The Design and Analysis of Algorithms For Sort and Merge using Comparisons

Mai Thanh

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

June 1983

ABSTRACT

THE DESIGN AND ANALYSIS OF ALGORITHMS
FOR SORT AND MERGE USING COMPARISONS

Mai Thanh

This thesis studies the Design and Analysis of algorithms for merging and sorting using comparisons. A Top-Down approach was developed which combines with heuristic approaches to derive algorithms for merging 4 elements with n other elements. A hybrid method was designed which significantly improves the upper bound of the number of comparisons in the worst case to sort t elements.

ACKNOWLEDGEMENTS

I wish to express my great gratitude and grateful thanks to my supervisor, Prof. T.B.Bui. His support is greatly appreciated throughout the years of both my Undergraduate and Graduate studies. His advices, knowledge and guidance have helped me to initiate, pursue and complete this thesis.

I am also grateful to Prof. V.S.Alagar for various advices and discussions which make the completion of this thesis possible.

Dr. Frank Hwang of the Bell Research Laboratories, Murray Hill, N.J. has contributed greatly through many private communications.

I wish to thank the Natural Science and Engineering Research Council (NSERC) for providing me a Postgraduate scholarship.

Last, but not least, I wish to express my special thanks to my parents, sisters and brothers for their moral support and encouragement during my studies.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION.

We are now in the computer era. The computer is one of the most versatile and powerful tools available for solving problems and for processing information. Its use has revolutionized and will further revolutionize many aspects of business, science and engineering and will effect the life of every human being. In some areas, computers can be used to solve problems that can not be solved by human beings. No one can negate the usefulness of computers. But the important thing is how should we use them to solve problems in an efficient way. The methods or procedures to solve problems or to process information are called algorithms. Efficient algorithms are important since they make use of the computers efficient.

Suppose we are given a problem. How do we find an efficient algorithm for its solution? Once we have found such an algorithm, how do we compare this algorithm with other existing ones that solve the same problem? How should we judge the goodness of an algorithm? Questions of this nature are of interest both to programmers as well as to

theoretical computer scientists. These questions have led to a new branch of computer science : analysis of algorithms.

Analysis of algorithms is a new area of research. It emerged as a new scientific subject during the sixties and has been quickly established as one of the most active fields of study. Today, analysis of algorithms is an important part of computer science. This field of research, as a discipline, relies heavily on both computer science and mathematics. It covers the design of efficient algorithms for solving problems and the examination of their complexities. By "efficient" we mean that an algorithm is designed in the best possible way and in many cases this means optimal. To examine the complexity of an algorithm usually means to consider space and time required by an algorithm for its execution. These aspects, space and time, are the most important considerations in algorithmic efficiency. Since both are limited, it is important to determine how much space and time an algorithm requires. The measurement of space and time complexities are based on a fundamental quantity : the size of the input data. Since all data can be encoded, we customarily associate the input with an integer, which is the measure of the input data. Mathematically speaking, space and time required to solve a problem are functions of n, the size of the input data. Our task is to determine these functions.

Sorting and merging problems are of interest in many

areas. Both problems are related to the determination of the linear order of a given list of input items. Sorting is the process by which a list of items or records, normally disordered , is put into order according to certain criteria based on the content of each record. Why is it important to have ordered lists rather than disordered ones? A simple example will suffice to illustrate this. Consider a list of a thousand records. Suppose we wish to find a particular record. Whether the list is ordered or not, we must look at 500 records on the average to find the record we want, if our search is conducted from the beginning to the end. Indeed, for an unordered list , there is no better procedure than a serial search. If the list does not contain our record, we must examine all the records in the list to find it out. But if the list is ordered, certain techniques will greatly reduce the searching effort, whether or not the record is present. For example, using binary search, the search for a record and the determination of its presence in or absence from an ordered list of 1000 records takes only 10 passes on the average. An ordered list provides the following three advantages :

- Faster search for a single record, using various sophisticated search techniques.
- Faster determination for the presence or absence of a record.
- More rapid retrieval of several records at once.

