfying (4.2.2), then it is also in a state satisfying (4.2.3). Our construction here is a two-step process. First, we go from (4.2.1) to (4.2.2) by introducing gaps. Then invariant (4.2.2) is weakened to (4.2.3) to allow more efficient updating. Let \( A \) satisfy (4.2.3). If we ignore gap markings, then the sequence \( A[1:m+1] \) is still
monotonic and Binary Search can be carried out in time $O(\log 2n) = O(\log n)$. To insert a new value the place for the insertion is determined by a Binary Search, but now shifting proceeds only as far as the nearest gap to the right. This gap is used up by the insertion of the new element. If insertion causes a shift past location $m$, then $m$ is increased by two and a new gap is created at location $A[m-1]$. Note that $m$ is increased by at most two whenever $n$ is increased by one so the relation $m \leq 2n$ is satisfied.

As long as each data value is followed by a nearby gap, the insertion operation will be fast. However, long gap-free sequences will build up after many insertions causing the algorithm to run more slowly. At this point, it would be better if the array were returned to a state satisfying (4.2.2). The program of figure (4.2.1) models a sequence of $N$ insertions starting from an empty set. A search operation can be performed at any time since it does not modify the array. Predicate (4.2.2) is invariant for the outer loop; (4.2.3) is invariant for the inner loop.

The unspecified condition "Sched" controls how often the array is returned to a state satisfying (4.2.2). The operation "re-establish

\begin{verbatim}
A[1] := \infty marked as a gap; n := 0;
ed
if n \neq n
    do n \neq n and met Sched +
       let x be the next value to be inserted;
       insert x preserving (4.2.3)
    od;
    re-establish (4.2.2)
ed
\end{verbatim}

Figure 4.2.1 Process a sequence of $N$ insertions
\((4.2.2)^n\) can be done in time \(O(n)\). This is expensive but must be scheduled often enough to avoid the build up of dense sections in the array. Section 2 of chapter 5 proves the following result. Let \((4.2.2)\) be re-established whenever \(n\) becomes greater than \(3/4 \cdot m\) or whenever an insertion encounters a gap-free section of length \(\geq \sqrt{m}\). Then the worst-case time for \(N\) insertions is \(O(n^{1.5})\).

The asymptotic bound \(O(n^{1.5})\) is inferior to the \(O(N \log N)\) bound possible with balanced trees[1, Ch 4]. However, empirical studies show that the array structure is significantly faster than a balanced tree for \(N\) as large as several thousand, even when the array structure is forced to its worst-case performance. A multi-level construction leads to a worst-case time of \(O(n^{1+1/k})\) for \(k\) levels. However, the programming becomes messy for \(k > 2\) and much of the simplicity of the algorithm is lost.

Section 4.3 A Turing Machine construction

The two common resource measures for the Turing Machine model are \(\text{TIME}\) -- the number of steps in a computation and \(\text{SPACE}\) -- the number of different tape cells visited during the computation. A third measure, the \(\text{INK}\) measure, counts how many times the machine changes a symbol of its worktape -- i.e. reading, state change and head movement are free. This measure has some of the properties of \(\text{TIME}\) and some of the properties of \(\text{SPACE}\) and is an abstract model of memory that is cheap to read but expensive to modify.

We consider the design of a universal machine, \(U\), which given as input an encoding of any machine \(M\) and an input \(x\) can simulate the computation of \(M\) on \(x\). To simplify the following discussion we assume
that any such $M$ presented to $U$ for simulation will **eventually halt** on its input. The universal machine can, in fact, be designed to detect the condition that $M$ is caught in an infinite loop but is not using ink. Being able to detect this condition is essential to the diagonalization proof of section 5.1.5, but these details are not needed for our description of predicate weakening here. A number of important but messy implementation details of the simulation are omitted from the following discussion. The careful or interested reader is referred to the complete presentation of section 5.1.

We want the machine $U$ to be efficient in its own INK expenditure, that is, to incur as little simulation overhead as possible. The naive simulation we present first is quadratic: if machine $M$ uses $t$ units of ink during the first $n$ steps of processing $x$ then $U$ might use as much as $O(t^2)$ units of ink to simulate these first $n$ steps. An improved simulation algorithm, developed as an instance of predicate weakening, gives an $O(t^{1.5})$ bound. The detailed construction of section 5.1 extends this bound to $O(t^{1+1/k})$ for any fixed $k > 0$.

**Section 4.1.1: Quadratic simulation**

The universal machine will organize its worktape as shown in figure 4.3.1. Track 1 contains a copy of the worktape of $M$; Track 2 contains the encoding of $M$ along with the current state of $M$ to be consulted during the simulation. Let the length of this encoding be some $c$ symbols. Between simulation steps, the workhead of $U$ is positioned at

---

1 Here we assume that the tape alphabet of $U$ is at least as large as the tape alphabet of $M$. If not, then several symbols from $U$'s alphabet will be used to encode each tape symbol of $M$. 
the left edge of the machine and state encoding; moreover, $U$ follows the convention that the symbol on track 1 just above this left edge is the symbol that would be be under the workhead of $M$. To perform one simulation step, $U$ uses the current state and track 1 symbol as indices into the transition table of $M$. If the required action by $M$ is to change state or change the symbol under its workhead then $U$ can easily update tracks 1 and 2 as appropriate. If the action of $M$ is to shift its head left or right, then $U$ must shift the entire contents of track 2 one position left or right as required. This will cost $O(c)$ units of INK. Here is a problem: a shifting move costs $M$ no ink, yet $U$ must still spend $O(1)$ ink to simulate it.

The lemmas proven in section 5.1.1 imply the following facts about halting machines. First, after printing $t$ times $M$ can form non-blank tape contents of length at most $O(t)$. Second, $M$ can move at most $O(t)$ steps between printing moves. In the worst case, $M$ might move $O(t)$ steps between each of $t$ printing move forcing $U$ to spend $O(c) \times t \times O(t) = O(t^2)$ ink.

![Diagram of tape layout for quadratic simulation](image)

Figure 4.3.1 Tape layout for quadratic simulation
Figure 4.3.1 is actually an informal statement of an invariant. The invariant requires that track 1 be a replica of the tape contents of M and that the head of U follows the movements of the head of M. This invariant, like (4.2.1), is very "strict" and it is this strictness that causes inefficiency. We attempt to weaken the requirements of the simulation in some fashion; let us introduce a different representation of the worktape that does not require such close correspondence between the head movements of M and U, and for this reason, the resulting simulation is more efficient. Again, our construction is a two-step process. We introduce a new data structure and then specify a weaker invariant for the simulation.

After n steps of operation including t printing moves, suppose M has formed non-blank tape contents of length t. U will represent this tape as a series $\sqrt{t}$ segments, each of length $\sqrt{t}$. The ends of each segment are delimited by a segment table as shown in figure 4.3.2. A segment table records, for each state q of M, what would happen if M were to enter the segment in state q. The possible outcomes are:

1. print inside the segment;
2. halt inside the segment;
3. exit on the left in some new state (without having printed);
4. exit on the right in some new state (without having printed).

U will write the current state of M under a segment table and then consult the table. In case (1) U simulates M inside the segment until it prints and then must update the segment table because the segment has now been changed. Updating the segment table for a segment of length $\sqrt{t}$ can be done in time and ink $O(\sqrt{t})$ (see section 5.1). Therefore the
1/2

\[ \text{track 1: } \ldots \text{ST}10110100110\text{ST}1001101101\text{ST}10110110010\text{ST} \ldots \]

\[ \text{track 2: } \underline{3} \]

\[ \text{track 3: } \text{encoding of } M \]

|ST| segment table

Figure 4.3.1 Tape layout for improved simulation

Simulation overhead for printing moves is $O(\sqrt{t})$ per printing move and $O(t^{1.5})$ for all $t$ printing moves. In cases (3) or (4) $U$ must move the state record to one end of the segment or the other. The state record has constant length (i.e., depending only on $M$) and, so, can be transferred across a segment using $O(1)$ ink. In the worst case described above, $M$ might move its head $O(t)$ times between printing moves crossing $O(\sqrt{t})$ segments. Once again, this simulation overhead is $O(\sqrt{t})$ ink for $U$ to follow these segment crossings for a total of $O(t^{1.5})$ for $t$ printing moves by $M$.

The improved simulation maintains a weaker invariant in the following sense: The head positions of $M$ and $U$ must now match only at segment boundaries. When $M$ enters a segment, makes a complicated series of head shifts, and comes out again on the other side, $U$ "abbreviates" this movement to a single transfer of its state record.
Section A.4 In-place array permutation

Let $a[1:n]$ be an array and $f$ a permutation of the integers between 1 and $n$. To permute $a$ by $f$ means to replace $a[1]$ with the original contents of $a[f(1)]$, replace $a[2]$ with the original contents of $a[f(2)]$, ..., etc. This operation can be described neatly as the concurrent assignment

$$a[1], a[2], \ldots, a[n] := a[f(1)], a[f(2)], \ldots, a[f(n)].$$

However, such an assignment is not possible unless $n$ is known at compile time. The permutation is easily programmed if a temporary array $b[1:n]$ is available, but this may require too much storage in some applications and we consider algorithms that operate "in-place". A permutation algorithm operates in-place if the only operation modifying the array is the interchange of the contents of two locations, say $a[i]$ and $a[j]$, which we write as "swap(i,j)".

