SHELLSORT AND SORTING NETWORKS

BY

VAUGHAN R. PRATT

STAN-CS-72-260
FEBRUARY 1972

COMPUTER SCIENCE DEPARTMENT

School of Humanities and Sciences

STANFORD UNIVERSITY

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Stanford University	Unclassified
	2b. GROUP

3. REPORT TITLE

SHELLSORT AND SORTING NETWORKS

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Technical Report - February 1972

5. AUTHOR(S) (First name, middle initial, last name)

Vaughan R. Pratt

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
February 1972	63	10

8a. CONTRACT OR GRANT NO	9a. ORIGINATOR'S REPORT NUMBER(S)
ONR N-00014-67-A-0112-0057	STAN-CS-72-260
b. PROJECT NO NSF GJ 992	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	None

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Office of Naval Research, Pasadena Branch

13. ABSTRACT

Shellsort is a particular method of sorting data on digital computers. Associated with each variant of Shellsort is a sequence of integers that characterizes that variant. In this paper we answer some open questions about the speed of Shellsort with certain characteristic sequences, and suggest a novel application of Shellsort, namely to sorting networks.

Shellsort with any characteristic sequence that approximates a geometric progression and that has short coprime subsequences through takes $O(n^{3/2})$ units of time. For any sequence that approximates a geometric progression with an integer common ratio, this bound is the best possible. (The notion of "sorting template" us used to prove this.) However, if the sequence consists of the descending sequence of positive integers less than n and having only 2 and 3 as prime factors, then Shellsort takes only $O(n \log^2 n)$ units of time. Sorting networks based on Shellsort with this sequence operate approximately 1.5 times as fast as with previous methods.

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
none						

SHELLSORT AND SORTING NETWORKS

by Vaughan R. Pratt

Abstract

Shellsort is a particular method of sorting data on digital computers. Associated with each variant of Shellsort is a sequence of integers that characterizes that variant. In this paper we answer some open questions about the speed of Shellsort with certain characteristic sequences, and suggest a novel application of Shellsort, namely to sorting networks.

Shellsort with any characteristic sequence that approximates a geometric progression and that has short coprime subsequences throughout takes $O(n^{3/2})$ units of time. For any sequence that approximates a geometric progression with an integer common ratio, this bound is the best possible. (The notion of "sorting template" is used to prove this.) However, if the sequence consists of the descending sequence of positive integers less than n and having only 2 and 3 as prime factors, then Shellsort takes only $O(n \log^2 n)$ units of time. Sorting networks based on Shellsort with this sequence operate approximately 1.5 times as fast as with previous methods.

ACKNOWLEDGMENTS

Table of Contents

List of Illustrations

Chapter 1

Introduction to Shellsort

The problem is to sort the elements of the array

A $A[1], A[2], \ldots, A[n]$ into ascending order, given some total ordering

on the possible values of the elements of A . The high cost of

random-access memory together with the speed of in-core sorting

motivates the consideration of algorithms that sort arrays "in their

own length", with little or no auxiliary storage requirements beyond

what is needed to hold the array. A number of such algorithms are

known, and all but Shellsort [Shell, 1959] have proved more or less

amenable to an analysis of bounds on their running time, as a function

of n . Chapter 2 shows that $O(n^{3/2})$ units of time is the best

possible upper bound on the more conventional variations of Shellsort.

To discuss Shellsort requires some terminology. A p-chain of A

is a sequence of elements of A occurring at intervals of p . For

instance, if $n = 8$, then A has three 3-chains, $\{A[1], A[4], A[7]\}$,

then $\{A[2], A[5], A[8]\}$, and then $\{A[3], A[6]\}$. In general, A has

$\min(n,p)$ p-chains, each of length $\lceil \frac{n}{p} \rceil$ or $\lfloor \frac{n}{p} \rfloor$.

When A's p-chains are in ascending order, A is defined to be

p-ordered. To p-sort A is to sort A's p-chains.

Shellsort works by repeatedly p-sorting A for a characteristic

sequence (abbreviated to "sequence" hereafter) of p's , with the last p

being 1 . This last value ensures that A is sorted by this process, since

a 1-ordered array must be ordered. Furthermore, Shellsort prescribes a

particular technique for sorting each p-chain, namely insertion sorting.

Insertion sorting is a technique whereby one starts with an array

of no elements, and some source of n entries, and progressively builds

1

up a sorted array starting with A[1],A[2],... by (i) determining for each
entry where in the array so far constructed it should go in order to
keep the array sorted, (ii) moving the appropriate array elements up one
place to make room for it, and (iii) inserting it. Since the space consumed
by the partially constructed array and that consumed by the remaining
uninserted entries is just n items, this method can be used to sort
in place, requiring almost no auxiliary storage, by combining all the
operations for each entry into the one loop, as follows:

procedure insertionsort(A);

    for i := 2 until length(A) do

        for j := i step -1 until 2 while A[j-1] > A[j] do

            swap (A[j-1],A[j]);

The while clause signifies that the iteration is to be terminated if
the expression following the "while" becomes false. The procedure "swap"
exchanges the contents of the locations named by its arguments. The
expression " length(A) " is supposed to be what it says. The variables
i and j are assumed to be declared implicitly, as in ALGOL W, by being
named as the controlled variable of a for loop.

    The outer loop of the procedure cycles through the source of entries.
A[1] is not processed since the destination of its contents must be A[1] .
The inner loop takes each entry and shuffles it backwards through the
array to its proper place. After each execution of the body of the
outer loop, but before i is incremented, the array A[1:i] is ordered.

    Let us define an inversion in an array A to be a pair of elements,
A[i] and A[j] , such that i < j but A[i] > A[j] . Thus A is ordered

2

if and only if there are no inversions in A . Define an adjacent
inversion to be one whose elements are adjacent.   Then the insertion
sort above can be seen to eliminate adjacent inversions. No other
inversions appear or disappear because every other pair of elements
maintain their relative positions after the exchange. Thus each exchange
reduces the number of inversions in A by one. Since A can have up
to $\binom{n}{2}$ inversions (when A is in descending order, i.e., $A[i-1] > A[i]$
everywhere in A), this technique may take up to $\binom{n}{2}$ exchanges to sort A ,
or $O(n^2)$ exchanges.

The idea underlying Shellsort is that moving elements of A long
distances at each swap in the early stages, then shorter distances later,
may reduce this $O(n^2)$ bound.

An algorithm for Shellsort using the procedure "insertionsort" is
not easy in ALGOL. We might write, in near-ALGOL:

```
procedure Shellsort (A,P,m)
    for i := 1 until m do
        for j := 1 until P[i] do
            insertionsort (A[*xP[i]+j]);
```

The expression $A[*xp+j]$ denotes simply the j-th p-chain of A .
The more usual way to write Shellsort carries out the insertion
sort on a time-shared basis, thus:

```
procedure Shellsort (A,P,m);
    for i := 1 until m do
        for j := P[i]+1 until length(A) do
            for k := j step -P[i] until P[i]+1 while A[k-P[i]] > A[k] do
                swap (A[k-P[i]],A[k]);
```

Because Shellsort works by correcting inversions within p-chains, it is convenient to call such inversions p-inversions.

The time spent by Shellsort is made up of what it would do with an ordered array, plus an amount of time at most proportional to the number of exchanges it must do to sort the array. Since the former time is n times the number of passes, and since the number of passes considered in the next chapter is always $O(\log n)$, we shall measure the time required by Shellsort in **units of the number of exchanges performed**. To convert this figure to seconds, multiply it by the number of seconds required for an exchange, a decrement of k, a test for $k \geq P[i]+1$ and a subsequent comparison, and add the time required to Shellsort an ordered array of the same size. Since the dominant term in the expressions derived in Chapter 2 is $O(n^{3/2})$, the time for exchanges asymptotically dominates the $O(n \log n)$ time for Shellsorting an ordered array, which is why the number of exchanges is an adequate measure in that chapter.

Let us now summarize the remainder of the thesis. In Chapter 2, we show that Shellsort takes time proportional to $n^{3/2}$ in the worst case. Prior to this, only Papernov and Stasevich's [1965] upper bound of $O(n^{3/2})$ for Shellsort with Hibbard's sequence was known. In Chapter 3, we describe a considerably faster Shellsort that operates with only $O(n \log^2 n)$ units of time, and in Chapter 4 we show that under quite reasonable conditions this version of Shellsort can be used to build a sorting network that requires $0.3 \log^2 n$ units of delay, about 1.5 times as fast as was previously possible. [Batcher, 1968].

Further prologue relevant to Chapter 2 may be found in section 2.1 of that chapter. Chapter 5 presents a more detailed summary and unification of the results of Chapters 2 to 4, and also suggests problems for further research.

Chapter 2

Least Upper Bounds for Most Shellsorts

2.1 Discussion

A natural characteristic sequence to follow when Shellsorting is a geometric progression. If one thinks of Shellsorting as progressively bringing each element closer to its final position, in jumps of decreasing size, it is "natural" to arrange that these jumps decrease geometrically; this is what happens in a binary search, for example. Possibly some such consideration has motivated the choice of a (usually slightly perturbed) geometric progression for almost all Shellsorts.

If a sequence of even numbers, followed by 1, is used, Shellsort may take up to $n(n-2)/8$ exchanges when 1-sorting. This would happen if one 2-chain were $1, 2, \ldots, n/2$ and the other were $\frac{n}{2} + 1, \frac{n}{2} + 2, \ldots, n$. Since this array is 2-sorted, it is 2k-sorted for all $k > 0$. Thus at the last pass, the original array is being 1-sorted, that is, it is simply being insertion-sorted, which is readily seen to take $1 + 2 + 3 + \ldots + (\frac{n}{2} - 1) = n(n-2)/8$ exchanges, for even n, an $O(n^2)$ figure.

Shell [1959] originally suggested the sequence $\lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{4} \rfloor, \ldots, \lfloor \frac{n}{2^k} \rfloor \ldots, 1$. If n is a power of 2, this is readily seen to be a sequence dealt with in the previous paragraph. This problem was recognized by Lazarus and Frank [1960], who proposed that the even elements in Shell's sequence be incremented by one. Thus every element can be expressed as $2k+1$, and its successor must be either k or $k+1$, depending on whether k is odd or even respectively. Now $k | 2k$

and $k+1 | 2k+2$, so $(2k+1,k) = (2k+1,k+1) = 1$; that is, every consecutive pair of elements in the sequence is coprime. We shall see shortly that $O(n^{3/2})$, not $O(n^2)$, is the (least) upper bound for Lazarus and Frank's sequence, mainly because of this coprimeness property.

Hibbard [1963] suggested the descending sequence of all numbers of the form $2^k-1 < n$, integer $k \geq 1$. When n is one less than a power of 2 , this sequence coincides with both Shell's sequence and Lazarus and Frank's sequence. Many other sequences have been suggested [cf. Knuth 1972], almost all of them having in common that they form "fuzzy" geometric progressions, with every element relatively prime to at least one of its nearby predecessors. (It is interesting to note that both of these guidelines are ignored in the sequence of Chapter 3 for the $O(n \log^2 n)$ Shellsort.)

The next part of this chapter will prove a theorem enabling us to show that the above Shellsorts take at most $O(n^{3/2})$ units of time, provided their sequences have the coprimeness property. The last part will prove a theorem applicable to Shellsorts whose sequences are fuzzy geometric progressions with _integer_ common ratios, enabling us to prove that the $O(n^{3/2})$ figure cannot be improved in such a case.

2.2 An Upper Bound For Most Shellsorts

The first result is essentially a generalization of Papernov and Stasevich's theorem [1965] that Shellsort with Hibbard's sequence takes at most $O(n^{3/2})$ units of time. The basic properties we shall impose on the class of sequences covered by the result are that they approximate geometric sequences and that every d consecutive elements in the latter part of the sequence form a coprime set of integers, for some d .

We shall need in advance some auxiliary lemmas. The first is the "non-messing-up" theorem for p-sorting and q-sorting.

Lemma 2.1. Given positive integers p and q , and a p-ordered array A with n elements, q-sorting A leaves it p-ordered.

Proof. (This is a slight modification of a proof in [Boerner, 1955, p137].)

Let j be such that $A[j-p] > A[j]$ after q-sorting. We shall give one method of q-sorting which contradicts this, whence it follows that all methods contradict this, since the outcome of q-sorting is unique.

Let $A[j-p]$, $A[j]$ belong to q-chains B and C respectively. Now B and C must be distinct, otherwise $A[j-p] \leq A[j]$ because each q-chain is ordered.