We assume that each record has a specific field, called

the sort-key (or simply key). Comparisons are made on these keys to determine the position of the corresponding record in the ordered list. Various kinds of sort algorithms have been considered : alphabetical sort such as sorting a dictionnary, numerical sort such as sorting a sequence of numbers, indexical sort such as sorting the call numbers of the library's books. It becomes clear now that we can associate t, the number of items to be sorted, as the input size of any algorithm for sorting t items. We denote an algorithm to sort t items by sort(t). As for merge problems, merging algorithms are a special case of sorting. This is the process which determines the linear ordering of the union of more than one, usually two, ordered lists. In the examination of an algorithm for merging , we can consider the size of all the lists to be merged as input size. An algorithm for merging m items with n items will be denoted by merge(m,n). Again, various merge algorithms have been designed : multiway merge algorithm, tape linear algorithm, polyphase merge algorithm, etc... In order to determine the linear order of a pair of items, comparisons have to be made on the sort-keys of these items. It is natural that one may ask the following question: how many comparisons a given algorithm requires to solve such a sort/merge problem? Does there exist another algorithm which requires fewer comparisons? What is the smallest number of comparisons required to solve a given sort/merge problem? Except for some special cases, these questions

have remained unanswered for a long time.

It is the purpose of this thesis to design efficient algorithms for some sort/merge problems, and to examine their complexities. An algorithm to merge 4 elements with n other elements will be given in the worst case using a Top-Down approach together with heuristic approaches. The average case analysis will be carried out for the worst-case optimal algorithms for merging (2,n) and (3,n).

In sorting, we have developed a hybrid method which improves significantly the well-known Ford-Johnson algorithm(fja) for sorting. Our algorithm is better than the two existing algorithms which surpass the fja. We believe that our algorithm is optimal among the class of all hybrid methods.

In chapter 2, we discuss some fundamental issues of data structures for input/output purposes. We also describe the information theoretic lower bounds for sorting and merging problems. Since this is essentially related to binary trees, definitions and properties of binary trees will also be given. Lastly, the binary insertion is described since it is the only merge algorithm which achieves the theoretic lower bound for any n. It is used also in other merging algorithms. The concept of optimality of an algorithm will be described for use in the subsequent chapters.

Chapter 3 describes the merge algorithms in the general case, i.e. the merge(m,n) algorithms where $m \geq 0$, $n \geq 0$. We

introduce the splitting method in this chapter. It is a powerful method used to improve the lower bounds of the optimal merge problem in the worst case. Next, we describe various known algorithms : the Tape-Linear algorithm, the Binary Hwang-Lin algorithm, the Fractile Insertion algorithm, and the Forward-Testing Backward-Insertion algorithm. We present a new algorithm which improves the Binary Hwang-Lin algorithm. The analysis in the worst case and the average case will be discussed together with their description.

In chapter 4, we consider the case of merging (m,n) where m is small, namely $1 \leqslant m \leqslant 5$. The merge algorithms (2,n) and (3,m) are briefly rewieved. As for merging(4,n), we develop a Top-Down approach to solve the optimal merge(4,n) problem. This approach is a new method to find the worst-case bound for the merge(m,n) problem. It is used together with heuristic approaches to solve the merge problems. Also, the merge(5,n) bounds are given. The average-analysis will be carried out for the merge (2,n) and (3,n) problems.

Chapter 5 considers the worst-case analysis of sorting algorithms. A hybrid method using merging algorithms in an efficient way for sorting n items will be described in this chapter. This hybrid method makes use of efficient merging algorithms to save comparisons used as much as possible. It is also shown that the Ford-Johnson algorithm is not optimal for a "big" subset of the set of natural integers.

CHAPTER 2

DATA STRUCTURES AND THEORETIC BOUNDS

In this chapter we study the data structures used to store the input/output data for a sort/merge algorithm using comparisons. Since binary trees are intimately related to this problem, the definition and properties of binary trees are also considered. The theoretical lower bounds for the sort/merge problems are derived. Lastly, the binary insertion algorithm is examined since this algrithm achieves the theoretic lower bounds and it is widely used.

2.1 Data structures.

Sort/Merge algorithms are methods to determine the linear ordering of a given set. In the case of sorting, no information about this set is knouwn in advance. In the case of merging, this set is known as the union of two subsets which are totally ordered. In sort/merge algorithms, it is convenient and useful to store input items into arrays or lists (or equivalent). In the case of sorting t items, we store these items into a list A(1:t). In the case of merging m items with n items, we store these items into two lists A(1:m) and B(1:n), respectively. Using list structures to store input data, we get some

conveniences : it is easy to decide which items to be compared by indicating their subscripts. Data movement, data exchange are also easily performed. The complexities of sort/merge algorithms can be easily expressed as functions of the input size. The only assumption to be made is that the memory of the computer is large enough to store the whole input data. The numbers of comparisons required are functions of t and (m,n), respectively. The objective is to design algorithms such that these functions are minimized.