Any permutation is a composition of disjoint cycles and we write cycle(i) for the set $(i, f(i), f^2(i), \ldots, f^{s-1}(i))$ where $f^s(i) = i$ and $s$ is the length of the cycle. This cycle is permuted in-place by the $s$-1 swaps

$$\text{swap}(i, f(i)); \text{swap}(f(i), f^2(i)); \ldots; \text{swap}(f^{s-2}(i), f^{s-1}(i)).$$

A straightforward solution uses a boolean array mark[1:n+1] to record permuted and un-permuted cycle members; for $i$ between 1 and $n$, the desired value has been placed in location $i$ if and only if mark[$i$] is true. The ($n+1$)st location provides a convenient stopping point for the program given below. An outline of the algorithm is given in figure 4.4.1; (4.4.1) below is the main loop invariant.
The running time of the program of figure 4.4.1 is clearly $O(n)$.

We develop a new algorithm for this problem that provides a continuous time/space tradeoff. That is, for $t$ between 1 and $n$ we give an algorithm that uses a $t+1$ element boolean array and has worst-case running time $O(n^2/t)$. Our algorithm reduces to the linear time algorithm given above for the case $t=n$.

The algorithm operates in $\lceil n/t \rceil$ "phases", each phase re-using the same storage for a mark array. At the beginning of a phase a mark array of length $\leq t+1$ is allocated with bounds $lo...hi$ and initialized with all locations false. The meaning of the mark array is somewhat different from the meaning of the mark array in the program above. We must weaken the logical equivalence of (4.4.1) to just an implication: if a location is marked, it is a member of a permuted cycle. However, a location may be unmarked and still be a member of a cycle permuted during an earlier phase. These requirements are summarized in invariant
Variable \( k \) functions much as in the first version.

1. \( \text{lo} \leq k \) and the mark array has bounds \( \text{lo} \ldots \text{hi} \) where \( \text{hi} = \min(\text{lo} + t, n + 1) \);
2. \( \text{not mark}[k] \) and \( \text{not mark}[\text{hi}] \);
3. a cycle is permuted if and only if it has a member \( < k \);
4. \( \forall i: k \leq i < \text{hi} \)
   \[ \text{mark}[i] \Rightarrow \text{cycle}(i) \text{ permuted}; \]
   \[ \text{not mark}[i] \Rightarrow \text{cycle}(i) \text{ unpermuted or} \]
   \[ \text{cycle}(i) \text{ has a member } < \text{lo} \]

Figure 4.4.2 gives the new algorithm. The program traces the sequence \( k, f(k), f^2(k), \ldots \) etc. until

1. the sequence encounters an index \( < k \)
2. the sequence encounters a marked location
3. the sequence returns to \( k \) without ever going below \( k \) or encountering a marked location.

In case (3) the cycle is un-permuted and is traced again to perform the necessary swaps. Note the essential use of the sequential logical operator \( \text{and} \) which does not evaluate its second operand if the result is known from the value of the first operand.

The discussion of section 5.3 proves that the cost of any one phase is limited to the sum of the lengths of all cycles, which is \( O(n) \). Since there are \( \lceil n/t \rceil \) phases, the total running time is \( O(n^2/t) \).
\( k := 1 \); 
\begin{verbatim}
for \( k \in [n+1] \)
    declare mark array with bounds \( lo = k \) and \( hi = \min(lo+t,n+1) \);
    (\( \forall i : lo \leq i \leq hi \): mark\( i \) := false);
    ((4.4.2) holds vacuously)
    for \( k \leftarrow k+1 \)
        if \( k \geq hi \) and not mark\( k \) +
            \( j := k \);
            do hi := \( j + f(j) \)
            if \( k \leq j \) and \( j < hi \) and not mark\( j \) +
                \( j, \) mark\( j \) := \( f(j), \) true
            od;
        fi;
end; 
\end{verbatim}

((4.4.2))

\textbf{Figure 4.4.2 In-place permutation, second version}
Chapter 3

Specific Algorithms Developed Using the Paradigms

Section 3.1 Writing bounded Turing Machine simulation

Section 3.1.1 Problem statement and preliminary lemmas

Let $M$ be a one-tape deterministic Turing machine. We say that $M$ is $f(n)$ ink-bounded if $M$ changes a symbol of its work tape at most $f(n)$ times while processing any input of length $n$. If $M$ is scanning its work tape and changing state, but not changing symbols on its tape, we say it is silent. Note that an ink-bounded machine could still diverge if it is caught in an infinite silent loop.

Reference [12] discusses INK complexity and demonstrates that this measure is ill-defined for a machine with several tapes if the initial contents of the tapes are all blanks. The machine could mark a zero point on both tapes and then use the head positions as counters to store an arbitrarily large integer and so compute any recursive function with only finite ink. Our discussion is limited to single-tape machines simulated by a single-tape machine. Moreover, we make no assumptions about the initial contents of the tape of the universal machine.

We want to design a universal machine $U$ that given any $M$, as above, and input $x$ can simulate the computation of $M(x)$ or detect that $M$ is diverging. We try to minimize the ink expenditure of $U$.

Next we give three simple lemmas which are used later in the efficiency analysis of $U$. We suppose $M$ to have a one-way infinite tape
and \( r \) states.

Lemma

If \( M \) scans silently more than \( r \) cells past the right end of the nonblank portion of its work tape, it will continue to move right forever.

Proof

\( M \) must repeat a state while scanning a blank and moving right.

Lemma

\( M \) can form nonblank tape contents of length at most \( n+rxf(n) \).

Proof

This follows from the previous lemma since \( M \) can move at most \( r \) steps to the right without printing.

Lemma

If \( M \) moves silently more than \( r(n+rf(n)+r) \) steps it is looping.

Proof

\( M \) must keep its head within the first \( n+rf(n)+r \) cells of the tape. If \( M \) repeats one of its \( r \) states at the same tape cell it must loop since the tape remains unchanged.

Section 3.1.2 The simulation

If \( U \) is to detect \( M \) looping, \( U \) must be able to count \( O(f(n)) \) silent moves of \( M \). This suggests a straightforward quadratic simulation. \( U \) acts like any universal machine and also maintains a count (in unary) of the number of silent moves of \( M \) simulated since \( M \) last printed. If this count ever exceeds \( r(n+rf(n)+r) \) \( U \) knows that \( M \) is looping silently. \( U \) must spend \( O(1) \) ink to simulate one move of \( M \), whether or not \( M \) printed on that move, since \( U \) must also count. In the worst case, \( M \) might move silently \( r(t+r)-1 \) steps between printing, where \( t \) is the length of the
mon-blank portion of $M$'s work tape. Since $t$ can become $O(f(n))$ (at most) and $M$ may print $O(f(n))$ times, this processing may require $O(f(n)^2)$ ink by $U$.

Two observations lead to an improved simulation:

1. if $M$ is scanning silently back and forth over a small portion of the tape, it will soon loop.

2. if $M$ scans silently across most of the tape and then prints, the bulk of the tape is unchanged and $M$ would repeat the same silent computation on the unchanged portion of the tape.

Let $M$ have states $q_1, q_2, \ldots, q_r$. Given a tape segment of length $s$ and $M$ in state $q_i$ scanning the leftmost symbol of the segment there are five possibilities:

1. $M$ scans the segment silently and exits on the right in some state $q_i'$

2. as in (1) but exit on the left

3. $M$ loops silently within the segment

4. $M$ prints within the segment

5. $M$ halts within the segment.

There are the same five possibilities if $M$ starts at the right of the segment.

For any state $q_i$ and tape segment of length $s$ we can determine which of these five cases occurs using at most $O(s)$ ink. This follows from a previous lemma since if $M$ moves more than $m_s$ steps without printing, it must be looping silently within the segment.
Given a segment we can compute the left-entry (respectively right-entry) segment table which records, for each state \( q \), which of the five possibilities above occurs, assuming \( M \) enters the segment from the left (respectively right). These tables are written in space proportional to \( r \), the number of states, and are computed using ink proportional to the length of the segment.

We demonstrate the simulation for a bound of \( O(f(n)^{4/3}) \). This will suffice to illustrate all the major points of our construction. The simulated work tape of \( M \) is partitioned into segments and each segment is delimited by a left-entry and a right-entry segment table. Furthermore, to achieve the claimed bound, this partitioning is applied recursively once again so that segments are partitioned into sub-segments also delimited by segment tables. We refer to the outer segments as "big" segments or "level 1" segments and the inner segments as "little" or "level 2" segments.

Figure 5.1.1 shows a possible tape layout for \( U \). Here \( t < O(f(n)) \) is the length of the non-blank portion of \( M \)'s work tape so far in the simulation. There are \( t^{1/3} \) big segments each of length \( t^{2/3} \). Each big segment is divided into \( t^{1/3} \) little segments of length \( t^{1/3} \) each.

Track 1 holds the simulated tape contents and the segment tables for levels 1 and 2.

Track 2 stores the current state, \( q \), of the simulated machine written in the tape alphabet of \( U \). The length of this state description is some constant "qsize" depending on the number of states in \( M \) and the size of the tape alphabet of \( U \). The universal machine will keep this state description under the segment table currently being accessed.
Track 3 holds a counter written in unary notation. This counter is incremented for every silent segment transition made by M since the last printing move and is used to detect infinite silent looping.