Call the least element of A , $-\infty$, and the greatest, ∞ .

Sort all the q-chains except B and C . Now put B and C into correspondence, with $A[j-p]$ corresponding to $A[j]$, $A[j-q-p]$ to $A[j-q]$, $A[j+q-p]$ to $A[j+q]$, etc. If necessary, extend B and C to

ensure that every element has a mate, using $-\infty$ for B and ∞ for C . Call the extended q-chains B' and C' . We now have the situation of Figure 2.1, as the reader may check. (Here (a,c) and (b,d) are two instances of corresponding elements. Lower valued subscripts of A correspond to elements closer to the top of the figure.)

$$
B \left\{ \begin{array}{ccc}
-\infty & \leq & \\
\vdots & & \\
-\infty & \leq & \\
a & \leq & c \\
b & \leq & d \\
& \leq & \infty \\
& \leq & \infty
\end{array} \right\} C
$$

B'   C'
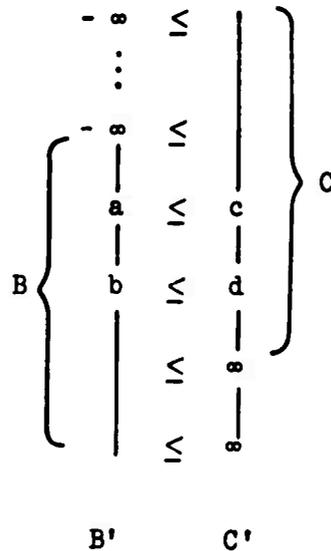
Figure 2.1

Corresponding q-chains B' and C' .

Now sort B' and C' thus:

1. Use a sorting algorithm which sorts every array of a given size n by using a fixed sequence of pairs (i,j) depending only on n and drawn from [1,n] x [1,n] . For each such pair, it puts A[i] and A[j] in order. The insertion sort of Chapter 1, with the <u>while</u> condition deleted, is such an algorithm.

2. For each pair (A[i],A[j]) in B' ordered by this algorithm, simultaneously order the corresponding elements (A[i+p],A[j+p]) in C' . Thus B' and C' are sorted in parallel.

8

Let a,b in B' and c,d in C' be four elements participating in one step of this algorithm, with a,b,c,d in the order shown in Figure 2.1. Suppose before the step, we had $a \leq c$ and $b \leq d$. We claim that after the step, the two resulting corresponding pairs will still be ordered. This is trivially true if neither or both of (a,b) and (c,d) are swapped. If only (a,b) is swapped, we must have had $b < a \leq c \leq d$ before, and if only (c,d) is swapped, then we had $a \leq b \leq d < c$; in both cases, both elements of B are less than or equal to both elements of C , proving the claim.

Since corresponding pairs are ordered at the start, they must therefore be ordered at the end, by induction on the steps of the algorithm. Now the extensions clearly cannot have moved, so they may be removed. The result is just as if we had q-sorted B and C . But now $A[j-p] \leq A[j]$. This contradiction completes the proof.

An immediate corollary is that if an array is p_1-sorted, then p_2-sorted, and so on up to p_k , it is then p_i-ordered for $i = 1,2,\ldots,k$.

If the diophantine equation

$$p_1 x_1 + p_2 x_2 + \cdots + p_k x_k = q , \quad \text{all } p_i > 0 , \tag{1}$$

has non-negative solutions in the x_i , then an array p_i-ordered for these p_i's is also q-ordered, by the transitivity of the ordering relation, since the solution indicates the existence of a sequence

$$A[j] \leq A[j+p_1] \leq A[j+2p_1] \leq \cdots \leq A[j+x_1 p_1] \leq A[j+x_1 p_1 + p_2] \leq \cdots$$
$$\leq A[j+p_1 x_1 + \cdots + p_k x_k] = A[j+q] , \quad \text{for all } j .$$

Lemma 2.2. When $\gcd(p_1, p_2, \ldots, p_k) = 1$, and $q \geq p_m(p_1 + p_2 + \ldots + p_k - p_m)$ for some p_m, equation (1) always has non-negative integer solutions in the x_i.

Proof. It is well known that when $\gcd(p_1, p_2, \ldots, p_k) = 1$, the diophantine equation (1) always has a solution in $x_1, \ldots, x_k$. The set of possible solutions must be closed under the operation of simultaneously adding p_j to x_i and subtracting p_i from x_j, since this adds $(p_i p_j - p_j p_i) = 0$ to the left-hand side of the equation. Thus there must be a solution in which for all $i \neq m$, $0 \leq x_i < p_m$, since each x_i other than x_m can be adjusted by increments of p_m, at a cost to x_m. But now we have

$$p_m((p_1 + p_2 + \ldots + p_k) - p_m) > (p_1 x_1 + p_2 x_2 + \ldots + p_k x_k) - p_m x_m, \quad (i \neq m \supset x_i < p_m)$$

$$= q - p_m x_m, \qquad \text{(equation (1))}$$

$$\geq p_m(p_1 + p_2 + \ldots + p_k - p_m) - p_m x_m, \quad \text{(Hypothesis)}$$

from which it follows that $x_m > 0$ in this particular solution.

Q.E.D.

We may infer from Lemma 2.2 that if an array A is p_i-ordered for $p_1, \ldots, p_k$, and $\gcd(p_1, p_2, \ldots, p_k) = 1$, A is p-ordered for all $p \geq p_m(p_1 + p_2 + \ldots + p_k - p_m)$, where p_m is any one of the p_i's. Thus an upper bound on the number of elements of a p-chain B which may precede and be greater than a given element of B is $p_m(p_1 + p_2 + \ldots + p_k - p_m)/p$ since the elements of a p-chain are spaced p apart.

Hence to p-sort A requires at most $np_m(p_1+p_2+\ldots+p_k-p_m)/p$ exchanges.
Call this the <u>first upper</u> bound.

Within a p-chain (of at most $\lceil \frac{n}{p} \rceil$ elements), the average element
can participate in at most $\frac{1}{2}(\lceil \frac{n}{p} \rceil - 1)$ exchanges during p-sorting.
So the total number of exchanges required is at most $\frac{1}{2} n(\lceil \frac{n}{p} \rceil - 1)$.
Hence $n^2/2p$ is also a (larger) upper bound, the <u>second upper bound</u>.

Before proceeding with the formalism of the main result, let us
provide some insight into what is going on. The two upper bounds we
have just derived are about to be used to bound the time required by
Shellsort using a characteristic sequence having properties shared by
most of the suggested sequences, excluding those for which Shellsort is
clearly an $O(n^2)$ algorithm. The properties we require of a sequence S
are (for the moment): that there is a sequence S' such that to each
element p of S there corresponds an element p' of S' , with fixed
bounds on $p-p'$ ("additive fuzziness", namely $-a \le p-p' \le b$, for fixed
a, b); and that each element of S' is between r and s times its
successor, for fixed $r, s > 1$. That is, S' is a sloppy decreasing
geometric progression ("multiplicative fuzziness").

These conditions are general enough to cover most sequences that
could be called "fuzzy" geometric progressions.

We also impose a coprimeness condition on neighboring elements of
the sequence, to satisfy the conditions required for the first upper bounds.
For some integer d independent of n, every d consecutive elements must be
relatively prime.

We shall use the first upper bound to bound the time spent by
Shellsort when p-sorting for small p . The second upper bound is for
large p . While the latter remark makes sense ($n^2/2p$ is small for
large p), the former may seem not to, at first, since p appears in

11

the denominator of the first upper bound also. However, the numerator is an $O(p^2)$ expression, and the p_i's will be just those elements in the sequence that immediately precede p . Our conditions ensure that these decrease in approximate proportion to p , so the first upper bound is really an $O(p)$ expression rather than an $O(1/p)$ one.

Because we do not need the first upper bound for large elements of S , we shall actually restrict the condition that S be a fuzzy coprime geometric progression to the small elements of S . We impose a much weaker condition on the large elements of S , that the sum of their reciprocals be an $O(1/\sqrt{n})$ quantity. (This condition is readily seen to hold for the first half of a fuzzy geometric progression with all elements less than n , since the smallest element in that half is itself an $O(\sqrt{n})$ quantity.)

We now proceed with the formalism.

Theorem 2.4. Let r,s,t,u,v be reals, with $r,s > 1$ and $t,u,v > 0$.

Let a,b,d be integers, with $a,b \geq 0$, $d \geq 2$.

To each array size n , suppose there corresponds a sequence $p_1, p_2, \ldots, p_m$ and an index c (denoting the cut-off point p_c) such that

(i) $p_m \quad 1$ (to ensure that Shellsort really sorts)

(ii) $\displaystyle\sum_{1 \leq j < c} \frac{1}{p_j} \leq \frac{u}{\sqrt{n}}$ (the large p_j for the second upper bound)

(iii) $c > d$ (so that the first upper bound is usable for

               elements $p_c, p_{c+1}, \ldots, p_{c+d-1}$)

(iv) $p_c < t/n$ (to keep small those elements covered by

the first upper bound, in conjunction with

condition (vii))

(v) $j \geq c$ implies $\gcd(p_{j-1}, \ldots, p_{j-d}) = 1$ (for the first

upper bound)

(vi)   There is a sequence $S' = p'_c, p'_{c+1}, \ldots, p'_m$ in which

$-a \leq p_i - p'_i \leq b$, for $i = c, \ldots, m-1$ and $p'_m \geq v$.

(vii)   In S' , $p'_i \geq rp'_{i+1}$, for $i = c, \ldots, m-1$.

(viii)   In S' , $p'_i \leq sp'_{i+1}$, for $i = c, \ldots, m-1$.

Then with these conditions, Shellsort takes $O(n^{3/2})$ units of time.

•

Proof. Applying the second upper bound is easy. The total time
required for p_j-sorting, for $j < c$, is at most

$$\sum_{1 \leq j < c} \frac{n^2}{2p_j}$$ (using the second upper bound)

$$\leq \frac{1}{2} u n^{3/2}$$ (by condition (ii))

The remainder of the sequence requires a little more work. However,
the underlying idea remains simple, that the first upper bound decreases
approximately geometrically as Shellsort progresses through the sequence,
and hence the total time required is proportional to that required for
p_c-sorting alone.

First note that

$$P_i \leq p_i' + b \qquad \text{(condition (vi))}$$

$$\leq s^k p_{i+k}' + b \quad , \quad k \geq 0 \qquad \text{(condition (viii))}$$

$$\therefore \quad P_i \leq s^k p_{i+k} + s^k a + b \qquad \text{(condition (vi))} \qquad (2)$$

and

$$P_i \geq p_i' - a \qquad \text{(condition (vi))}$$

$$\geq r^k p_{i+k}' - a \quad , \quad k \geq 0 \qquad \text{(condition (vii))}$$

$$\therefore \quad P_{m-k} \geq r^k v - a \qquad \text{(condition (vi))} \qquad (3)$$

Also

$$P_{i+k} \leq (p_i + a)/r^k + b \qquad \text{(condition (vi))} \qquad (4)$$

Then the total time required for p_j-sorting, for $j \geq c$, is at most

$$\sum_{c \leq j \leq m} m p_{j-1}(p_{j-2} + \ldots + p_{j-d}) / p_j \qquad \text{(the first upper bound,}$$
$$\text{and conditions (iii), (v))}$$

$$\leq \sum_{c \leq j \leq m} n(s p_j + sa + b)[(s^2 p_j + s^2 a + b) + (s^3 p_j + s^3 a + b) + \ldots + (s^d p_j + s^d a + b)] / p_j$$

$$\text{(by inequality (2))}$$

$$\leq (s^2 + \ldots + s^d)n \sum_{c \leq j \leq m} \left(s + \frac{sa+b}{p_j} \right) (p_j + a + b) \qquad \text{(since } s \geq 1)$$

$$\leq (s^2 + \ldots + s^d)n \sum_{1 \leq p_j < t\sqrt{n}} (sp_j + sa + b + (2sa + b)(a + b)) \qquad \text{(since } p_j \geq 1 ;$$
$$\text{also note use of}$$
$$\text{condition (iv))}$$

$$\leq (s^2 + \ldots + s^d)n \sum_{1 \leq r^k < \frac{t\sqrt{n}+a}{v}} \left(\frac{s(t\sqrt{n}+a)}{r^k} + K \right)$$

$$\text{(K = } (s(2a+1)+b)(a+b)) \text{ ; } s \geq 1 ;$$
$$\text{and using inequalities (3), (4))}$$

14

$$= (s^2 + \ldots + s^d)n \sum_{\frac{a+1}{v} \le r^k < \frac{t\sqrt{n}+a}{v}} (sr^k v + K)$$

$$< (s^2 + \ldots + s^d)n \left(\frac{rs(t\sqrt{n}+a)}{r-1} + K \log_r \left\lceil \frac{t\sqrt{n}+a}{v} \right\rceil \right) \qquad \text{(summing the geometric progression)}$$

$$= (s^2 + \ldots + s^d) \left(\frac{r}{r-1} s(t\, n^{3/2} + an) + K \log_r \left\lceil \frac{t\sqrt{n}+a}{v} \right\rceil \right) n \qquad .$$

Hence the total time required for Shellsort is less than

$$\left(\frac{u}{2} + (s^2 + \ldots + s^d) \frac{rst}{r-1} \right) n^{3/2}$$

$$+ (s^2 + \ldots + s^d) \left(\frac{rsa}{r-1} + \frac{(s(2a+1)+b)(a+b)}{\log_2 r} \left(\log_2 \left(\frac{t\sqrt{n}+a}{v} \right) \right) \right) n \qquad .$$

This completes the proof of the $O(n^{3/2})$ upper bound result for Shellsort with this class of characteristics sequences.