Since pairwise comparisons are the main operations to determine the linear ordering of the input. Each comparison is made on a pair of items, say x and y. There are 3 possible outcomes : $x < y$, $x = y$, and $x > y$. We assume that all input data are totally different hence the outcome $x = y$ is not possible. If we represent a pairwise comparison by a node, then the outcome $x < y$ is represented by the left branch of that node and the outcome $x > y$ by the right branch. Therefore, the entire comparison strategy can be represented by a binary tree with the root represents the first comparison made. Each internal node represents a pairwise comparison made on two items, and the leaf-nodes are the outcomes that the list are totally ordered. Hence, a sort/merge algorithm using comparisons can always be represented by a binary tree.

2.2 Binary tree.

Definition :

A binary tree is a finite set of elements called nodes, which is either empty or satisfying the following :

a) There is a distinguished node called root.

b) The remaining nodes are divided into two disjoint subsets, called left subtree and right subtree, each of which is a binary tree.

Figure 2.1 gives some examples of a binary tree :

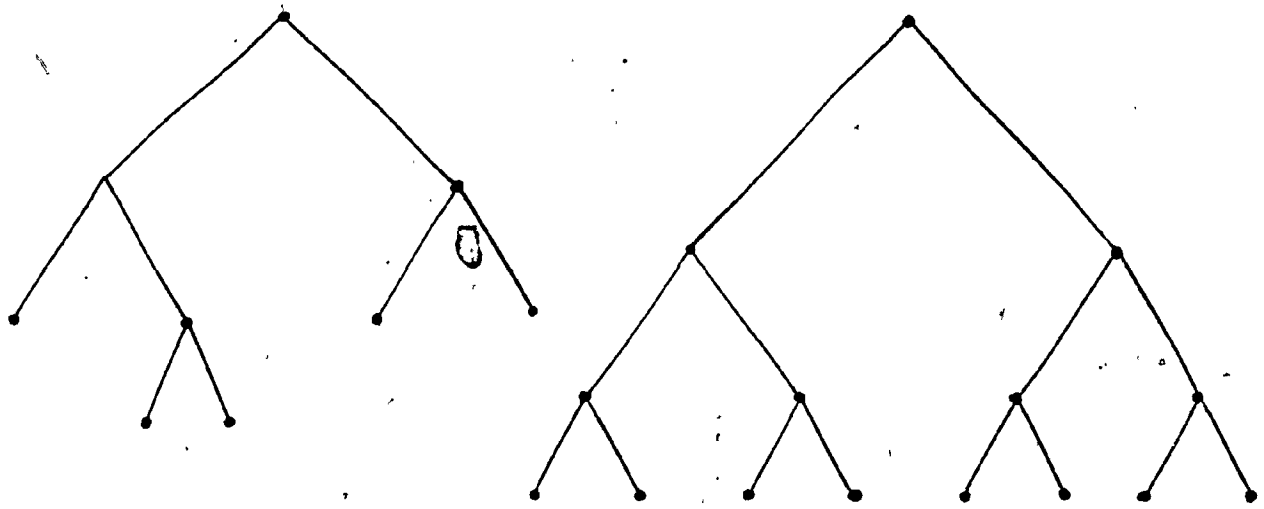


Figure 2.1

If a node w is the root of the left(right) subtree of a node v, w is called the left(right) child of v, and v is called the parent of w.

The degree of a node is the number of nonempty subtree it has.

A node with degree zero is called a leaf node.

A node with a positive degree is called internal node.

The level of the root is zero, and the level of any other node is one plus the level of its parent.

The depth of a binary tree is the maximum of the levels of its leaves.