Track 4 holds an encoding of the simulated machine M, again written in the tape alphabet of U. The length of this encoding will be some constant "msize". To begin the simulation, U will compute segment tables for the initial contents of the work tape of M. Then U writes the initial state q₀ under the first segment table using constant ink. Now U consults the segment table:

1. If the table indicates halting, U finishes;
2. If the table indicates silent looping, U finishes;
3. If the table indicates exit in state q’, U writes q’ under the appropriate adjacent segment table;
4. If the table indicates printing, U performs the updating operation described below.

| 2/3 |
|-----|----------------------|
|     |                      |
| 1/3 |                      |
|     | -t ---               |

```
track 1 ... BST LST segment LST segment ... segment LST BST ...
track 2   q
track 3   1111111111...111
track 4   encoding of M
```

Figure 5.1.1 Tape layout of simulating machine
The first three cases require constant ink by U independent of the length of the segment; this is an important implementation point of our construction. In order to manipulate the state information and to carry out the re-configuration described below, U may have to transfer information across long stretches of its work tape. This may require lots of head motion but the ink required is proportional to the length of the string being transferred, not to the distance over which the string is transferred. Figure 5.1.2 shows how U would move the state encoding of M from one end of a segment to the other. Between successive symbols the head of U must move from one end of the segment to the other but this takes no ink.

Suppose M has entered an infinite silent loop that spans segments. If M makes more than \(2rt^{1/3}\) silent segment transitions, U may conclude that M is looping. This bound counts \(r\) states, \(t^{1/3}\) segments, and two ways to enter each segment. Here is the essential efficiency of the construction: long silent operation by M can be simulated by U in jumps of length \(t^{2/3}\), each requiring only constant ink. To detect silent looping, U need only count to \(O(t^{1/3})\) between printing moves by M. Since \(t\) is bounded by \(f(n)\) this counting can cost at most \(O(f(n)^{4/3})\) for the entire simulation.

Suppose M decides to print within a segment. The segment tables then become invalid and must be recomputed requiring as much as \(O(f(n)^{2/3})\) ink. This could happen \(O(f(n))\) times and would be too expensive. Here is where the level 2 segment tables come into play. If M decides to print in a big segment, U simulates the operation of M in this big segment to determine the little segment to be changed. This can be done with only \(O(t^{1/3})\) ink using the tables for the little
\textbf{Figure 5.1.2 Transfer operation}

\begin{itemize}
  \item segments. The symbol is printed and the two tables for the modified little segment are recomputed using \(O(t^{1/3})\) ink. Now the two tables for the containing big segment must be recomputed. However, \(U\) can now consult the up-to-date little segment tables to do this with \(O(t^{1/3})\) ink rather than \(O(t^{2/3})\) ink, since \(U\) can move the head of \(M\) within the segment in jumps of size \(t^{1/3}\). Again, since \(M\) can print at most \(f(n)\) times and \(t\) is at most \(f(n)\) all this updating requires at most \(O(f(n)^{4/3})\) for the entire simulation.
\end{itemize}
After a printing move the (simulated) head of \( M \) is positioned inside some little segment of length \( t^{1/3} \). \( U \) continues the simulation inside of this small segment. If \( M \) specifies a printing move \( U \) updates the segment table for this little segment then updates the tables for the containing big segment. All this takes \( O(t^{1/3}) \) ink as before. If \( M \) moves silently within the little segment more than \( 2t^{1/3} \) steps \( U \) concludes that \( M \) is looping. If the head of \( M \) moves to the end of the small segment \( U \) may start using the little segment tables to simulate the action of \( M \) up to the next printing move. Again, \( M \) may jump across little segments at most \( 2t^{1/3} \) times without printing or looping. If the head of \( M \) finally moves out to the end of a big segment \( U \) resumes the simulation at the top level.

The value of the counter on track 3 was defined as the number of silent moves by \( M \) since the last printing operation. Therefore the counter must be reset to zero whenever \( M \) prints. This is done by erasing all the tally marks and takes ink proportional to the length of the counter. The ink cost for erasing is the same as the ink cost for incrementing the counter so we may charge the cost of erasure against the cost of the incrementation and not change the asymptotic bound for the ink usage of \( U \).

Section 5.1.3 Reconfiguration

We have tacitly assumed above that \( t \) is not changing. However, some of the printing operations of \( M \) may increase the length of the non-blank tape contents as simulated by \( U \). This can be absorbed by a padding segment on the end. However, if too many new segments are added without increasing the length of the segments, the number of
segments may exceed $O(f(n)^{1/3})$. This would ruin the previous analysis. Periodically during the simulation $U$ must reconfigure the simulated tape of $M$. That is, the big segment and little segment tables are cleared, $t^{1/3}$ and $t^{2/3}$ are computed afresh, and new big segments and little segments are marked off. This can all be done by $U$ using $O(t)$ ink.

Suppose the tape is reconfigured every time that it doubles in length from a previous reconfiguration. Then there can be at most $O(\log f(n))$ reconfigurations and the reconfiguring contributes an ink cost of at most $O(f(n) \log f(n))$ for the entire simulation.

Right before a reconfiguration we may have a tape of length $t$ divided into segments of length $(t/2)^{2/3}$. The number of segments is then $t/(t/2)^{2/3} = 2^{2/3}t^{2/3} = O(t^{1/3})$ and there are never too many segments.

Section 5.1.4 The general case

In general, for fixed $k>1$, we can give a $(k-1)$ level construction achieving a bound of $O(f(n)^{1+1/k})$. At the $i$ th level, $U$ maintains segments of length $< f(n)^{(k-1)/k}$. The smallest segments at level $k-1$ are of length $< f(n)^{1/k}$.

Updating the tables for one of these smallest segments costs $O(f(n)^{1/k})$ ink. Then we must work back up to the containing level 1 segment. After updating a segment of length $f(n)^{(k-1)/k}$ we must update the containing segment of length $f(n)^{(k-1+1)/k}$ but this requires only $\lceil f(n)^{(k-1+1)/k} \rceil / \lceil f(n)^{(k-1)/k} \rceil = O(f(n)^{1/k})$ ink since we can now use the up-to-date tables of the smaller segments.

During the simulation, if the head of $M$ is not located inside of some smallest segment it is positioned at some $i$-th level segment table.
Simulation inside a smallest segment requires $O(t^{1/k})$ ink per printing move of M. Within $2rt^{1/k}$ moves M should print or reach one end of the smallest segment; if not U may conclude that M is looping.

When U has the head of M positioned at an i-th level segment table it may simulate silent operation of M in jumps of length $t^{(k-i)/k}$. Within $2rt^{1/k}$ jumps M should print or reach the end of the level i segment; if not U may conclude that M is looping.

Notice that without printing, the (simulated) head of M may only move "up" in levels. Therefore, U need count to at most $k2rt^{1/k}$ or $O(t^{1/k})$ between printing moves of M.

Section 5.1.2 Programming details

Complete details for the entire construction would be tedious. We only mention some points which might be unclear.

U is to reconfigure every time the simulated work tape doubles in length. After a reconfiguration has finished with a tape of length $t$ U should make a mark on the tape $2t$ spaces from the end. If the tape extends to this mark a reconfiguration is needed.

U must compute segment length which are rational powers of $t$. We show how mark off $t^{3/4}$ using $O(t)$ ink. The computation for other rational powers $< 1$ should be clear.

U stores an integer $k$ in unary as a string $\_^k$. U maintains four variables $k$, $k_2$, $k_3$, $k_4$ satisfying:

\[ k_2 = k^2, \quad k_3 = k^3, \quad k_4 = k^4. \]

U executes the loop of figure 5.1.3. Addition is done by concatenating strings and takes ink proportional to the length of the sum. The total
\[
\begin{align*}
&k = 1 \\
k2 := 1 \\
k3 := 1 \\
k4 := 1 \\
\text{while } k4 < t \text{ do} \\
&k := k + 1 \\
k2 := k2 + 2k + 1 \\
k3 := k3 + 3k2 + 3k + 1 \\
k4 := k4 + 4k3 + 6k2 + 4k + 1
\end{align*}
\]

Figure 5.1.3 Reconfiguration computation.

ink used by the loop would be proportional to the length of the longest string formed. This is \( O(t) \). When the loop finishes, \( k \) approximates \( t^{1/4} \) and \( k3 \) approximates \( t^{3/4} \).

**Section 5.1.6 Corollary**

Let \( \text{INK}(f(n)) = \{ L(M) \mid M \text{ is an } f(n) \text{-ink-bounded deterministic machine} \} \).

Define a function \( g: \mathbb{N} \to \mathbb{N} \) to be "ink-constructable" if some deterministic machine \( M \) can mark off \( g^8(n) \) for input of length \( n \) using \( O(g(n)) \) ink.

For any \( \epsilon > 0 \), if

\begin{align*}
&\text{inf}_{n \to \infty} \frac{f(n)^{1+\epsilon}}{g(n)} = 0 \\
&g \text{ is ink-constructable}
\end{align*}

then

there is a language in \( \text{INK}(g(n)) \) not in \( \text{INK}(f(n)) \).
Proof

Our proof uses the standard technique already developed for tape bounded classes [16] and time bounded classes [23]. We give only a sketch.

Construct a universal machine $U_g$ which for input of length $n$ marks off $\#^g(n)$. During subsequent processing if $U_g$ prints a symbol it also erases one $. If U_g ever erases the last $#$ it stops; this assures that $U_g$ is $g(n)$ ink bounded. Now $U_g$ decodes its input $w$ as a machine description "M" and padding $x$ (if the input is not of this form, $U_g$ simply stops). $U_g$ simulates $M(w)$ as above until:

(1) $U_g$ runs out of ink

(2) $M(w)$ loops silently in which case $U_g$ accepts $w$

(3) $M(w)$ halts and accepts $w$ in which case $U_g$ rejects $w$

(4) $M(w)$ halts and rejects $w$ in which case $U_g$ accepts $w$.