The reader may readily calculate the values of r, s (both 2 in the sequences of Chapter 1), u (a function of t, clearly, as well as of r, s, a and b), a, b and v for various sequences, and may amuse himself determining the value of t that minimizes the bound in each case.

For the case of Hibbard's sequence, for example, where $p_i = 2^{\lfloor \log_2 n \rfloor - i + 1} - 1$, take $r = s = t = v = d = 2$, $u = a = 1$, $b = 0$, $c = \lfloor \log_2 n \rfloor - \frac{1}{2} \log_2 n$. Condition (i) is satisfied since Hibbard's sequence contains 1. Condition (ii) is satisfied since $p_c = 2\sqrt{n} - 1 \ge \sqrt{n}$ (for $n \ge 1$), and $p_i = 2p_{i+1} + 1$. Condition (iii) holds for $n \ge 4$. Condition (iv) holds since $p_c = 2\sqrt{n} - 1$ (see above). Condition (v) holds since $(p_i, p_{i+1}) = 1$ trivially. Conditions (vi) to

(viii) hold if p_i' is taken to be p_i+1. Thus our theorem is true for all $n \geq 32$. Making the substitutions, the dominant term of our upper bound is $32.5 \, n^{3/2}$.

2.3 Optimality of the $O(n^{3/2})$ Bound

In this section we shall construct arrays that take time proportional to $n^{3/2}$ to sort using Shellsort with sequences that approximate a geometric progression with integer common ratio. Most of the proposed sequences to date have this property.

The basic tool for the construction is a sorting template. (Visualize this as a strip of cardboard with some holes in a straight line; the elements of the template are the hole locations, numbered right to left.)

Definition 2.1. A sorting template is a set of natural numbers containing 0 and closed under addition.

Definition 2.2. The sorting template generated by a set is the least sorting template containing that set.

For example, the sorting template generated by $\{1\}$ is the set N of natural numbers, while that generated by $\{2,5\}$ is $N-\{1,3\}$.

Definition 2.3. An array element $A[i]$ is visible through a sorting template T at j when $j-i$ is in T.

(Visualize A as being written on a sheet of paper underneath T, with subscripts numbered from left to right. The zero hole of T is over $A[j]$.)

Definition 2.4.   An array A is _constructed_ _under_ _a_ _sorting_ _template_ T
_at_ _a_ _sequence_ q[1],...,q[m] thus:

l := 0;
    for i := 1 **until** m **do**
        for j := 1 **until** n **do**
            if A[j] is undefined **and** A[j] is visible through T at q[i]
                then **begin** l := l+1; A[j] := l **end**;

Note that each element of A is initially _undefined_, and _becomes_
defined by assigning l to it, after which it _is_ _defined_.

Intuitively, we put the template down with the zero hole of the
template on A[q[1]] ; then we move the template to A[q[2]] and so on.
At each position, we fill in all the visible but as yet undefined
elements of A , using ever-increasing numbers. Some of the language
we employ later assumes that this intuitive view has been grasped.

Lemma 2.5.   If p∈T , then an array A constructed under the sorting
template T is p-ordered. (Hence the name _sorting_ template.)

Proof.   Say A[j] becomes defined when T is at q . Then q-j ∈ T .
So q-(j-p) ∈ T also, since p∈T and T is closed under addition.
Thus A[j-p] must be assigned its value before A[j] , whence
A[j-p] < A[j] .

<div align="right">Q.E.D.</div>

Let us use the notation [a,b] to denote (the set of integers in)
the interval from a to b inclusive. By the _length_ of [a,b] we
shall mean b-a+1 . By A < B , for intervals A and B , we
shall mean every element of A is less than every element of B .

Let us now give an informal preview of the formalities to follow. Our goal is to be able to construct arrays that take time $n^{3/2}$ to Shellsort.

As the preceding section showed, p-sorting for p near the beginning and end of sequences takes only linear time; only near the middle can an additional factor of $\sqrt{n}$ creep in to spoil things. Thus if we are going to find arrays that take time $n^{3/2}$, we should arrange things so that Shellsort finds the going toughest halfway through the sequence. To do this, we shall construct arrays that look as if Shellsort is already halfway through sorting them, and yet that have many inversions. Thus when sorting these arrays, Shellsort will zip through the first half of the sequence finding nothing to do, and then suddenly hit a brick wall, and take time $n^{3/2}$ in a single p-sorting pass. We do not much care what happens for the rest of the sequence.

The preceding definitions and lemmas established the basic tools for the construction. The following lemmas establish some quantitat ⋅ results of use in the analysis of the actual construction, which is described in the first paragraph of the Proof of Theorem 2.11.

<u>Lemma 2.6.</u> The sorting template T generated by [a,b] is $\bigcup_{k \geq 0} [ka,kb]$.

<u>Proof.</u> The union certainly contains {0} and [a,b] . To see that the union is closed under addition, take m,n such that $k_1 a \leq m \leq k_1 b$ and $k_2 a \leq n \leq k_2 b$. Then $(k_1 + k_2)a \leq m+n \leq (k_1 + k_2)b$. Thus the union is a sorting template containing [a,b] .

To see that it is the least such sorting template, suppose m is the least integer which is in the above union, but is not in every other sorting template containing [a,b] . Then $(k+1)a \leq m \leq (k+1)b$ for some $k \geq 1$. But every number in this range is expressible as the sum of two numbers from [a,b] and [ka,kb] respectively. This contradicts the closure property of the template lacking m .

<u>Lemma 2.7.</u> If T is generated by [a,b] and a < b , then iϵT for all $i \geq a^2/(b-a)$.

<u>Proof.</u> The complement of T , $\bar{T}$, is the set [1,a-1] $\cup$ [b+1,2a-1] $\cup$ ... $\cup$ [kb+1,(k+1)a-1] $\cup$ ... , by Lemma 2.6. When kb+1 $\geq$ (k+1)a , these intervals vanish. This happens for $k \geq (a-1)/(b-a)$. Thus the largest possible element of $\bar{T}$ is $((a-1)/(b-a))a-1$, which is certainly less than $a^2/(b-a)$.

<u>Lemma 2.8.</u> If T is generated by [a,b] , a < b , then for any non-negative integer p < a there is an interval I of length p in $\bar{T}$ such that T has exactly $\lceil \frac{a-p}{b-a} \rceil$ intervals of the form [ka,kb] , $k \geq 0$, which are less than I .

Proof. Choose I of length p and lying in $[(\lceil\frac{a-p}{b-a}\rceil-1)b+1, \lceil\frac{a-p}{b-a}\rceil a-1]$.

This latter interval lies in $\overline{T}$ since it is of the form $[kb+1,(k+1)a-1]$ (see proof of Lemma 2.7), and is of length $(\lceil\frac{a-p}{b-a}\rceil(a-b)+b-1)$, which is certainly not less than p. Now take the $\lceil\frac{a-p}{b-a}\rceil$ intervals of T to be $[0,0]$, $[a,b]$, $[2a,2b]$, $\ldots$, $[(\lceil\frac{a-p}{b-a}\rceil-1)a,(\lceil\frac{a-p}{b-a}\rceil-1)b]$, all of which are clearly less than I.

Lemma 2.9. With T,I as in Lemma 2.8, the number of elements of T which are less than any element of I are at least $\frac{1}{2}(a-p)(\frac{a-p}{b-a}-1)$.

Proof. We shall sum just the complete intervals $[ka,kb]$ of T for k to $\lceil\frac{a-p}{b-a}\rceil$.

$$\sum_{0 \leq k < \lceil\frac{a-p}{b-a}\rceil} k(b-a) = \frac{1}{2}(b-a)\lceil\frac{a-p}{b-a}\rceil\lceil\frac{a-p}{b-a}-1\rceil$$

$$\geq \frac{1}{2}(a-p)(\frac{a-p}{b-a}-1) .$$

Lemma 2.10. Let $c\epsilon T$ and let A be constructed under T at $c,2c,3c,\ldots,mc$, for some m. Then if some $A[j]$ is visible through T at ic, it is visible through T at jc for all $j \geq i$. That is, once visible, always visible. Conversely, every invisible element must be undefined.

<u>Proof</u>. If $A[j]$ is visible through T at ic , the $ic-j \in T$.
But $c \subset T$, whence $jc-j \in T$, by closure of T under addition. Thus
$A[j]$ is visible through T at jc.

We are now ready for the main theorem.

<u>Theorem 2.11</u>. Let r,s be reals greater than 1 . Let a,b be
non-negative integers. Then there exist non-negative reals
$t,u,c,$ with $rt < u$, such that

> <u>if</u> for every array size n Shellsort uses a sequence
> $S_n = p_1,\ldots,p_m$ with the properties that
> (i) there is some element p in S_n such that $t/n \le p \le u/n$
> and for all p_j preceding p , there is an integer m
> for which $m(p-a) \le p_j \le m(p+b)$; and
> (ii) the successor of p in S_n, q , satisfies
> $r(q-b)-a \le p \le s(q+a)+b$ and also $q < p-a$;
> <u>then</u> Shellsort takes at lease $cn^{3/2}$ units of time on some arrays
> of length n .

<u>Proof</u>. Construct A under the sorting template T generated by the
integers in $[p-a,p+b]$ at the sequence $p,2p,3p,\ldots,gp,$ where g is
large enough to ensure that A is completely defined. By Lemma 2.7,
every element of A is visible through T beyond $n+(p-a)^2/(a+b)$,
whence it suffices to take $g=\lceil n+(p-a)^2/(a+b)\rceil$.

We now show that A takes $cn^{3/2}$ units to Shellsort.

22

For the p_k in S_n up to and including p , the conditions of the theorem ensure that p_k is in T . Thus after p_k-sorting, for all p_k preceding q , A is left unchanged (Lemma 2.5). So the number of q-inversions in the original A gives a lower bound on how fast A will be sorted. We shall give a lower bound on the number of these inversions.

Initially T is at p in the construction, whence the first interval of $\bar{T}$ "covers" some of the first p elements of A . As T advances by p units each time, exactly one new interval of $\bar{T}$ (which of course must move as T does) appears, to cover elements of A . Eventually the I of Lemma 2.8 (of length q in this case) appears. There are $\lfloor \frac{n}{p} \rfloor$ disjoint contiguous sequences of elements of A of length p such that as T progresses, I will partly cover each such sequence in turn. Hence there must be at least $\lfloor \frac{n}{p} \rfloor$ positions of T during this construction for which q elements of A are covered by I .

During this process there is an f such that when T is at fp , interval I covers some contiguous set V of elements of A within $A[1]$ to $A[p]$. By Lemma 2.10, the elements of V must be undefined at this time. By the construction, the set D of those elements of A visible through T at fp and that lie to the right of V must be or become defined at this time. Hence the value of each element d in D must be less than that of each element of V (since the latter becomes defined later) and in particular less than one that is in d's q-chain, since every q-chain must have a representative in any interval of length q . Thus there are at least $|D|$ inversions within the q-chains of A .

From this time on, as T is advanced, new elements of A become covered by I , resulting in $|D|$ more q-inversions by the same argument. Eventually T will be at some point beyond n , and the number of inversions contributed in this way for T at each subsequent position will start to

decrease. After $\lfloor \frac{n}{p} \rfloor$ such advances of T, I "falls off" the right hand end of A.