Now suppose $L(U_g) = L(H_0)$ where $H_0$ is $f(n)$ ink bounded. For some long enough padding $x_0$, $U_g$ can complete the simulation of $H_0('H_0',x_0)$ and do the opposite. This shows that $L(U_g)$ is not computed by any $f(n)$ ink bounded machine.

Section 3.1.7 Extensions and open problems

One natural extension of this work would be to study non-deterministic ink complexity. It is also possible that our techniques could be used to improve the universal machine to $O( f(n) \log f(n) )$ ink usage but this is not at all straightforward. Passing our construction to the limit where $k = O( \log f(n) )$ requires storing the simulated work
tape as a tree of depth $O(\log f(n))$. Now moving up and down in this
tree is difficult since $U$ cannot keep its place in the recursive
structure by reading one of a fixed number of symbols.

Section 5.2 The Controlled-Density Data Structure

Section 5.2.1 Introduction

A sorted sequence of data values from an ordered universe is
conveniently represented as an array -- the sorted sequence of elements
is the sequence of array values in order of increasing subscript value.
A test for membership of a new element among the members of the
sequence, extracting the minimum or maximum value, or printing the
sequence can be done easily with this representation in optimal time
(respectively: logarithmic, constant, and linear). Unfortunately,
updating the sequence by insertion of a new element or deletion of an
existing element is an expensive operation since the sequence must be
kept packed; the worst case time might be linear in the number of
elements in the array at the time the operation is performed.

Suppose a sequence of $N$ data values is collected on-line and kept
in sorted order in an array $a[1:N]$. At each step there is a sorted
portion $a[1:n]$ and an untouched portion $a[n+1:N]$. Each insertion
consists of a binary search to find the correct location $a[j]$, $1 \leq j \leq n+1$,
for the next element and then shifting the elements $a[j:n]$ up one place
to make room for it. This shifting can cost $O(n)$, leading to an $O(n^2)$
algorithm. Shifting, not searching, is the expensive operation. The

---

1 The results in this section were developed jointly with David Gries
and were first presented as Cornell Computer Science Technical Report
78-365.
data structure we discuss in this section allows binary search just as
with the packed array representation. Moreover, insertion of a new
element is much cheaper. We present the data structure in the context
of a sorting algorithm. If necessary, the sort can operate on-line.
However, to simplify the presentation we assume that a total of \( N \) values
are to be sorted and these values initially are in locations \( a[0:2N+1] \)
of an array \( a[0:2N+1] \). To reduce the amount of shifting, we introduce
\( \text{gapa} \) into the array; empty entries into which new elements can be
inserted. For example, suppose at one point the already sorted values
\( v_1, v_2, \ldots, v_n \) appear in \( a[1:m] \), where \( m = 2n \), with exactly one gap
before each value:

\[
gap, v_1, \text{gap}, v_2, \ldots, \text{gap}, v_n
\]  
\[ (5.2.1) \]

With a suitable encoding of gaps (described below) the place \( a[j] \) to put
\( v_{n+1} \) can still be determined in time \( O(\log n) \) using binary search. But
insertion of \( v_{n+1} \) requires no shifting since it can fill a gap.
Likewise, insertion of \( v_{n+2}, v_{n+3}, \ldots \) will be cheap. This is similar to
hashing with linear probing[20, pp506-550]. Provided the table does not
get too full (i.e., provided not too many gaps are used up) and given a
uniform distribution of values, only a small amount of shifting is
needed to insert a new value.

At some point, however, the table will become full and shifting
will become costly. At this point the array is configured to arrange
the sorted values into form (5.2.1) with one gap before each value. An
outline of the algorithm is shown in figure 5.2.1.

The problem is to determine what "not too full" means. The balance
between inserting and configuring will determine the worst-case cost of
initialize \(a[1:m]\) to contain \(V_1\) and have form (5.2.1);
\[
\text{do } n := N + 1
\]
\[
\text{do } \text{table } a[1:m] \text{ is not too full and } n := N + 1
\]
\[
\text{insert } V_n \text{ into } a[1:m]
\]
\[
\text{od}
\]
\[
\text{if } n = N \Rightarrow \text{configure } \square n = N \Rightarrow \text{skip } \square
\]
\[
\text{od}
\]

Figure 5.2.1

the algorithm. We analyze this problem in section 5.2.2. In section 5.2.3 we give the algorithm in detail and in section 5.2.4 we extend the data structure to support deletions and compare its performance to a balanced tree scheme [1, Ch.4].

Section 5.2.2 Worst-case time analysis

The algorithm outlined above contains three major parts: searching \(a[1:m]\) to determine the location \(a[j]\) where a new value \(x\) should be placed; inserting \(x\), including shifting values \(a[j], a[j+1], \ldots\) up one position to make room for \(x\); and configuring when \(a[1:m]\) becomes too dense.

Let us consider inserting and configuring together since their costs are related. The following theorems prove worst-case bounds for inserting and configuring using several different schedules for configurations.

Theorem 5.2.1: Let \(n_i\) be the number of values in the table just after the \(i\)-th insertion, with \(n_0 = 1\). Suppose that \(c \sqrt{n_i}\) values are inserted between the \(i\)-th and the \((i+1)\)-st configurations for some fixed \(c > 0\):
\[ n_0 = 1, \quad n_{i+1} = n_i + c\sqrt{n_i} \quad \text{for } 0 \leq i \]

where the square root is truncated to an integer. Then the total worst-case times for configuring and inserting are both \(O(N \sqrt{N})\).

**Proof:** Let there be \(c\) configurations, so that \(n_c < N \leq n_c + c\sqrt{n_c}\). Each \(n_i\) satisfies \(n_i < N\). The number of values inserted is \(N\) where:

\[
N > 1 + \sum_{i=1}^{c} n_i > c + \frac{c}{\sqrt{n_i-1}} > c \frac{c}{\sqrt{n_i-1}}
\]

The total number of values moved during all configurations is

\[
\sum_{i=1}^{c} \frac{c}{\sqrt{n_i-1}} = \frac{c}{\sqrt{n_1-1}} + \frac{c}{\sqrt{n_2-1}} + \ldots + \frac{c}{\sqrt{n_c-1}} < N + \frac{c}{\sqrt{n_c-1}} < N + N^{1/2} N/c.
\]

Hence the worst-case cost of all configuration is \(O(N \sqrt{N})\).

The worst-case time for inserting occurs when the maximum number of shifts is made at each step. Between the \(i\)-th and \((i+1)\)-st configuration this is:

\[
0 + 2 + \ldots + (c\sqrt{n_i}) < (c\sqrt{n_i})(c\sqrt{n_i+1}) = c^2 n_i + c\sqrt{n_i}
\]

shifts. The total number of shifts over all insertions is thus bounded by

\[
\sum_{i=0}^{c} (c^2 n_i + c\sqrt{n_i}) = c^2 \sum_{i=0}^{c} n_i + c \sum_{i=0}^{c} \sqrt{n_i} < c^2 + c^2 N \sqrt{N} N + cN.
\]

Hence the worst case for insertions is \(O(N \sqrt{N})\). Q.E.D.
Theorem 1.2.2: Let \( n_i \) be the number of values in the table just after the \( i \)-th configuration, where \( n_0 = 1 \). Suppose for some fixed \( \alpha \), \( 0 < \alpha < 1 \), that \( \alpha n_i \) values are added between the \( i \)-th and \((i+1)\)-st configurations:

\[
n_0 = 1, \quad n_{i+1} = n_i + \alpha n_i = (1+\alpha)^i \text{ for } 0 \leq i.
\]

Then the worst-case total times for configuring and inserting are proportional to \( N \) and \( N^2 \) respectively.

**Proof:** Let there be \( e \) configurations so that \( n_e < N < n_e + \alpha n_e \). Thus \( (1+\alpha)^e < N < (1+\alpha)^{e+1} \). The total number of values moved during all configurations is

\[
\sum_{i=0}^{e} (1+\alpha)^i = \frac{(1+\alpha)^{e+1} - 1}{(1+\alpha) - 1} < \frac{N}{\alpha}.
\]

Hence all configuring takes linear time.

The worst-case for inserting occurs when the maximum number of shifts is made at each step. This happens, for example, when the values to be processed are in reverse sorted order. Given \( n_i \) values, with a gap between each pair, to insert \( n_i \) values takes at most

\[
0 + 2 + 4 + \cdots + (2n_i - 2) < \alpha n_i^2 - \alpha n_i
\]

shifts. The total number of shifts over all insertions is bounded by

\[
\sum_{i=0}^{e} \frac{(\alpha^2 n_i^2 - \alpha n_i)}{\alpha + 2} < \frac{(1+\alpha)^2 - 2}{\alpha + 2} \leq \frac{2(1+\alpha)^2 - 2}{\alpha + 2} \leq \frac{(1+\alpha)^{1+1} - 1}{(1+\alpha)^2 - 1} < \frac{N^2}{\alpha + 2}. \]

Hence the insertions in the worst-case take \( O(N^2) \) time. Q.E.D.
Theorem 5.2.1: Let the table contain \( m \) entries and \( n \) data items (so there are \( m-n \) gaps) and suppose a configuration is performed whenever an insertion requires more than \( c \sqrt{m} \) shifts or \( n \) becomes greater than or equal to \( \alpha m \) where \( c > 0 \) and \( \alpha, 0 < \alpha < 1 \) are parameters of the algorithm. Then the worst-case times for inserting and configuring are both \( O(N\sqrt{N}) \).

Proof: We consider first the cost of configuring according to this strategy. A configuration may be of two types corresponding to the two clauses of the scheduling condition stated above. Call a configuration type I if it is triggered by an insertion requiring more than \( c \sqrt{m} \) shifts and type II if it is caused by \( n \geq \alpha m \).

Let there be \( e \) configurations of type I and let \( n_i \) denote the value of \( n \) at the time of the \( i \)-th type I configuration. Immediately after the \( i \)-th type I configuration, \( m = 2n_i \). The \((i+1)\)-st type I configuration requires \( c \sqrt{m'} \) values in a row generated by \( \frac{c}{2} \sqrt{m'} \) adjacent insertions and \( m' \geq m \) therefore \( n_{i+1} \geq n_i + \frac{c}{2} \sqrt{2n_i} \).

Rearranging, \( n_{i+1} - n_i \geq \frac{c}{2} \sqrt{2n_i} \) for \( 1 \leq i < n \).

Observe:

\[
N \geq \sum_{i=1}^{e-1} n_{i+1} - n_i \geq \sum_{i=1}^{e-1} \frac{c}{2} \sqrt{2n_i}.
\]

The total cost of type I configurations is

\[
\delta_I = \sum_{i=1}^{e-1} n_i \leq N + \sum_{i=1}^{e-1} \frac{c}{2} \sqrt{N \sqrt{n_i}}.
\]

Now for some suitable constant \( d \):
\[ N + \sum_{i=1}^{e} \sqrt{N \cdot d_i} \leq N + d \sum_{i=1}^{e-1} \frac{c}{2\sqrt{2d_i}}. \]

Substitute inequality (5.2.2):

\[ S_1 \leq N + d \sqrt{N \cdot N} = O(N \sqrt{N}). \]

We estimate the cost of the type II configurations informally. Between successive type II configurations, \( m \) must increase by at least some fixed fraction determined by \( \alpha \). In other words, the size of the array right before type II configurations is increasing at an exponential rate. Therefore there can be at most \( O(\log N) \) type II configurations at a total cost of not more than \( O(N \log N) \).

The bound on inserting is determined as follows. Suppose \( e \) configurations are performed and at configuration \( t \) let \( i_t \) be the number of insertions since the previous configuration or the start of the algorithm as appropriate. After \( i_{t-1} \) insertions the longest possible gap free sequence in the array has length \( 2i_{t-1} \), so that the insertion causing configuration \( t \) can cost no more than \( 2i_t \) shifts. Hence the insertions causing configurations can require no more than

\[ 2i_1 + 2i_2 + \cdots + 2i_e \leq 2N \]

shifts. Each of the \( N-e \) insertions that does not trigger a configuration needs less than \( c\sqrt{m} \) shifts and \( m \leq 2n \leq 2N \) so the total for \( N \) insertions is \( \leq O(N \sqrt{N}) \). Q.E.D.

The scheduling strategy analyzed in the last theorem above is the one used in our program. For random input, most of the configurations will be of type II and the configuration cost is small. However, the program still enjoys the worst-case bound of theorem 5.2.1.
Section 3.2.1 The complete sort algorithm

The input to the algorithm is an array \( a[0:2*N+1] \) where \( N \geq 1 \) and the \( N \) values to be sorted \( V_1, \ldots, V_N \) are in locations \( a[2N+1:2*N] \). Upon termination:

\[ a[1:m] \text{ contains a sorted permutation of } \{V_1, \ldots, V_N\} \quad (3.2.2) \]

together with up to \( N \) gaps and \( m \leq 2N \)

The sorted sequence without gaps can be recovered easily in linear time.

To facilitate searching, a gap contains the value of the array element to its right. The algorithm uses a boolean function \( \text{isgap}(i) \) with the meaning "\( a[i] \) is a gap" for \( 0 \leq i \leq 2N+1 \). This is only one possible implementation. If the \( V_i \) are positive integers, the sign bit of each element can be used to mark a gap. If the \( V_i \) are known to be all distinct then no extra storage is necessary and \( \text{isgap}(i) \) is equal to \( i > m \text{ xor } (i = m \text{ and } a[i] = a[i+1]) \). The requirements of a gap encoding are that gap values be disjoint from the data values and that the value of a gap can be determined in constant time.

Besides variables \( m \) and \( n \), the algorithm uses a variable \( s \), which gives the number of shifts required by the last insertion and which is used to determine when to configure. The padding elements \( a[0] \) and \( a[m+1] \) eliminate special tests on boundary conditions.

Invariant (3.2.3) describes precisely the meaning of the program variables.
1 ≤ n ≤ k and n ≤ m ≤ 2k and even(m);
\{v_1, \ldots, v_n\} ⊆ a[1:m] and \{v_{1+1}, \ldots, v_{n+k}\} = a[k+1:2k];
(∀i: 0 ≤ i < n/2: i = \text{isgap}(2i)) and \text{isgap}(m+1) and \(5.2.3\)
\(a[1:m]\) contains \(m-n\) gaps;
(∀i: 1 ≤ i ≤ m: a[i] ≤ a[i+1]);
\(s\) = number of shifts needed in most recent insertion

The algorithm itself is given in figure 5.2.2.

The operation "insert \(x\) into \(a[1:m]\)" shown in figure 5.2.3, has three main steps. First the position \(j\) where \(x\) belongs is determined using local variables \(j, h\) and \(e\). The invariant for the binary search is:

\[1 ≤ h < j ≤ m \text{ and } a[h] ≤ x ≤ a[j]\]  \(5.2.4\)

and the loop terminates with \(a[j-1] ≤ x < a[j]\). This means that upon termination \(a[j-1]\) is not a gap and that \(x\) belongs in \(a[j]\).

The second step shifts the sequence \(a[j..]\) upwards, changing \(x\) as it progresses, until a gap is found. At the same time the number of shifts required is counted. The final step stores \(x\) in its place. Note that if \(x\) belongs after \(a[m]\), \(m\) is increased by 2 to include both the new value and a new gap.

The configuration operation is performed by the program of figure 5.2.4. The values in \(a[1:m]\) are spread out into \(a[1:2\times n]\) so that one gap precedes each value. The invariant (5.2.5) of the loop is given below. The last line of (5.2.5) indicates that \(a[j+1:2\times n]\) contains a gap at each odd position and a value \(V_i\) in each even position. Termination is proved by observing that \(i\) decreases by at least one at each iteration. If \(i=j\) then \(a[1:i]\) must contain a gap before each value
\begin{verbatim}
s[0],s[1],s[2]: a[n+1],a[n+1],a[n+1];
\text{isgap}(0),\text{isgap}(1),\text{isgap}(2),\text{isgap}(3): = \text{false,true,false,true};
m,s:=1,2,0;
do n < N + if n < 3*m/4 and s^2 < 2*m + skip
\square n \geq 3*m/4 or s^2 \geq 2*m + configure
\text{fi};
n:= n+1;x:= a[N+n];
Insert x into a[1:m] to re-establish (5.2.3)
ed
\end{verbatim}

Figure 5.2.2 Sort algorithm.

Insert x into a[1:m] to re-establish P:
Find the position a[j] to place x:
\text{if } a[m] \leq x + j:= m+1
\square a[m] > x + j:= 1
\square a[1] \leq x and x < a[m] +
Find j such that a[j-1] \leq x < a[j] by binary search:
h, j:= 1,m;
do h+1 \leq j \rightarrow e:= (h+ j)/2;
\text{if } a[e] \leq x + h:= e
\square a[e] > x + j:= e
\text{fi}
ed
\text{fi}:
Shift upwards:
s:= 0; do not isgap(j) + a[j].x:= x,a[j]; j,s:= j+1,s+1 ed;
Insert x at a[j] -- note that isgap(j) and j=m:
\text{if } j < m + a[j].isgap(j): = x, false
\square j > m + m:= m+2;a[m-1],a[m]: = x,x;
isgap(m),isgap(m+1): = false,true
\text{fi}

Figure 5.2.3 Insertion

and hence a[1:2n] is configured. Note also that i=0 implies that a[j+1:2n] contains n values and n gaps, so that j=0.
\[ 0 \leq i \leq j \leq 2n \text{ and even}(j); \]
\[ \text{not isgap}(0) \text{ and not isgap}(i); \]
\[ a[1;i] \text{ and } a[i+1;2n] \text{ contains } V_1, \cdots, V_n \text{ sorted} \]
\[ \text{together with } n-(j-i) \text{ gaps}; \]
\( (\forall k: j<k\leq 2n \text{ and even}(k): \text{isgap}(k-1) \text{ and not isgap}(k)) \]

---

**Figure 5.2.4 Configuration**

---

**Section 5.2.4 Extensions of the Controlled-Density Structure**

The algorithm of section 5.2.2 can be modified to delete as well as insert values. This yields an algorithm that supports INSERT, DELETE, SEARCH, and MIN as defined in [1].