We might start to count the number of q-inversions in A by multiplying $|D|$ by $\lfloor \frac{n}{p} \rfloor$ since the q-inversions are all distinct (because T moves further than q places each time). However, this would then include those q-inversions whose right-hand members lie outside A. So we need an upper bound on the number of those q-inversions, which we shall then subtract from $|D| \cdot \lfloor \frac{n}{p} \rfloor$ to give a lower bound on the number of q-inversions in A.

At the first position of T at which we start to lose q-inversions to this effect, we lose q-inversions corresponding to just that element in the first interval of T, namely the zero element. At the next position we lose at most those q-inversions corresponding to the zero, and the interval $[p-a,p+b]$, a total of $1+(b+a+1)$. At the k-th position, we lose a number of q-inversions bounded by

$$\sum_{0 \leq j < k} (j(b+a)+1) \; .$$

This process stops as soon as the interval I of $\bar{T}$ leaves the array. Hence k may be bounded by the result of Lemma 2.8. So the subtraction term is at most

$$\sum_{1 \le k < \left\lceil \frac{(p-a)-q}{(p+b)-(p-a)} \right\rceil} \quad \sum_{0 \le j < k} (j(b+a)+1)$$

$$= \sum_{1 \le k < \left\lceil \frac{p-a-q}{a+b} \right\rceil} (\tfrac{1}{2}(a+b)k^{\underline{2}} + k + 1) \quad , \quad \text{where} \quad k^{\underline{i}} = (k(k-1)\ldots(k-i+1)$$

$$= \tfrac{1}{6}(a+b)\left[k^{\underline{3}} + 3k^{\underline{2}} + 6k \right]_1^{\left\lceil \frac{p-a-q}{a+b} \right\rceil} \quad , \quad \text{since} \quad \sum_{a \le k < b} k^{\underline{j}} \quad \text{is} \quad \tfrac{1}{j}[k^{\underline{j+1}}]_a^b$$

$$< \tfrac{1}{6}(a+b)\left[k^3 + 5k \right]_0^{\frac{p+b-q}{a+b}}$$

$$= \tfrac{1}{6}(a+b)\left((\tfrac{p+b-q}{a+b})^3 + \tfrac{5(p+b-q)}{a+b} \right) \quad .$$

Now $|D|$ is given by Lemma 2.9, namely $\tfrac{1}{2}(p-a-q)(\tfrac{p-a-q}{a+b} - 1)$. Hence the number of inversions in q-chains is at least

$$\tfrac{1}{2}(\tfrac{n}{p} - 1)(p-q-a)(\tfrac{p-q-a}{a+b} - 1) - \tfrac{1}{6}(a+b)((\tfrac{p-q+b}{a+b})^3 + 5\tfrac{p-q+b}{a+b})$$

$$\ge \tfrac{1}{2}(\tfrac{\sqrt{n}}{u} - 1)(\tfrac{t}{r}(r-1)\sqrt{n} + \tfrac{a}{r} + b - a)\left(\frac{\tfrac{t}{r}(r-1)\sqrt{n} + \tfrac{a}{r} + b - a}{a+b} - 1 \right)$$

$$- \tfrac{1}{6}(a+b)\left[\left(\frac{\tfrac{u}{s}(s-1)\sqrt{n} + \tfrac{b}{s} - a + b}{a+b} \right)^3 + 5\left(\frac{\tfrac{u}{s}(s-1)\sqrt{n} + \tfrac{b}{s} - a + b}{a+b} \right) \right] \quad .$$

There is a term in $n^{3/2}$ in the above, namely

$$\left(\frac{1}{2u} \cdot \left(\frac{t}{r}(r-1)\right)^2 / (a+b) - \frac{1}{6}(a+b)\left(\frac{u(s-1)}{s(a+b)}\right)^3\right) n^{3/2} \quad,$$

This coefficient is not always positive. (Consider arbitrarily large u .) However, the two main terms of the coefficient are always positive, since $r,s > 1$. The first term is proportional to t^2/u , the second to u^3 . Thus one can choose suitably small t and u to ensure that the first term exceeds the second, so that their difference is then positive.

This completes the proof.

In the case of Shellsort with Hibbard's sequence, the parameters are $r = s = 2$, $a = 0$, $b = 1$.

To verify that Hibbard's sequence satisfies the conditions of the lemma with these parameters, note that for any p_j in the sequence, the interval $[p_j-a, p_j+b]$ is $[2^k-1, 2^k]$ for some k , and thus every larger element 2^i-1 is contained in the interval $[m(2^k-1), m2^k]$ where $m = 2^{i-k}$. Indeed, this condition will be satisfied for any "fuzzy" geometric progression with r being an integer > 1 and $s = r$. The other conditions are easily verified.

Chapter 3

An $n \log^2 n$ Shellsort

We show in this chapter that, using the sequence $2^p 3^q < n$, p, q non-negative integers, Shellsort takes $\frac{1}{2} n (\log_2 n)(\log_3 n)$ steps, and also admits of a simplification in which the innermost loop can be replaced by one instruction.

Let us establish some properties of arrays that are both 2-ordered and 3-ordered.

Lemma 3.2. If $p > 1$, then p is representable as a sum of 2's and 3's ; that is, there are non-negative integers r, s, with $p = 2r + 3s$.

Proof. If p is even, then $p = 2(p/2)$. Otherwise, $p = 2(\frac{p-3}{2}) + 3$.

Corollary 3.3. If A is 2-ordered and 3-ordered, it is p-ordered for all $p > 1$.

Proof. Follows from Lemma 3.2 and the transitivity of $\leq$.

Lemma 3.4. If A is 2-ordered then for all j satisfying $1 < j < n$, either $A[j-1] \leq A[j]$ or $A[j] \leq A[j+1]$.

Proof. If not, then $A[j-1] > A[j] > A[j+1]$, a contradiction.

An immediate corollary is that if A is p-ordered for all $p > 1$, no element $A[j]$ may be a member of more than one inversion, and every inversion must involve two adjacent elements. Thus to sort a 2-ordered and 3-ordered array, it suffices to swap the inverted adjacent pairs, which can be done in one pass, during which each of the $n-1$ adjacent pairs are compared and exchanged if necessary.

All of the above applies equally well to the p-chains of an array. In particular, when A is 2p-ordered and 3p-ordered, its p-chains are 2-ordered and 3-ordered, which means we can p-sort as above, and only take $n-p$ comparisons and exchanges.

Applying all this to Shellsort, we then deduce that any sequence will serve our purposes if (a) it contains 1, and (b) if p is in the sequence p is preceded by $2p$ and $3p$. Furthermore, we only need use those elements less than n. The smallest such sequence contains every number of the form $2^p 3^q < n$, for integer $p,q \geq 0$. The inequality $2^p 3^q < n$ can be written as $p/\log_2 n + q/\log_3 n < 1$, an inequality linear in p and q.

Estimating the length L of this smallest sequence is made easy using a geometric argument. In Figure 3.1, associate the lattice point (p,q) with the element $2^p 3^q$. The three bounds $p \geq 0$, $q \geq 0$ and $p/\log_2 n + q/\log_3 n < 1$ define the three sides of a triangle.
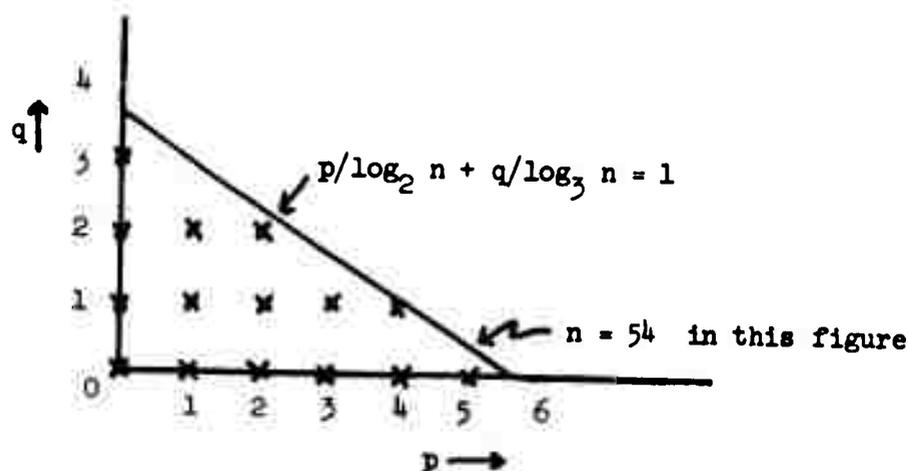
Figure 3.1. A triangular array of elements of $\{2^p 3^q | p,q \geq 0\}$.

In Figure 3.1, the p and q intercepts of the sloping line are $\log_2 n$ and $\log_3 n$ respectively, whence the area of the triangle is $\frac{1}{2} \log_2 n \log_3 n$, a first approximation to L which we improve thus:

We cla'm that the interior of the triangle is completely covered by those unit squares whose lower left vertices are lattice points in or on the triangle (other than on its hypotenuse). For suppose (x,y) is in the triangle. Then (x,y) is covered by the square whose lower left vertex is $(\lfloor x \rfloor, \lfloor y \rfloor)$. But $(\lfloor x \rfloor, \lfloor y \rfloor)$ is easily seen to be in or on the triangle but not on the hypotenuse (since (x,y) is not on the hypotenuse). This proves the claim.

But those lattice points in the claim correspond exactly to the elements of our sequence. Hence there are at least $\frac{1}{2} (\log_2 n)(\log_3 n)$ elements in the sequence, since the area of the squares corresponding to these elements exceeds the area of the triangle. So our first approximation turned out to be in fact a lower bound.

29

This bound is far from attainable, since the boundary of the squares is very ragged near the hypotenuse, as in Figure 3.2 (a).
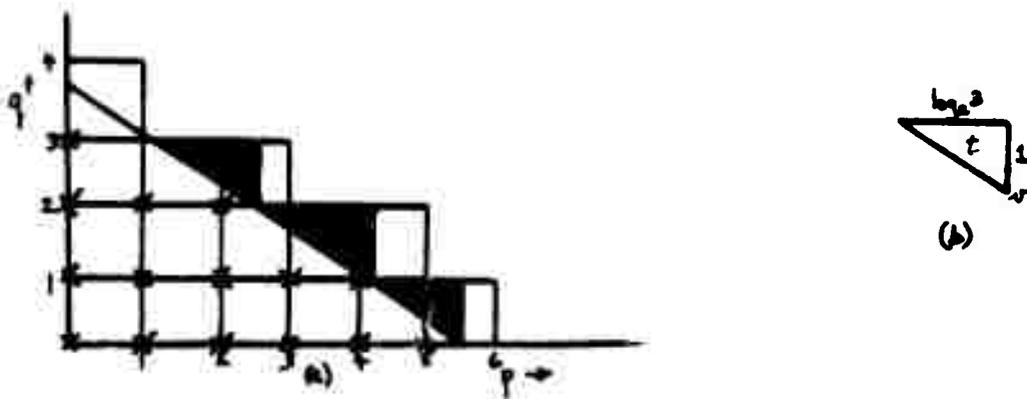
Figure 3.2(a). A triangle covered by squares.

(b). A triangular extension.

One way to improve the bound is to give the triangle a more ragged hypotenuse. Take copies of the triangle of Figure 3.2(b) and paste them over the dark regions of Figure 3.2(a). We need $\lfloor \log_3 n \rfloor$ such triangles, each of area $\frac{1}{2} \log_2 3$, giving a total area of more than $\frac{1}{2} (\log_3 n - 1) \cdot \log_2 3$, which is more than $\frac{1}{2} \log_2 n - 0.8$.

To see that the interior of these triangles are all covered by the squares, note that if (x,y) is <u>in</u> some small triangle t , then $(\lfloor x \rfloor, \lfloor y \rfloor)$ must be on the horizontal lattice line passing through the lower right vertex v of t , and must be to the left of v , and hence inside the main triangle.

Hence the area of the squares, and thus the number of elements of the sequence, is bounded below by $\frac{1}{2} \log_2 n(\log_3 n + 1) - 0.8$.

To get an upper bound we use a very similar argument. First replace the hypotenuse by one almost parallel to it:

$$p/\log_2(n-1) + q/\log_3(n-1) = 1 \quad .$$

(This uses the fact that $2^p 3^q \leq n-1$, and simply gives us a marginally tighter upper bound.) Now associate with each lattice point p <u>in</u> the new triangle or <u>on</u> its hypotenuse (but not on the p or q axes, and hence not on the endpoints of the hypotenuse) the unit square whose upper right vertex is p . These squares are all clearly <u>inside</u> the triangle. We left out $\lfloor \log_2(n-1) \rfloor + \lfloor \log_3(n-1) \rfloor + 1$ points on the axes (the "1" is the origin). Hence there are less than

$\frac{1}{2} \log_2(n-1)\log_3(n-1) + \lfloor \log_2(n-1) \rfloor + \lfloor \log_3(n-1) \rfloor + 1$ elements in the sequence.