Care is required to maintain gaps, because there may be more than one gap between adjacent values after a deletion. Let us represent a gap by the closest real value to its right, so that each gap sequence looks like:

\[
\langle y \cdots y \cdots y \cdots y \cdots y \cdots \rangle
\]

- isgap, isgap, isgap, \cdots, isgap, \neg \text{isgap}
Insertion causes no problem since it changes only a leftmost gap into a value. To maintain gaps during a deletion, however, all the gap values to the left of the value being deleted must be changed. Hence configuring should take place whenever a gap sequence of length $\sqrt{2m}$ or a value sequence of length $\sqrt{2m}$ is detected. Note that configuring is more complicated because it could cause the portion $a[1:m]$ to shrink.

The data values in the controlled-density array occur in nondecreasing order at successive array locations. Therefore, Interpolation Search[24] can be carried out on the array. It is tempting to claim that Interpolation Search will give an average case search time of $O(\log \log n)$ when applied to the controlled-density array. However, the running time analysis of [24] may be disturbed by multiple copies of the same value once as data and once as a gap. We leave as a problem to prove or disprove that Interpolation Search runs in average time $O(\log \log n)$ per search on the controlled-density structure.

The technique of interspersing gaps with data values can be extended to a recursive construction with some fixed number $k$ (say) of levels. The resulting data structure supports INSERT, DELETE, SEARCH and MIN in worst-case time $O(N^{1+1/k})$ for $N$ operations. The data structure is adaptive so that some sequences of $N$ operations may require only $O(N)$ time. We describe a workable implementation of this data structure and give an informal worst-case running time analysis. The practical value of these algorithms for $k > 2$ or $3$ is questionable.

The data structure consists of a one dimensional array $A$ containing data and gaps, together with $k-1$ arrays of indices into $A$. The array $A$
is partitioned into segments, sub-segments, etc., with the array itself regarded as a level 0 segment. For \( n \) data values and \( 0 \leq d < k \) a level \( k \) segment satisfies:

1. the level \( k \) segment has \( n^{1/k} \) data areas alternating with \( n^{1/k} \) gap areas;
2. a gap area has length \( \leq n^{(k-d)/k} \);
3. a data area is of length \( \geq (k-d)n^{(k-d)/k} \) and contains \( n^{1/k} \) level \( d+1 \) segments;
4. the beginning of the \( i \)-th data area is indexed by \( da^{(d)}[i] \) and the beginning is the \( i \)-th gap area is indexed by \( ga^{(k)}[i] \).

The smallest segments on level \( k \) are contiguous sequences of data values of length \( n^{1/k} \). Note that the sizes of the segments are balanced so that the number of sub-segments in a segment is always \( n^{1/k} \). The choice of constant ratio across all levels is optimal for this arrangement of segments.

As in the earlier algorithm the data area is sorted and a gap is a marked copy of the value following it. A SEARCH operation, then, can be done in time \( O(\log kn) = O(\log n) \) by binary search. An INSERT operation determines by SEARCH the proper location for the new item, then searches forward for the closest gap -- by the arrangement of the data structure this takes time at most \( O(n^{1/k}) \). If the insertion of the item does not cause a level \( k-1 \) gap area to be filled, the insertion is finished; if a level \( k-1 \) gap area is filled, the level \( k-1 \) segment containing the new item is configured using \( n^{1/k} \) space from the gap area of the containing level \( k-2 \) segment. If taking space from the level \( k-2 \) gap area uses up this gap area the the level \( k-2 \) segment is configured using space from
the containing level k-3 gap area, etc. All this manipulation references and modifies the pointer arrays da and ga.

A configuration at level d costs $O(N(k-d)/k)$. In a sequence of N operations a configuration at level k occurs no more often than once every $O(N(k-d-1)/k)$ steps, so it can happen at most $O(N/k) = O(N(k-d-1)/k)$ times.

The total cost for N operations is then

$$\leq O(N \log(kN) + N^{1/k}) + \sum_{d=0}^{k-1} O(N(k-d)/k)$$

$$\leq O(N^{1/k}) + O(kN^{1+1/k}) = O(N^{1+1/k})$$

As discussed earlier, deletion can be programmed in several different ways, but any reasonable choice should not disturb the asymptotic analysis.

Section 5.2.3 Empirical results

Versions of our algorithm were compared with Heapsort, Quicksort and a balanced tree scheme taken from reference[13]. The results of this comparison are summarized in the tables of figure 5.2.5. All programming was done in the "C" programming language running under UNIX on a PDP-11. The times reported here were obtained with the UNIX "time" command and should be accurate to a resolution of about 0.1 seconds. Test data described as "random" was generated with the UNIX library "rand" function. Each result reported for random input represents an average over ten trials.

As described above, the controlled-density algorithm schedules configurations by two conditions:
(1) density of data in array $a$

(2) shift or length longer than square root of array size

where $a$ is a parameter of the algorithm. The storage required is always $\leq 2N$ but is typically less -- the maximum amount of storage used is called $M$ and is reported along with the running times.

For random input data, Quicksort is hard to beat. The sorted input was designed to force quadratic performance from the Quicksort program and in this case controlled-density sorting is a bit faster. The asymptotically superior running time of controlled-density sorting for this case would be more apparent for larger input. As is to be expected, the non-adaptive Heapsort algorithm runs in more or less the same time on both sets of inputs. The interesting comparison is between the balanced tree and the controlled-density data structure. Even when the controlled-density structure is forced to worst-case performance, it is still much faster than the balanced tree. To compare storage requirements, let $D$ be the amount of storage needed to store a data item, $G$ the amount of storage needed to store a gap and $P$ the storage required for a pointer in the tree. The balanced tree will take $N*D + 2N*P$ storage plus any additional storage for balance information. The controlled-density array requires at most $N*D + N*G$ storage. Since $G$ and $P$ will be about equal in reasonable implementations we conclude that the controlled density structure takes only about $2/3$ the storage of a balanced tree. Finally, the simplicity of programming the controlled-density structure make it attractive for in-core table management.
Sorting Algorithms

<table>
<thead>
<tr>
<th>N</th>
<th>Random Input</th>
<th>Sorted Input</th>
<th>Heapsort</th>
<th>Quicksort</th>
<th>Controlled Density a = 2/3</th>
<th>a = 3/4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>time</td>
<td>time</td>
<td>time</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>5000</td>
<td>4.8</td>
<td>0.9</td>
<td>1.7</td>
<td>9234</td>
<td>1.7</td>
<td>8264</td>
</tr>
<tr>
<td>7000</td>
<td>7.8</td>
<td>1.3</td>
<td>2.4</td>
<td>11813</td>
<td>2.3</td>
<td>12789</td>
</tr>
<tr>
<td>5000</td>
<td>5.2</td>
<td>11.5</td>
<td>10.2</td>
<td>9902</td>
<td>10.3</td>
<td>9898</td>
</tr>
<tr>
<td>7000</td>
<td>8.6</td>
<td>22.8</td>
<td>16.9</td>
<td>13808</td>
<td>17.1</td>
<td>13804</td>
</tr>
</tbody>
</table>

Balanced Tree Comparison

<table>
<thead>
<tr>
<th>N=4000</th>
<th>Balanced Tree</th>
<th>Controlled Density</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td>Random Data</td>
<td>8.8</td>
<td>2.0</td>
</tr>
<tr>
<td>Sorted Decreasing</td>
<td>13.0</td>
<td>9.0</td>
</tr>
<tr>
<td>Sorted Increasing</td>
<td>13.1</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Figure 5.2.5 Empirical results

Section 5.3 A Time-Space Tradeoff for In-place Array Permutation

Section 5.3.1 Problem statement

Let \( a[1:n] \) be an \( n \) element array (of any type) with initial values \( A_1, A_2, \ldots, A_n \) and let \( f \) be a permutation of the integers between 1 and \( n \). To permute \( a \) by \( f \) means the following:

Initially:
\[
a[1] = A_1, a[2] = A_2, \ldots, a[n] = A_n
\]

After permutation the array should contain:
\[
a[1] = A_{f(1)}, a[2] = A_{f(2)}, \ldots, a[n] = A_{f(n)}
\]

This operation can be described neatly as the concurrent assignment:
\[
a[1], a[2], \ldots, a[n] := a[f(1)], a[f(2)], \ldots, a[f(n)].
\]

However, a concurrent assignment statement is usable unless the value of \( n \) is known at compile-time.
Section 5.1.2 Alternative solutions

Permuting the array is easy if a second array b[1:n] is available as temporary storage. Move all a[i] to b[f(i)] then copy b back into a. However, 2n storage may be excessive if individual array entries are large. An "in-place" permutation algorithm manipulates the array only by interchanging pairs of array entries. We denote the interchange of a[i] with a[j] by "swap(i,j)". Any number of swap operations can be performed with only a single additional storage location for array elements.

Two algorithms for in-place permutation are well known. Call these "permute1" and "permute2". Algorithm "permute1" requires an additional bit array of length n+1 and performs the permutation in time $O(n)$. Algorithm permute2 requires only a fixed amount of additional storage but has worst-case running time $O(n^2)$. The contribution of this section is to show permute1 and permute2 as limit cases of a continuous time/space tradeoff. We give a new algorithm, permute3, which uses a bit array of length $t$ and achieves worst-case time $O(n^2/t)$. Algorithms permute1 and permute2 are obtained by setting $t = n$ and $t = 1$ respectively. Reference [21] describes permute2 in detail and analyzes the average case running time. Reference [10] discusses a variation of permute2 that also has a quadratic worst-case.

Algorithm permute1 is given below. The array is permuted by tracing the cycle structure of $f$. A cycle $i, f(i), f(f(i)), \ldots, f^s(i) = i$ is easily permuted in-place by $s-1$ swaps. The loop labeled 'permute' permutes an entire cycle and marks all entries of the cycle. At the conclusion of this loop any unmarked entry is a member of an
unpermuted cycle. The algorithm terminates when all entries are marked, implying that all cycles are permuted. An invariant of the outer loop would be:

\[ (\forall 1 \leq i \leq n: \text{mark}[i] \iff \text{the } f \text{ cycle containing } i \text{ is permuted} ) \]  

Algorithm permute2 also permutes an entire cycle with the second inner loop but the test for an unpermuted entry is more expensive since there is only a fixed amount of additional storage. The outer loop invariant for this algorithm is:

\[ (\forall p: 1 \leq p < k: \text{cycle containing } p \text{ is permuted and any permuted cycle has a member } < k ) \]  

Therefore \( k \) is a member of a permuted cycle just if \( f^s(k) < k \) for some \( s > 0 \). This test is performed by the loop labeled "test". This algorithm

\[
(\forall j: 1 \leq j \leq n+1: \text{mark}[j] = \text{false}); \kern1em k := 1; \kern1em \text{do } \text{mark}[k] + k := k+1 \kern1em \text{if } \neg \text{mark}[k] \text{ and } k \leq n+1 \kern1em \text{Permute and mark cycle containing } k; \kern1em j, f j := k, f(k); \kern1em \text{permute}; \kern1em \text{do } f j \neq k \kern1em \text{swap}(j, f j); \kern1em \text{mark}[f j] := \text{true}; \kern1em j, f j := f j, f(f j) \kern1em \text{od}; \kern1em \text{mark}[k] := \text{true} \kern1em \text{od}
\]

Figure 5.3.1 Algorithm permute2
\( k := 1; \)
\( \text{do } k := n+1 \rightarrow \)
\( \quad j := f(k); \)
\( \quad \text{test: do } j > k \rightarrow j := f(j) \quad \text{end}; \)
\( \quad \) if \( j < k \rightarrow \text{skip (cycle containing } k \text{ already permuted )} \)
\( \quad \) then \( j := k \rightarrow \text{permute:} \)
\( \quad \quad \text{do } f(j) := k \rightarrow \text{swap}(j, f(j)); \)
\( \quad \quad j := f(j) \)
\( \quad \text{end} \)
\( \quad \)fi;
\( \quad k := k + 1 \)
\( \)end

Figure 5.3.2 Algorithm permute2

has worst-case complexity \( O(n^2) \) since the loop labeled "test" may cost \( O(n) \) for each of \( O(n) \) iterations of the outer loop.

Algorithm permute3 given below represents a compromise between permute1 and permute2. It uses a bit array of length \( t+1 \) where \( t = t(n) \) can be any integer between 1 and \( n \). Like permute2 our algorithm advances an index variable \( k \) such that all cycles with a member \( < k \) are permuted. The outer loop invariant is the same as for permute2:

\[(\forall p: p < k; \text{ cycle containing } p \text{ is permuted and any permuted cycle has a member } < k). \quad (5.3.3)\]

The computation of the outer loop is broken into phases, each of which allocates a mark array of length \( t+1 \) with bounds 0...hi. The mark array will be used to indicate the status of locations \( k, k+1, \ldots, \) \( hi = \min(k+t, n+1) \), however, the meaning of the mark array is a bit different than in permute1. If a location \( j \) is unmarked we cannot conclude that location \( j \) is unpermuted (i.e. does not have its desired
final value). To determine if \( k \) is a member of a permuted cycle, the algorithm traces the sequence

\[ k, f(k), f^2(k), f^3(k), \ldots \]

until a marked location or a location with index \( < k \) is encountered. Cycles are disjoint, so if the cycle containing \( k \) is currently unpermuted the sequence must return to \( k \) without encountering a marked location or a location with index \( < k \). If the sequence returns to \( k \) the required swaps are performed to permute the new cycle. Moreover, to avoid quadratic performance, the locations visited during this tracing are marked so that they will not be inspected again. The invariant for the middle loop is:

1: \( 1 \leq lo \leq k \) and

the mark array has bounds \( 1o \ldots hi \) where \( hi = \min(lo+t, n+1) \);

\( \text{not mark}[k] \) and \( \text{not mark}[hi] \);

a cycle is permuted if and only if it has a member \( < k \);

\( \forall i: k<i<hi: \)  

\[ \text{mark}[i] \Rightarrow \text{cycle}(i) \text{ permuted}; \]

\( \text{not mark}[i] \Rightarrow \text{cycle}(i) \text{ unpermuted or} \)

\( \text{cycle}(i) \text{ has a member } < lo \)

To show the claimed running time we show that the cost of any one phase is at most \( O(n) \). All iterations of the first guarded command, \( \text{alpha} \), cost at most \( O(n) \). We evaluate the cost of the iterations of the second guarded command, \( \text{beta} \), by "charging" parts of the computation to cycle members. Suppose a portion of a cycle is traced first by \( \text{beta} \) starting at some point \( k_0 \). Since each point of the cycle has a unique predecessor, retracing the same portion of the cycle from another starting point \( k_1 \) would mean passing through \( k_0 \). This is
k := 1;
do k := k + 1 +
   allocate mark array with bounds lo = k and hi = \min(10 + t, n + 1);
   (\forall p : lo \leq p \leq hi: mark[p] := \text{false});
middle loop:
do mark[k] := alpha; k := k + 1
   if not mark[k] and k < hi +
      beta:
         j := k;
         innermost loop:
            do hi \leq j +   j := f(j)
               if k < j < hi and not mark[j] +
                  j, mark[j] := f(j), true
               od;
            if k = j + skip
               if k < j + permute:
                  do f(j) \neq k + swap(j, f(j));
                     j := f(j)
                  od
            fi
         od
od

Figure 5.3.3 Algorithm permute3

impossible since location k0 would be marked. Since a portion of a
cycle is never traced twice by different iterations of beta, all
iterations of beta can cost no more than the sum of all cycle lengths.
This is $O(n)$. Finally, the outermost loop iterates $n/t$ times so the
total running time of the algorithm is $(n/t) \times O(n) = O(n^2/t)$. 
Section 5.4

Two-dimensional Turing Machines with uninitialized workspaces

Section 5.4.1 Problem Statement

The conventional two-dimensional TM model[16] provides an infinite two-dimensional workspace assumed to be blank at the start of the computation. In other words, the machine assumes that any cell not previously visited in a computation contains some distinguished symbol b, an element of the workplane alphabet. As an example of a computation making essential use of this assumption, consider the two-dimensional self-crossing problem. The machine processes on-line an input sequence over the alphabet \{N,S,E,W\}. Interpreting each symbol as a unit step in the corresponding compass direction, the machine must report whenever the path so described "crosses" itself, i.e. when the current endpoint of the path coincides with some previous point of the path.

A solution by a two-dimensional TM is easy if we assume the workplane to be blank initially. The input alphabet of the machine is \{N,S,E,W\} and the workplane alphabet is \{b,1\}. The machine reads input symbols and moves its head accordingly. If a move is to a blank cell, the machine prints a 1; if the move is to a cell containing a 1, the machine reports a self-crossing. The time to process n input symbols is clearly O(n).

We criticize this solution as being too simple -- at least it is not quite fair to assign this algorithm only linear running time. Suppose the sequence of the first n/2 input symbols is \((NE)^{n/4}\). The machine records a path as in figure 5.4.1. The second half of the input could cause a self-crossing at any of the O(n²) cells inside the dotted
square, so these cells must be initially blank. A physical device implementing two-dimensional storage would have to initialize these cells to blank before they are visited. This observation casts some doubt on the assignment of only linear running time to the simple solution given above.

We propose a more realistic model for multi-dimensional TM storage: a cell of the workspace not previously visited during the computation may contain any symbol from the workspace alphabet. This stipulation invalidates the simple solution to the self-crossing problem since the machine could now get confused reading spurious 1's.

If a (multi-dimensional) Turing Machine M reading input data I performs exactly the same computation on I for any initial contents of its workspace, we say the machine is initialization-independent. The construction given below shows that two-dimensional Turing Machines may be made initialization-independent with only a slight loss of speed. A

![Figure 5.4.1](image-url)
computation of \( n \) steps in a blank workplane is simulated in time \( O(n \log n)^r \) for some fixed \( r \). Schnorr[26] calls bounds of this form "quasi-linear" and we propose the notation \( Q(f(n)) \) to denote \( O(f(n)(\log n)^r) \) for any fixed \( r \). In particular, arithmetic operations on integers of size polynomial in \( n \) can be done by a TM in time \( Q(1) \), analogous to the \( O(1) \) cost for such operations on a RAM[1,p12]. We restate our result in this notation:

**Theorem 1.4.1:** The first \( n \) steps of the computation of a one-head two-dimensional Turing Machine can be simulated in \( Q(n) \) steps by an initialization-independent two-dimensional machine with two heads.

**Section 1.4.2 The Construction**

The only difficult part of the simulation is the representation of the simulated workplane and head in uninitialized storage. I/O manipulation and state transitions can be implemented in any of several ways, all affecting the running time by only a multiplicative constant.