The reader may check that the earlier "ragged" argument still works, this time using the reflection of the triangle of Figure 3.2(b), and <u>removing</u> copies of this triangle from the main triangle. (The hypotenuses of the small triangles still coincide with the hypotenuse of the main triangle.) Hence we may subtract $\frac{1}{2} \log_2 n - 0.8$ from our upper bound.

In conclusion, we may bound the length L of our sequence by

$$\frac{1}{2} \log_2 n(\log_3 n + 1) - 0.8 \; < \; L \; < \; \frac{1}{2} \log_2(n-1)(\log_3(n-1) + 1 + 2\log_3 2) + 1.8$$

where $2\log_3 2$ is about 1.26 . The width of this bound is about $\log_3 n$ (note that $\log_2(n-1) = \log_2 n - n^{-1}\log_2 e$) . So $\frac{1}{2}\log_3 2(\log_2 n)^2$, or $0.315(\log_2 n)^2$, is a good approximation to L .

It follows immediately that $0.315 \, n(\log_2 n)^2$ is a good estimate of the best-case, worst-case and average number of comparisons required by Shellsort using this sequence. This is of interest in that, at least asymptotically, it is the fastest known way to Shellsort, in the sense that the other Shellsorts we could analyze take time proportional to $n^{3/2}$ on some arrays. However, it is of more interest for a reason we will pursue in Chapter 4. Here we shall confine ourselves to some remarks about this particular Shellsort.

There are many orders in which the sequence may be generated and yet have $2p$ and $3p$ precede p. One computationally convenient sequence generates all $2^p 3^q$ for $p+q = \lfloor \log_2(n-1) \rfloor$, then all $2^p 3^q$ for $p+q = \lfloor \log_2(n-1) \rfloor - 1$, and so on until $p+q$ vanishes. Thus $2 \cdot 2^p 3^q = 2^{p+1} 3^q$ and $3 \cdot 2^p 3^q = 2^p 3^{q+1}$, both of which are in the set preceding the one containing $2^p 3^q$. Within a set for which $p+q = i$, start with 2^i, then multiply by $3/2$ until the result is odd or $\geq n$.

Expressing this in near-ALGOL, we have

```
for i := 2 ↑ ⌊ 2 ↓ (n-1) ⌋ , i ÷ 2 while i ≥ 1 do
   for j := i, (3 × j) ÷ 2 while j mod 3 = 0 and j < n do
      for k := 1 step 1 until n-j do
         if A[k] > A[k+j] then swap (A[k],A[k+j]);
```

Here $a \downarrow b$ is an abbreviation for $\log_a b$, which is an abbreviation for $\ln(b)/\ln(a)$, and $\lfloor a \rfloor$ is an abbreviation for entier(a). The reader is asked to believe that $j \bmod 3 = 1$ if j was odd before doing $j := (3 \times j) \div 2$, otherwise it is 0 (an inelegant jump over a hurdle of ALGOL 60).

An interesting improvement to the algorithm capitalizes on Lemma 3.4, by avoiding a test following an exchange.

$\underline{\text{for}}$ i := 2 $\uparrow$ $\lfloor$ 2 $\downarrow$ (n-1) $\rfloor$, i $\div$ 2 $\underline{\text{while}}$ i $\geq$ 1 $\underline{\text{do}}$

$\underline{\text{for}}$ j := i, (3 $\times$ j) $\div$ 2 $\underline{\text{while}}$ j $\underline{\text{mod}}$ 3 = 0 $\underline{\text{and}}$ j < n $\underline{\text{do}}$

$\underline{\text{for}}$ k := 1 $\underline{\text{step}}$ 1 $\underline{\text{until}}$ j-1 $\underline{\text{do}}$

$\underline{\text{for}}$ l := k $\underline{\text{step}}$ j $\underline{\text{until}}$ n-j $\underline{\text{do}}$

$\qquad$ $\underline{\text{if}}$ A[l] > A[l+j] $\underline{\text{then}}$ $\underline{\text{begin}}$ swap (A[l],A[l+j]);

$\qquad\qquad\qquad\qquad\qquad$ l := l+j

$\qquad\qquad\qquad$ $\underline{\text{end}}$

Whereas before we simply corrected all p-inversions, starting at the left, we now correct all the inversions of one p-chain before going on to the next. This change is necessary if we are to take advantage of Lemma 3.4.

Suppose the body of the inner loop takes 1 unit of time if A[l] $\leq$ A[l+j] and 2 otherwise and that all other operations are negligible. Then the timing of this version of the algorithm becomes remarkably independent of how well ordered the initial array was, since for all but the last two elements of each p-chain, if the body ever takes 2 units, this increase is offset by skipping the next comparison. Thus each pass will require between n-p and n units of time (depending on how many p-chains had their last two elements inverted) which is quite a small range for most p's in the sequence, in comparison with n , for reasonably large n .

The numbers 2 and 3 are not the only possible choice for this algorithm. In fact, any set $x_1,\ldots,x_m$ will do if their greatest common

divisor is $\perp$, by Lemma 2.2. The corresponding sequence is the set of numbers less than n that have only $x_1, \ldots, x_m$ as factors, in descending order, say, giving a timing of $O(n \log^m n)$. The Shellsort of Chapter 1 must be used, since we now need the inner loop again. Preliminary investigation of various sets seems to indicate that $\{2,3\}$ is the best choice, as far as the upper bound is concerned. However, other sets with two elements may conceivably give a faster running time on the average.

Chapter 4

A Shell Sorting Network

4.1 Sorting networks

The most interesting feature of the algorithm of Chapter 3 is that
it suggests a fast sorting network. A **sorting** **network** is a set of
comparators wired such that when an array of numbers is presented to
the input terminals of the network, the same numbers rearranged in
ascending order are presented at the output terminals. A **comparator** is
a two-input two-output sorting network whcih may be treated as a black
box for the purpose of designing sorting networks with them as the basic
building blocks. The basic difference between a sorting network and a
general-purpose computer programmed to read, sort and output arrays is
that whereas the control structure of the computer is inherently serial,
forcing comparisons and exchanges to be done one at a time, the network's
control structure is defined by the comparators alone and can be made
highly parallel; all that is required in the way of control is that a
comparator start work just when it has received both of its inputs.

By way of example, the illustrated network of Figure 4.1 will sort
four numbers with 3 units of delay.
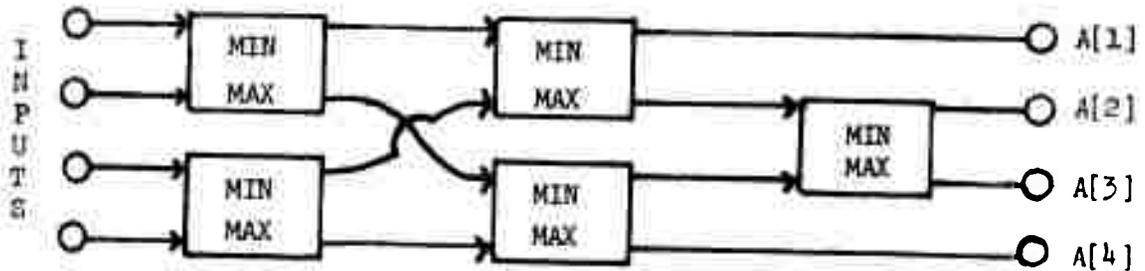
Figure 4.1. Sorting Network for four elements.

To show that this particular network indeed sorts, note that output 4 must get the maximum of the inputs, and output 1 the minimum. Hence $A[1] \leq A[2]$, and $A[3] \leq A[4]$. The last comparator guarantees $A[2] \leq A[3]$, and these three relations then mean that the array is sorted.

A convenient representation for a comparator and its wiring is as in Figure 4.2:
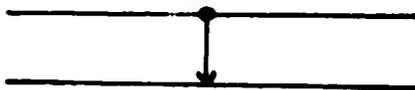
Figure 4.2. An abbreviated comparator.

where the vertical line denotes the comparator and the arrow denotes the direction the larger number goes. Thus our previous example would be drawn as in Figure 4.3.
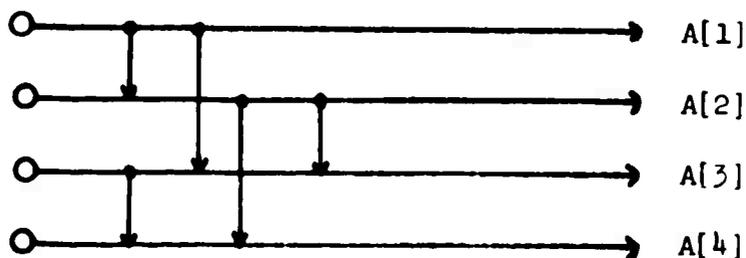
Figure 4.3. An abbreviated sorting network for four elements.

Note that the inputs to the final comparator as drawn here are inverted with respect to the corresponding inputs in the other diagram. This does not affect its operation; indeed the two inputs to a comparator never need be distinguished, since max and min are commutative functions.

The important thing about sorting networks is that they permit considerable parallelism, since up to $n/2$ comparators can be working simultaneously on n lines. One might deduce from this that, since there exist serial algorithms taking time $n \log n$, one could do $n/2$ of these steps at a time in a sorting network, taking $O(\log n)$ units of time. This deduction breaks down because it is not necessarily possible to predict which $n/2$ comparisons to do at any given time without knowing in advance the outcome of some of those same comparisons. So far, the best asymptotic timing to date has been Batcher's algorithm [Batcher 1968], which takes $\frac{1}{2} \log_2^2 n$ units of delay asymptotically (where a unit is the time to compare and exchange two elements) and uses $\frac{1}{4} \log_2^2 n$ comparators asymptotically. (See also Van Voorhis [1971].)

Further discussion of sorting networks can be found in Floyd and Knuth [1970].

4.2 Shellsort with standard comparators

In the algorithm of Chapter 3, where an array was sorted by being $2^p 3^q$-sorted for all integers $p, q \geq 0$ with $2^p 3^q < n$, it was noted that for each p and q, the corresponding $2^p 3^q$-sorting involved only correcting a small number, $\lfloor \frac{n}{2} \rfloor$, of adjacent inversions. It is possible to do all of this work in two stages of parallelism, by first simultaneously comparing every even-numbered location (numbering the elements of a p-chain $1,2,3,\ldots$) with its predecessor, and then doing the same for odd-numbered elements. For a single p-chain with 6 elements we would have Figure 4.4.
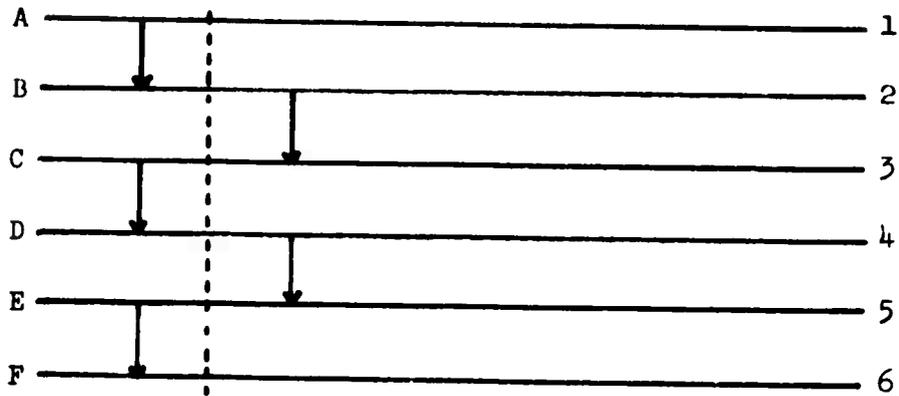
Figure 4.4. A sorting network for a 6 element p-chain that is already 2- and 3-ordered.

Since the p-chains are independent for a fixed p , we can extend this parallelism to the whole array. Thus if the above example were of a single 3-chain in an array of 16 elements, the whole 3-sorting stage would involve the network of Figure 4.5:
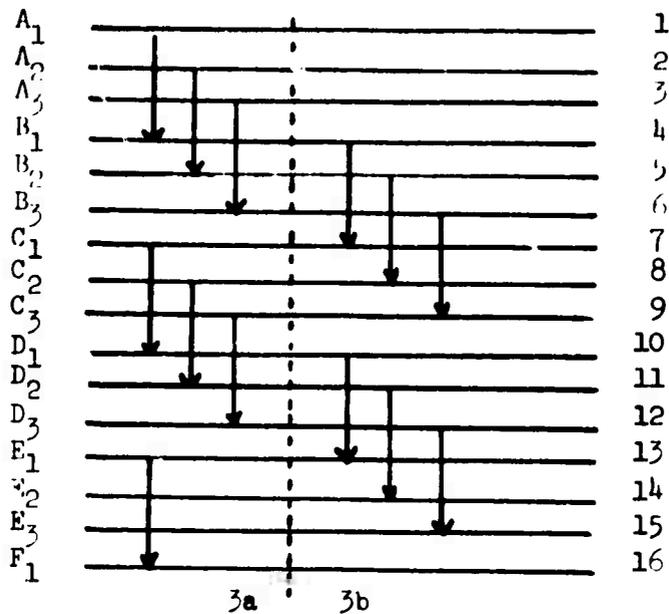
Figure 4.5

where 3a and 3b denote the first and second stages of 3-sorting respectively.

A complete sorting network for 8 elements would use the characteristic

sequence 6, 4, 3, 2, 1 , as in Figure 4.6.
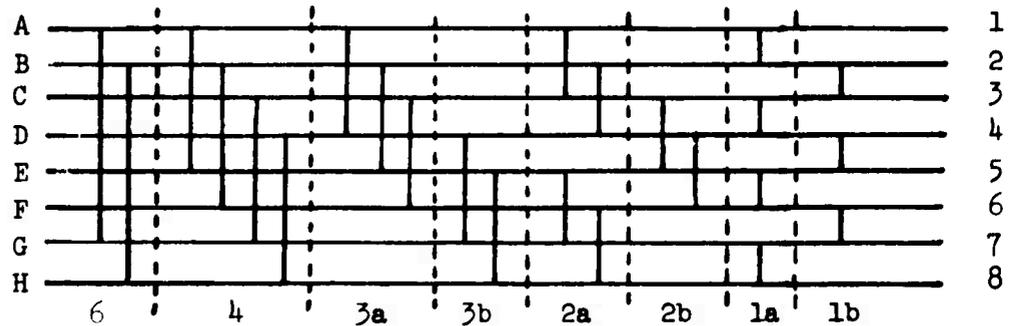
Figure 4.6

From the analysis of Chapter 3, we know that there are about

$\frac{1}{2}$ $(\log_2 n)(\log_3 n)$ elements in the sequence, whence we need at most

$(\log_2 n)(\log_3 n)$ stages of parallelism, each requiring one unit of delay.

So this network takes about $\frac{1}{1.58} \log_2^2 n$ units, slower than Batcher's

algorithm by a factor of 1.26 .

4.3 A faster network

In the above network, for each p in the characteristic sequence,

we had to partition into two groups those comparators responsible for

p-sorting, in order to avoid having two comparators working on the same

line simultaneously. But Lemma 3.4 tells us that two such comparators may

not both swap their inputs. This suggests a conjecture, that the comparators

can operate in parallel anyway, perhaps with some modification to the design

of the comparators. Were this possible with no increase in delay of a

comparator, our network would then function with $\frac{1}{2} \log_2 n \log_3 n$ units of delay asymptotically, which would represent an improvement of a factor of $\log_2 3 \doteq 1.585$ over the networks described by Batcher [1968].

Unfortunately, there is no universally applicable way of doing this. For if there were, we could apply it to the particular case where the quantities to be sorted may take on only the values 0 and 1 , and the only available components are two-input AND and OR gates, each with a delay of 1 unit. Now we may readily build a comparator for this domain as in Figures 4.7 and 4.8.
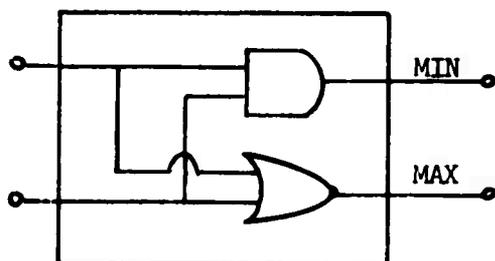
Figure 4.7. A comparator for zeroes and ones.
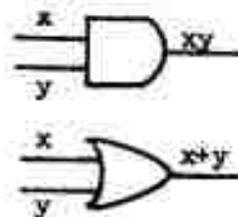
Figure 4.8. Notation.

Since our conjecture refers only to that part of a sorting network corresponding to a single element of the $2^p 3^q$ sequence, we need only consider, in isolation, the realization of this part as AND and OR gates. Moreover, within that part, we only need consider a single $2^p 3^q$ - chain as in Figure 4.9, since chains are not connected to each other within that part.
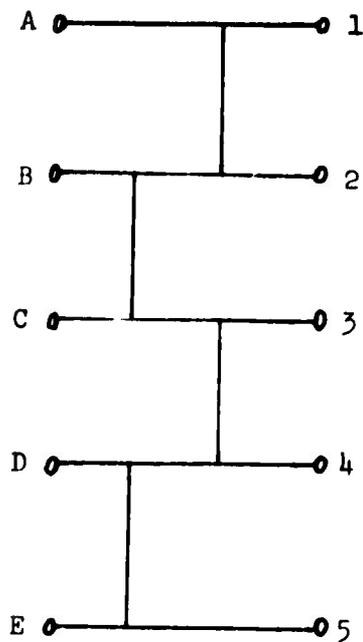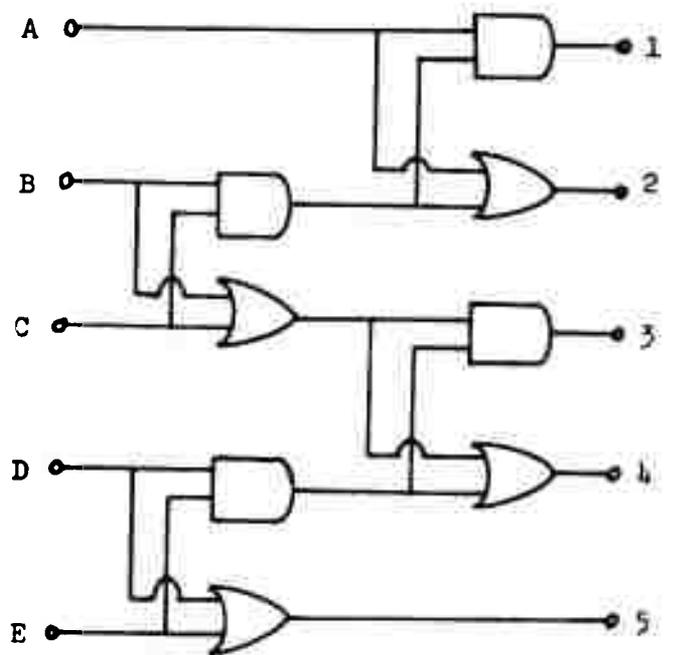
(a)   (b)

Figure 4.9. (a) A sorting netwcrk for one p-chain (as in Figure 4.4,
                  minus one input).

           (b) Implementation of Figure 4.9(a) using Figure 4.8.

The delay of this stage of the sorting network as implemented in

Figure 4.9(b) is just 2 units. In order to improve on this we must reduce

the delay to 1 unit. But then output 3 of Figure 4.9(a) would have to be

the output of some gate with two inputs chosen from the inputs B , C , D

and E (the only inputs that could affect output 3), since only 2-input

gates are available. But output 3 is a non-trivial function of B , C

and D , even under the condition that the input is p-ordered for all

$p > 1$. (Consider BCD = 001 vs. 101 (for B), 001 vs 011 (for C),

and 010 vs 011 (for D).) Hence it is not possible to reduce the delay

41

of this stage to 1 unit. (The reader may readily extend this result to the case when multiple-input AND and OR (and even NAND and NOR) gates are available.)

So the best we can hope for is that under some conditions we may be able to take advantage of Lemma 3.4. To show that our conjecture is not completely without grounds we shall give in detail an example of a situation much closer to real-life problems and state-of-the-art technology than the foregoing somewhat artificial one, and show that in this situation we may come closer to realizing the desired factor-of-two speed-up.

A more common domain for sorting purposes is that of the integers. A common representation for this domain is the familiar binary notation. Let us assume that we want to sort n words of w bits each, and that each word represents an unsigned binary number.

One way to implement a sorting network for this situation is to have each comparator process all w bits of each of its two inputs in parallel before outputting anything. This is fast but expensive, since each bit requires some hardware of its own. At first sight, serial processing (most significant bit first) would seem to involve a speed decrease of a factor of about w . However, it should be clear that as soon as a comparator has inspected a bit from each of its inputs, it may pass those bits on to the next comparator before it has seen any more of its input. (A problem that can arise here is that a comparator may not know at some time which of its inputs is the maximum. But in this case, all pairs of bits seen so far must have been equal, so that it doesn't matter which way it routes its output.) So the time required for

a serial network is really just that required by a parallel network, plus (rather than times) the time required to pass w bits (to be precise, w-1) through a comparator. Hence serial organization would appear to be economically sound here, and we shall assume it for our example.

Finally, let us assume that we have available NAND gates and NOR gates with any number of inputs, and flipflops. While this restricted repertoire does an injustice to the present state of the art of integrated circuits (where the effect of parasitic lead capacitances on timing dominates that of AND-OR-INVERT gate propagation delays in some cases), it will at least take advantage of the reader's presumed familiarity with the elements of traditional (and rather idealized) switching theory.

With our assumptions formulated, we shall exhibit an implementation of Figure 4.9(a) under these assumptions.

Since the input to the network of Figure 4.9(a) is p-ordered for all $p > 1$, and since the output is completely ordered, it follows that each output is just the median of its three closest inputs; e.g. output 2 is the median of A , B and C , output 3 is the median of B , C and D , and so on. Hence it suffices to implement Figure 4.9(a) as in Figure 4.10(a), where the output of each box labelled M' is the median of its inputs. (The 0 and 1 inputs at the top and bottom are fixed at those values throughout the operation, and thus correspond in their effect to values of $-\infty$ and $+\infty$ respectively.)
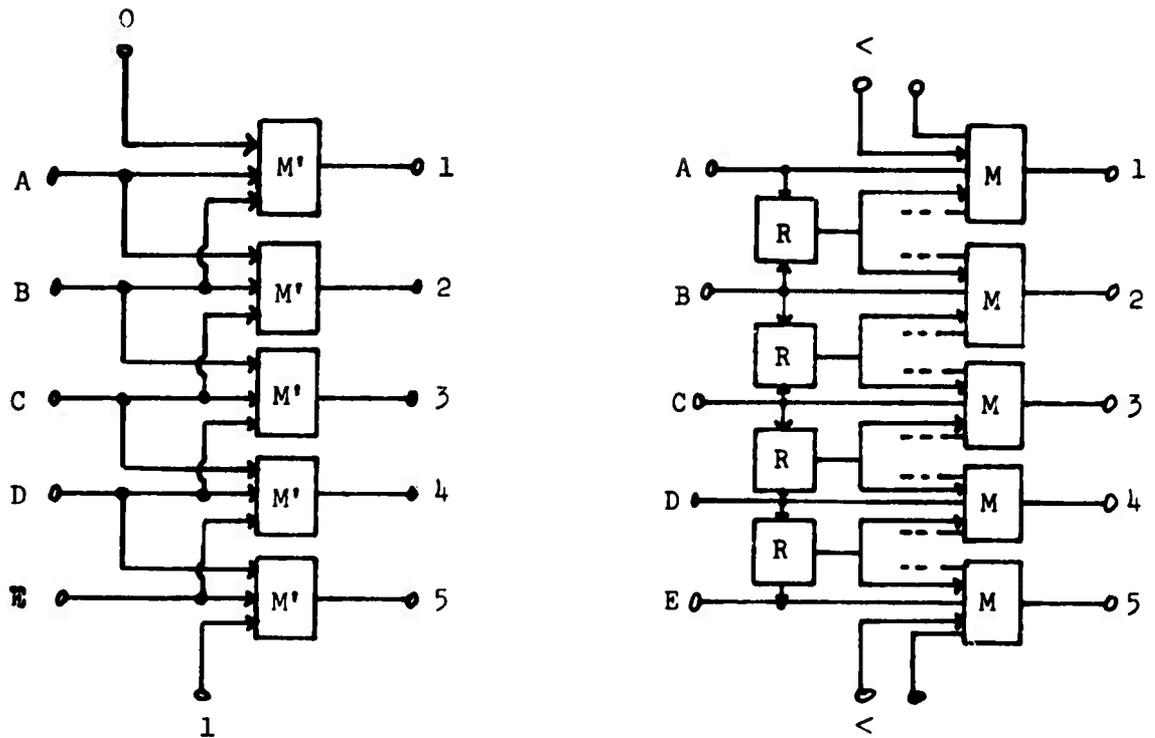
Figure 4.10 (a) Implementation of Figure 4.9(a) using median-finders.