In the sequel we speak as if \( n \), the number of steps to be simulated, is known in advance. This is not possible in an on-line simulation, but we can achieve the same effect by the following device. For simulation steps \( 2^t \ldots, 2^{t+1}-1 \) the machine uses the construction with \( n = 2^{t+1} \). The simulating machine also records the history of operations modifying the simulated workspace. Before simulating step \( 2^{t+1} \), the machine will spend time \( Q(2^{t+1}) \) to repeat steps \( 1.2 \ldots, 2^{t+1} \) but with \( n = 2^{t+2} \). Now steps \( 2^{t+1} \ldots, 2^{t+2}-1 \) can be simulated on the updated workspace representation. Since \( 2^{t+2} - 2^{t+1} = 2^{t+1} \) we can charge the cost of the updating against the cost of simulation steps \( 2^{t+1} \ldots, 2^{t+2} \) and the simulation is slowed down by only a constant
Let \( n = 2^t \). The simulating machine will organize its workplane as a grid of records, each record having two fields able to store an integer in the range 1...t. Therefore a record is \( \sqrt{2} \log t \) on a side and the whole workplane is magnified by this factor. A record occupying locations \( \{ i \sqrt{2} \log t, ... , (i+1) \sqrt{2} \log t \}, \{ j \sqrt{2} \log t, ... , (j+1) \sqrt{2} \log t \} \) is said to have coordinates \((i, j)\). The simulating machine will obey the following conventions:

1. A symbol \( \sigma \) written by the simulated machine at location \((x, y)\) on its workplane will be stored in record \((2x+1, 2y+1)\) of the simulating machine—that is, a whole record is used to contain a single symbol. Call these "data records".

2. A record with one odd and one even coordinate is called a border record; it is either uninitialized or contains some status information.

3. A record with even coordinates is uninitialized.

Figure 5.4.2 shows a portion of the workplane for \( t=4 \). The important point is that before moving to any data record the head must cross a border record.

The simulating machine maintains a recursive partition of its workplane into frames, sub-frames, etc. that delimit areas not yet visited. Delimiting frames are constructed in time proportional to the perimeter rather than the area and this is the essential reason for the efficiency of the construction. Because the head movements of the simulated machine form a path, the head must cross the perimeter of a
frame before accessing any cell inside the frame.

An \( n \times n \) frame surrounds the workplane at level 0. In general, a level \( i \) frame has a perimeter of \( 4 \times 2^{i-1} \) border records and four sub-frames at level \( i+1 \). A level \( i \) frame is marked by border records containing the level number \( i \) in one field. A border record could be common to two adjacent frames; hence the need for two fields. If two frames are adjacent vertically (respectively horizontally) the records forming the common border are type "vertical" (respectively "horizontal"). Moreover, some fields maybe marked as "corner" to indicate the limits of a frame.

The second head performs two functions during the simulation:
(1) record workplace contents for re-organization described above.
   This could be a sequence over the alphabet
   \( \{ \text{shift}_\delta \mid \delta \in \{N,S,E,W\} \} \cup \{ \text{print}_\sigma \mid \sigma \in \text{workspace alphabet} \} \)

(2) arithmetic on integers in the range 1...n or 1...t.

The arithmetic is used to sub-divide frames and to move the first head
in increments of \( \sqrt{2} \log t \) so that it respects record boundaries. The
arithmetic is carried out in some fixed number of registers of length t.
These registers can be safely stored and manipulated in uninitialized
storage by writing them inside of a box of known shape and size.

---

```
construct n x n level 0 frame; enter-frame( 1 . 0 , "" );
do simulated machine not in final state +
   determine next move from transition table;
   if state change += something O(1)
     □ printing move + print symbol in data record
     □ shift\delta + enter-frame( i . j , \delta ) where
       (i,j) is the border record crossed
       by the shift
   fi;
endo
```

```
enter-frame( i . j , \delta ):
   let k be the level of the frame entered by crossing record (i,j);
   do k < t + sub-divide as in figure 5.4.4 od
```

Figure 5.4.3 Simulation algorithm
When the head crosses border record \((i,j)\), the machine sub-divides the frame into four sub-frames as shown, then returns to record \((i,j)\). The level numbers of the sub-frames are one greater than the level number of the frame before the sub-division.

**Figure 5.4.4 Sub-division upon frame entry**

---

**Section 1.4.1 Running time analysis**

The key to the running time analysis is the following observation about paths in the plane.

**Lemma**

Let the plane be marked off in a grid of frames of side \(c\). A path of length \(n\) can visit at most \(\frac{2n}{c}\) different frames.

**Proof** Let \(F\) be the set of frames visited by the path. Construct \(G \subseteq F\) by the following "thinning" procedure:

To start, set \(G = F\). As long as \(G\) contains an element \(g\) such that
one of the eight immediate neighbors of $g$ is also in $G$, delete from $G$ all immediate neighbors of $g$.

Upon termination $\#G \geq \#F/9$. Let $g_1, g_2, \ldots, g_p$ be the sequence of members of $G$ in the order in which they are visited by the path.

$p \geq \#G$ since every member of $G$ is visited at least once

$p \leq n/c$ since $c$ steps of the path are needed to get from any $g_k$ to $g_{k+1}$

Collecting:

\[ n/c \geq p \geq \#G \geq \#F/9 \Rightarrow \#F \leq \frac{9n}{c} \]

Q.E.D.

The cost to enter a level $i$ frame is the cost to trace the perimeters of frames at level $i$, level $i+1$, ..., level $t$. This is:

\[
\sum_{k=i}^{t} Q(2^{t-k}) = Q(2^{t-i})
\]

We apply the lemma to each of the $t$ levels. The cost associated with level $i$ is the number of level $i$ frames entered times the cost to enter a level $i$ frame. This is:

\[
\frac{3n}{2^{t-i}} \times Q(2^{t-i}) = Q(n)
\]

Since there are $Q(1)$ levels, the entire cost is $Q(n)$. 

Chapter 6

Closing Remarks

Section 6.1 Comparison of the paradigms

In this section we show some relationships between the paradigms presented in chapters 2, 3, and 4. We would like to make the point that there is seldom one "best" way to understand any algorithm, and that considering an algorithm, or its design, from several points of view leads to deeper understanding.

Deferred Updating and Predicate Weakening are closely related. The table management algorithm of figure 4.2.1 defers the expensive operation "re-establish (4.2.2)" for some number of insertions and, for this reason, runs more efficiently. Deferred updating is also necessary in the Turing Machine simulation of section 4.3 because the length of the simulated tape may be increasing. This updating is explained in detail in section 5.1.3.

On the other hand, the programs we describe in chapter 3 can also be understood as examples of Predicate Weakening. Consider the queue programming problem of section 3.1. Like invariant (4.2.1), the representation of figure 3.1.1 is "strict". There is only one way to arrange the queue values into a linear list Q. Suppose we split the list Q in half -- that is, into two pieces H and T, where the lengths of H and T differ by at most one. The data structure must satisfy invariant (6.1.1) below, which is analogous to (4.2.2).
Q = APPEND(R, REVERSE(T)) and
ABS(LENGTH(H) - LENGTH(T)) = 1 and
LENGTH(H) = 1 \Rightarrow T = NIL  \quad (6.1.1)

Maintaining predicate (6.1.1) still requires quadratic time. However we can weaken (6.1.1) by dropping the requirement that H and T be nearly the same length

Q = APPEND(R, REVERSE(T)) and
NULL(H) \Rightarrow NULL(T).  \quad (6.1.2)

Now several combinations of H and T can satisfy (6.1.2). This weakened invariant is analogous to 4.2.3.

Section 6.2 Related work

There are not many detailed expositions of paradigms in the programming literature. Major texts, such as Aho, Hopcroft and Ullman[1] and Dijkstra, Dahl and Hoare[8] name some design techniques and illustrate them by example, but there is no attempt to formalize the paradigms. Two important exceptions are the SETL project (see chapter 2) and Dynamic Programming. Dynamic Programming[2] has been the subject of several formalizations by Operations Researchers (see [6] for a nice survey of Dynamic Programming algorithms).

In the complexity literature, some authors have demonstrated patterns of reasoning that lead to classes of running time bounds[4]. Bentley has given several examples and proposes a kind of algebraic formalization[3]. He examines several data structures that manipulate different data structures and have identical, efficient running times. He then abstracts this manipulation as a collection of operations on a general, unspecified data structure and states criteria the operations
must satisfy to achieve the fast running time. Now any data structure with operations satisfying these criteria constitutes an instance of the paradigm and has the desired fast running time.

A similar treatment, particularly in the context of fast data structures for geometry problems, is given in [30].

Section 6.1 Suggestions for further work

Finally, we propose some ideas for further research:

(1) Give more examples of Predicate Weakening not in the context of data structure design (i.e. like the permutation algorithm).

(2) How does our algebraic formalization of the paradigms compare with the algebraic model of abstract data types?

(3) To what extent can these paradigms be automated? Could a compiler introduce predicate weakening or deferred updating into a high-level data structure? Something like this has already been done with the SETL project.

(4) Improve the ink-bounded simulation to \(O(n (\log n)^k)\) for fixed \(k\) or consider non-deterministic ink complexity.

(5) Is the time/space tradeoff for the permutation algorithm of section 5.3 optimal under some reasonable model?

(6) Extend our result on initialization-independent Turing Machines to dimension higher than two.
Bibliography


[26] Schnorr P.C. Satisfiability is Quasilinear Complete in NQL. JACM Volume 25 pp 136-145

[27] Schwartz J. Private communication.