(b) Same as (a) (some detail omitted) with registers R added.

The device M' must keep track of the relations between its inputs.
Because of our serial organization, these relations are a function of
time, in that they depend on how much of the incoming three words has
been seen. It suffices to remember (say for the M' connected to output 2)
whether $A < B$, $A = B$ or $A > B$, and whether $B < C$, $B = C$ or $B > C$.
Thus it would seem that each M' must be able to distinguish nine cases
(three of which cannot arise, as we shall see later, leaving six cases).
This is wasteful since two adjacent M' s could share the information
about their two common inputs. This immediately suggests Figure 4.10(b)
as a more economical organization. The connections of Figure 4.10(a) are

preserved (although their detail is omitted for clarity in Figure 4.10(b)).
Box M' is now replaced by box M , which no longer has the responsibility
of remembering what happens from one bit to the next. Instead, this
responsibility is delegated to the R boxes. Box M now consults its
three inputs and two R boxes as each new set of 3 bits, A , B , C ,
arrives. The output from an R box is 3-valued, viz. < , = or > ,
corresponding to whether R thinks that A < B , A = B or A > B .

There are two approaches to the timing of M . Either M may wait
until the R boxes have decoded their inputs before deciding which of
A , B and C is now the median, or M may decide to go ahead with the
new A , B and C bits but using the old states of the boxes R_A ,
corresponding to the situation up to the previous A , B and C bits.
That is, M may anticipate the next states of the R boxes, without
waiting on them. The merit of this approach is that the delay of the
whole stage is then reduced by an amount possibly as great as the delay
of R . We shall adopt this second strategy.

We may build R boxes as in Figure 4.11.
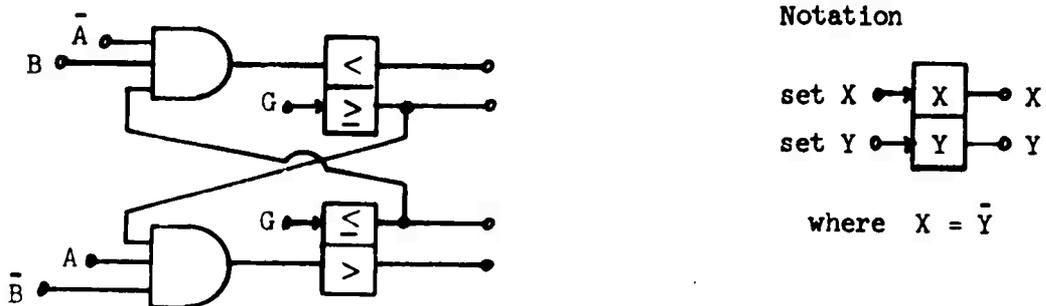
Figure 4.11.   Implementation of an R box, with 2 gates, 2 flipflops.

Note that the complemented inputs $\bar{A}$ and $\bar{B}$ may be derived from A , B respectively using inverters (single input NAND or NOR gates). However, it is highly likely that a practical design will interpose a flipflop between the output of a median finder and the next stage in order to control the movement of bits through the network, both to avoid one bit catching up with its predecessor and to ensure that the three inputs all arrive simultaneously at a median finder. Inherent in the design of flipflops made of NOR gates (the usual strategy) is the accessibility of both the flipflop's output and its complement. Thus the inverters would then not be needed.

The two AND gates may independently be replaced by NOR gates, with the appropriate changes to their inputs (that is, use the complemented value of each input instead). This follows from De Morgan's Law that $AB = \overline{\bar{A} + \bar{B}}$. We used AND gates to aid the reader's understanding of the circuit's operation.

The notation for flipflops should be self-explanatory. It was suggested to us by H. Stone and sidesteps the irrelevant issue of whether to use the Q or the $\bar{Q}$ output (a designation quite arbitrarily assigned by each flipflop's manufacturer, with R (reset) and S (set) inputs then named to correspond to this arbitrary choice) to denote some variable. Each half of the flipflop represents a buffer that "remembers" any logic level of 1 that arrives at its input. The juxtaposition of the two halves represents the fact that the buffer is made to "forget" (i.e., return to state 0) if a level of 1 arrives at the other buffer's input.

In the operation of the R box, input G (go) is momentarily set
to 1 prior to starting to sort, and then returns to 0 for the
remainder of the sorting operation. Thus every R box initially
supposes that both $A \leq B$ and $A \geq B$, that is, $A = B$.

By inspection of the circuit, as long as $A = B$ at the inputs, the
state of R will be undisturbed. Suppose $A = 0$ and $B = 1$ at some
time. If $A = B$ before this, then we must have $A < B$, and R digests
this fact by complementing the upper flipflop. The dual situation
obtains if $A = 1$ and $B = 0$.

Once one of the flipflops has been complemented, it is clear that
no further change of state of R is possible. The complemented flipflop
will never see another 1 at its G input, and the other flipflop's
input has been turned off by the complementing. So only three states
of R are possible, corresponding to $A = B$, $A < B$ and $A > B$, as
desired. These states are communicated to the outside world by four
outputs (abbreviated to one in Figure 4.10(b)) labelled $<$, $\geq$, $\leq$
and $>$ respectively.

Let us now proceed to a circuit for the M boxes, given by Figure
4.12. We use AND and OR gates for pedagogical reasons; an equivalent
circuit may be obtained by replacing every gate with a NAND gate; recall
De Morgan's Law that $A + B = \overline{\overline{A}\,\overline{B}}$.
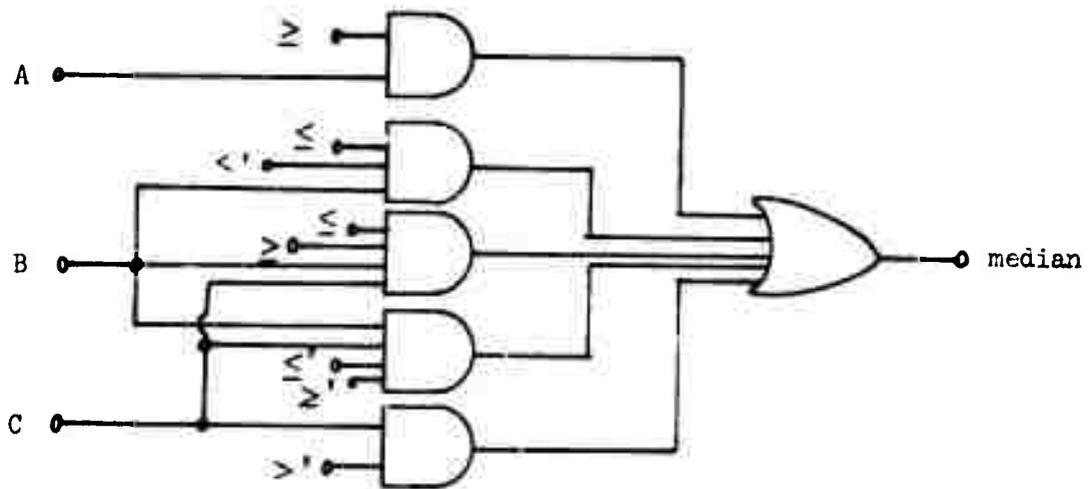
Figure 4.12. An implementation of a median finder M .

The unprimed inequalities labeling the input terminals denote the appropriate outputs of the R box that is to be connected between A and B . Call this R box simply R . The primed inequalities are for the R box between B and C . Call this box R' .

To verify that this circuit works, it suffices to enumerate the possible pairs of states of R and R' . The details are encapsulated in Figure 4.13.

(R,R')	(=, =')	(=, <')	(<, =')	(>, <')	(<, <')	(<, >'');
Non-zero inputs	$\geq \leq \geq'$ $\leq'$	$\geq$ $\leq$ $<'$ $\leq'$	$<$ $\leq$ $\geq'$ $\leq'$	$\geq$ $>$ $<'$ $\leq'$ $\geq'$	$\leq$ $<$ $<'$ $\leq'$	$\leq$ $<$ $\geq'$ $>'$
Equivalent Circuit						
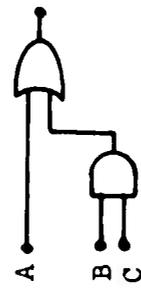
Figure 4.13. Equivalent circuits for the possible states of (R,R').

49

Three cases, $(>, >')$, $(>, =')$ and $(=, >')$ are not shown because they cannot occur. Each of these cases implies that $A > C$, contradicting Lemma 5.4. This remark is independent of what bits are seen later, even if eventually $(>, =')$ becomes $(>, <')$, say. The explanation in this case is that if there is so far no way to distinguish B from C , yet R can tell that $A > B$ for some reason, then we must deduce that $A > C$ for the same reason.

The figure shows those inputs that are set to 1 for each of the 6 possible cases. By eliminating those AND gates of Figure 4.12 that have a 0 input, and then simplifying the remaining circuit, it is easy to arrive at the equivalent circuits shown for each case.

To verify that the equivalent circuits are the desired ones, note that we have essentially reduced the problem to the case when the data to be sorted can have only the two values 0 and 1 . The median finder's responsibility is simply to decide which of three bits is the output. It is the responsibility of R and R' to decide which equivalent circuit is required for any particular set of 5 bits.

For the case $(, ')$, we clearly want the fullblown circuit of Figure 4.9(b). Inspecting output 2 of that circuit shows that we have the correct equivalent circuit.

For the case $(=, <')$, C cannot be the median, so we want $max(A,B)$. Figure 4.8 verifies this equivalent circuit.

In the case $(<, =')$, A cannot be the median, so we want $min(B,C)$. Again Figure 4.8 confirms the circuit.

The remaining cases correspond to $B < A < C$, $A < B < C$ and $A < C < B$ respectively, giving medians A , B and C respectively. So the circuit of Figure 4.12 does indeed work.

Note that if R is in state $<$, the output of M is independent of input A, and similarly for input C when R' is in state $<'$, as can be seen from Figure 4.13.

It follows that in Figure 4.10(b) the top input of the top M box need not be set to 0 as in Figure 4.10(a), and similarly for the very bottom input. Thus these two inputs may be tied to any convenient terminal in practice, provided the terminal's voltage does not interfere with the otherwise correct functioning of the gates thereby attached.

The crucial question now is that of speed. In particular, how does this circuit compare with the fastest possible circuit for a standard comparator for use in Batcher's network? Any answer to this will almost certainly have to depend on a detailed knowledge of the relative speeds of the available devices for building comparators.

Figure 4.14 exhibits a possible implementation of a comparator. The principle of operation of the structure in Figure 4.14(a) is the same as that of Figure 4.10(b). The only difference is that in place of the three-argument median finders, we now have _max_ and _min_ finders, each with only two arguments. The circuits for MIN and MAX are analogous to that of Figure 4.12, and the style of argument represented by Figure 4.13 carries over to these circuits quite trivially. As before, NAND gates may be used throughout. It is interesting to note that although the circuit for M was developed independently of those for MIN and MAX, the MAX circuit is obtainable directly from the M circuit by removing the bottom three AND gates of M and the $<'$ input of the second AND gate (that is, everything to do with input C). The MIN circuit is almost as easily obtained (together with some simplification) by suppressing anything to do with input A .

(a)

(b)

(c)
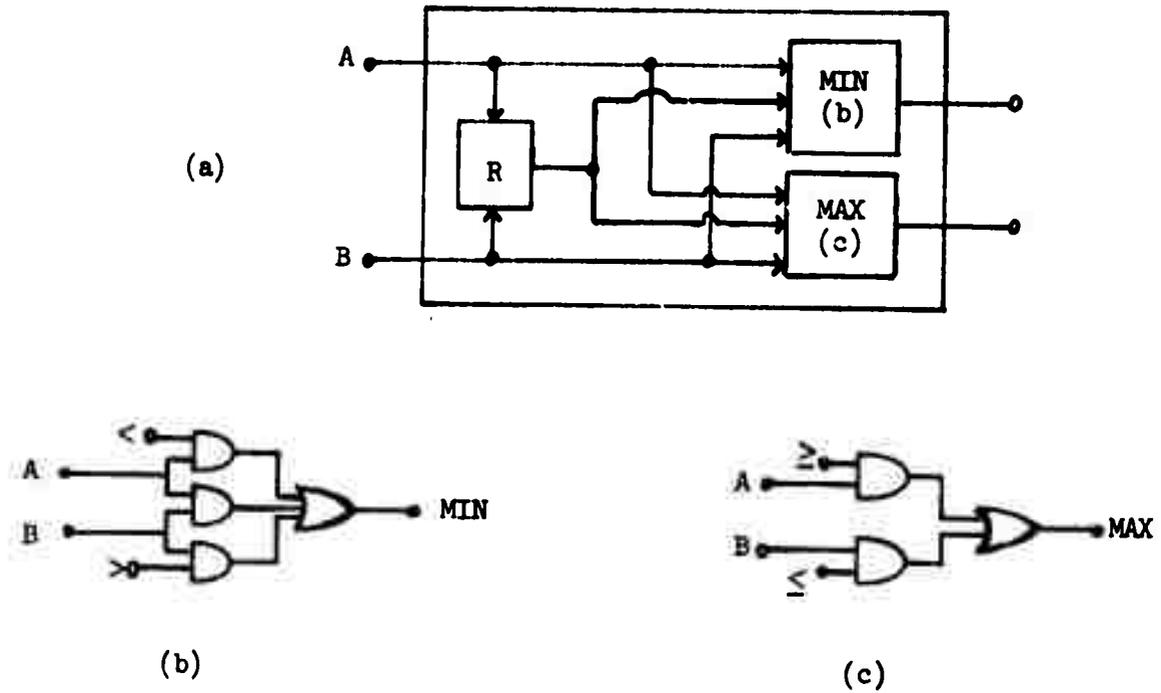
Figure 4.14   (a)   Structure of a comparator.

(b),(c)   MIN, MAX circuits.

We conjecture that the circuit of Figure 4.14 is very close to the fastest possible for a standard comparator, using the existing technology based on NAND and NOR gates. In support of this, we can prove that the two-gate delay of this circuit cannot be reduced to a one-gate delay. For if it could, each gate (necessarily one for each output) would have to be a NAND or NOR gate. But neither these nor AND nor OR gates are suitable. Consider the MIN output. This cannot be the AND or NAND of the inputs A and B , since there are occasions when one of A or B is 0 yet a 1 output is required (e.g. when R knows A < B , the current A bit is 1 and the B bit is 0) or a 0 output is required

(e.g. both A and B are simultaneously 0 at some time). A dual
argument says that the MIN output cannot be the OR or NOR of the inputs
A and B . A fortiori, the MIN output cannot be a single-gate function
of A , B and the state of R .

Hence the question of optimality of the circuits of Figure 4.14
involves mostly very technology-dependent issues such as the effect of
fan-in and fan-out on gate propagation delays, the ratio of turn-on
to turn-off delays (quite significant with bipolar   transistor TTL
technology) and whether it is possible to wire-OR gate output (as with
tri-state logic for example; this gives the effect of having OR gates
with no delay). Each gate in Figure 4.14 has a fan-in of at most 3 ,
and a fan-out of at most 4 . It would seem unlikely that this could be
significantly improved, especially in view of the fact that the delay of
currently available gates as quoted by their manufacturers is independent
of the fan-in for up to about six inputs, and increases by about 5 percent
(for fast gates) for each extra device loading the output, up to a fan-out
of about 10 .

If wired-OR is possible, this gives all our circuits (except for R)
the effect of one gate of delay, so the issue of the availability of
wired-OR logic would not appear to significantly damage our conjecture.
The issue of turn-on/turn-off delays is probably too transistor-dependent
to be worth discussion here. The reader is challenged (if he is
interested in technology-dependent arguments) to try to show constructively
that the ratio of turn-on to turn-off delays (within reasonable limits)
affects our conjecture.

Of course, none of this is very relevant if the delay of R

exceeds that of the other devices. In this case, our median-finder

is as fast as our comparator (ignoring the fan-out of R for the moment),

and our comparator in turn is probably close to optimal, in view of the

triviality of the circuit for R . Taking the fan-out of R into

consideration, this is at most 2 for each output from R in Figure 4.14

(counting the connections within R), and at most 3 in Figure 4.12

(provided we are using NOR gates in R ; with AND gates as shown, the

fan-out of the $\leq$ and $\geq$ outputs becomes 4). So a delay of at most

5 percent that of a gate (we can build flipflops from NOR gates), and

hence less than 3 percent of the whole circuit, is about the main

difference in timing between these circuits.

In the event that R turns out to be faster than our median finder,

we need to show that the latter is not much slower than our MAX and MIN

circuits. The only significant difference is that the fan-out of the

output of M is 5 more than that for our comparator outputs (6 if

we don't have flipflops at the output, for then R will require two

inverters). To get around this disparity, we can "move the fan-out back

a gate", by duplicating or triplicating the circuitry for the OR gate in

each of our circuits, at the cost of increasing the fan-out of the AND

gates. The optimum appears to be triplication for M and duplication

for each of MIN and MAX, independently of whether we use flipflops

between stages. (The flipflops if they are present must be duplicated

along with the OR gates.) Without flipflops, the optimized "accumulated

fan-out" (maximum fan-out of any AND gate plus maximum fan-out of any OR

gate) is 7 for M (3 for the ANDs, 4 for the ORs) and 5 for

54

MIN/MAX (2 for ANDs, 3 for ORs). With flipflops, it is 6 for M
(3 for the ANDs, 3 for the ORs) and 4 for MIN/MAX (2 for ANDs,
2 for ORs). The situation for M with flipflops is shown in Figure 4.15.
In both cases the difference between M and MIN/MAX is 2 , corresponding
to a difference in delay of about 10 percent of a gate, or at most
5 percent of a comparator without flipflops, even less for one with
flipflops.

3 + 3 + 3 = 9 = total load of next
stage
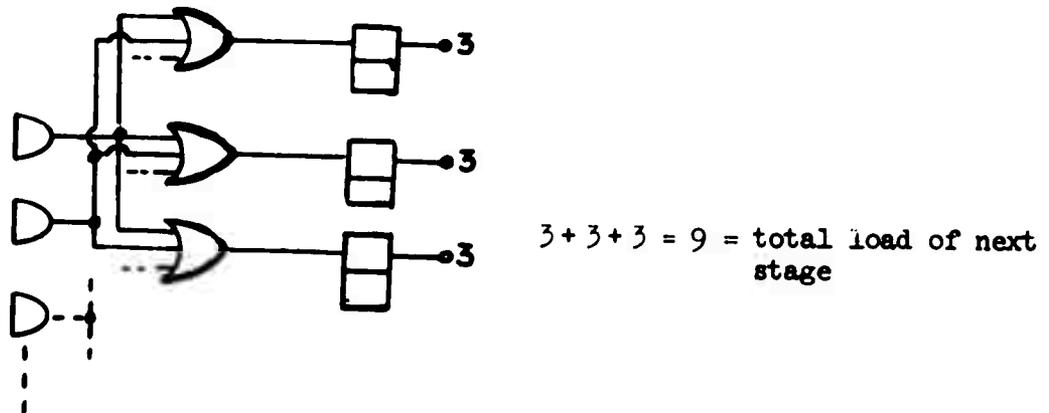
Figure 4.15.   M with triplicated OR gates and flipflop buffers.
(Not all AND gates shown.)

In conclusion, there seems little reason to doubt that with
state-of-the-art technology, we can build median-finders whose speed is
within 5 percent of the speed of the best comparators. Thus, using our
$2^p 3^q$ network, we may improve on Batcher's network by a factor of between
1.5 and 1.585 .

Chapter 5

Epilogue

5.1. Summary and Suggested Problems.

For each chapter, we shall summarize its result: and suggest problems associated with that chapter.

In Chapter 2, we gave an upper bound (namely $O(n^{3/2})$) on the worst-case time for Shellsorts that use "fuzzy" geometric progressions with short coprime subsequences throughout. In addition we showed that when these progressions had an integer common ratio, the upper bound could not be improved other than to within a constant factor. This leaves open the following problems.

1. What is the constant factor (as a function of the given characteristic sequence) for the worst case of Shellsort with the integer-common-ratio sequences?

2. Can the $O(n^{3/2})$ bound be improved if the ratio is not an integer, but, say, $\sqrt{2}$?

3. What other properties do Shellsorts with geometric sequences have? For example, what is the mean and the variance of the time for Shellsort with Hibbard's sequence, given some frequency distribution for the data?

In Chapter 3, we showed that $O(n^{3/2})$ is certainly not the ultimate fate of Shellsort. We did this by exhibiting one sequence for which Shellsort takes time $O(n \log^2 n)$. Some problems this raises are:

4. What is the ultimate speed of which Shellsort is capable? (Is $O(n \log^2 n)$ the best possible?).

5. What is the average time for Shellsort using sequences of the form $2^p 5^q$, etc.? Is it better or worse than that for the $2^p 3^q$ sequence?

In Chapter 4, we converted the serial algorithm of Chapter 3 into a highly parallel one. Our arguments were, unfortunately, based on the state-of-the-art of the electronics industry. We showed that there was no universal way to eliminate this dependency, by describing a rather trivial environment where our method failed to compete with Batcher's method. This raises these questions.

6. What environments less trivial than the domain $\{0,1\}$ also handicap our method?

7. Are there environments for which our method is still better than Batcher's, but only by, say, a factor of 1.2?

8. What is the advantage of our method when we can afford to build parallel comparators? (This costs many times more, with a disproportionately small return on the investment, making this question of interest mainly to the very rich.)

9. Is it a coincidence that all attempts to build faster sorting networks have resulted in networks that take time $O(\log^2 n)$, or is this the asymptotic lower bound?

5.2. Conclusions and Perspective.

The unifying basis for this thesis is the sorting technique
described by Shell [1959], generalized of course to consider a larger
class of characteristic sequences for Shellsort than the one considered
in Shell's original paper. The sequences we considered in detail could
be classified respectively as first- and second-order geometric progressions,
where an m-th order progression has m distinct ways of generating new
elements of the progression from old ones (e.g. multiplying by either 2
or 3, as in the second-order geometric progression of Chapter 3).

The behavior of Shellsort is strikingly different for first-order
geometric progressions as opposed to higher order ones. In the former
case, as remarked in Section 2.1, Shellsort takes time $O(n^{3/2})$ using
perturbed progressions, but time $O(n^2)$ using an unperturbed sequence
of, say, powers of two. The theorems and remarks of Chapter 3 depend
for their proof on the higher-order sequences remaining unperturbed.

The questions answered in Chapters 2 and 3 are of academic interest
only, since there already exist sorting techniques which, on theoretical
grounds alone, are as good as Shellsort, and which on empirical evidence
are much better for almost all applications. Chapter 4 gives a most
interesting exception to this rule, in that we show that in practice
Shellsort is the best method to use for sorting networks, at least
from the point of view of speed. This is not to say that Shellsort
will always be better than Batcher's method; a way of building
considerably faster comparators, which does not apply to our median-finders,
could upset this claim. But the arguments presented in Chapter 4 seem to
indicate that a different technology would be required for this to happen.

Bibliography

Batcher, K. E., 1968. "Sorting networks and their applications." Proc. AFIPS SJCC, 32, 307-314.

Boerner, H., 1955. Darstellung von Gruppen. Springer-Verlag, Berlin.

Floyd, R. W. and D. E. Knuth, 1970. "The Bose-Nelson sorting problem." Computer Science Department Report STAN-CS-70-177, Stanford University.

Hibbard, T. N., 1963. "An empirical study of minimal storage sorting." Comm. ACM, 6, 5, 206-213.

Knuth, D. E., 1969. "An analysis of Shell's sorting algorithm." Unpublished paper. Results appear in [Knuth, 1972].

Knuth, D. F., 1972. The Art of Computer Programming, 3, Addison-Wesley, Reading, Massachusetts.

Lazarus, R. B. and R. M. Frank, 1960. "A high-speed sorting procedure." Comm. ACM., 3, 1, 20-22.

Papernov, A. A. and G. V. Stasevich, 1965. "A method of information sorting in computer memories." Problems of Information Transmission, 1, 3, 63-75.

Shell, D. L., 1959. "A high-speed sorting procedure." Comm. ACM., 2, 7, 30-32.

Van Voorhis, D. C., 1971. "Large [g,d] sorting networks." Technical Report TR-18, Digital Systems Laboratory, Stanford University.