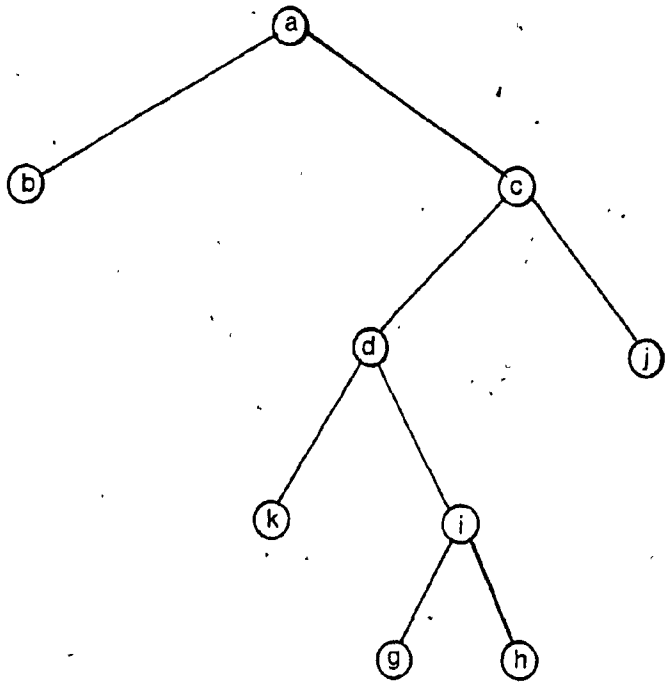


Figure 2.2

In figure 2.2, a is the root of the tree, and it is the parent of nodes b and c. d is the left child of c, j is the right child of c. a,c,d,i are of degree 2 and therefore are internal nodes. b,k,g,h,j are of degree 0 and therefore are leaves.

Level of a is 0, level of b,c is 1 and level of d,j is 2 and so on.

The depth of the tree is 4, which is actually the level

of g and h.

The following properties of binary tree will be used throughout the Thesis. We refer to [08,24] for proofs and further details of these properties.

Proposition 2.1

Let ℓ be the number of leaves in the binary tree and let d be its depth then

(2.1) $$\ell \leq 2^d.$$

Proposition 2.2.

Let ℓ, d be as in Proposition 2.1, then

(2.2) $$d \geq \lceil \log \ell \rceil *.$$

Proposition 2.3.

Among all binary trees with ℓ leaves, the epl is minimized if all leaves are on at most two adjacent levels.

Proposition 2.4.

The minimum epl for a binary tree with leaves is

(2.3) $$\lfloor \log \ell \rfloor + 2(\ell - 2\lfloor \log \ell \rfloor).$$

or

(2.4) $$(d-1) + 2(\ell - 2^{d-1}).$$

---

* Note : All log in this thesis are to the base 2 unless it is indicated otherwise.

2.3 The Information Theoretical lower bounds.

Most discussions on the performance of sort/merge algorithms assume that the input data is randomly distributed. This means that given a range of possible key values, the possibility that a key has any value in this range is the same. A permutation of a particular ordering of the number is as probable as any other. There are t! possible outcomes for sorting a list of t distinct items, and $\binom{m+n}{m} = \frac{(m+n)!}{m!n!}$ possible outcomes for merging m distinct items with n other distinct items.

Any algorithm for sorting t items has t! possible outcomes which are all equally likely. Hence, the corresponding outcomes are all equally likely. Hence, the corresponding tree (binary tree) has at least t! leaves, each corresponding to one possible outcome of the sorting algorithm.

Theorem 2.5.

Any algorithm to sort t items by comparisons of keys must do at least $\lceil \log t! \rceil$ comparisons and $\lfloor \log t! \rfloor + 2 - \frac{2^{\lfloor \log t! \rfloor + 1}}{t!}$ comparisons in the worst case and in the average case, respectively.

Proof :

These are the results of Lemmas 2.2 and 2.4. Since the number of comparisons required in the worst case of an algorithm is indeed the depth of the corresponding binary tree and the average number of comparisons is the external

path length divided by the number of outcomes which is t!.

Q.E.D.

These bounds are called the information theoretical lower bounds for sorting algorithms. We denote these bounds for the worst case and the average case, respectively, by :

$$(2.5) \qquad ITS(t) = \lceil \log t! \rceil$$

$$(2.6) \qquad \overline{ITS}(t) = \lfloor \log t! \rfloor + 2 - \frac{2^{\lfloor \log t! \rfloor + 1}}{t!}$$

If we use the Stirling approximation for t!, i.e. $t! \simeq \sqrt{2\pi t}(t/e)^t$, then (2.5) becomes

$$ITS(t) \simeq t\log t - (1/\ln 2)t + O(1)$$

(ln denotes natural logarithm)

$$(2.7) \qquad \simeq t\log t - 1.4427t + O(1).$$

Similarly, we have the information theoretic lower bounds for merging algorithms.

Theorem 2.6

Any algorithm to merge m items with n other items by comparisons of keys must do at least $\lceil \log \binom{m+n}{n} \rceil$ comparisons and $\lfloor \log \binom{m+n}{m} \rfloor + 2 - \frac{2^{\lfloor \log \binom{m+n}{m} \rfloor + 1}}{\binom{m+n}{m}}$ comparisons in the worst case and in the average case, repectively.

Proof :

Similar to that of theorem 2.5.

Q.E.D.

We denote these information theoretical lower bounds by

$$\underline{ITM}(m,n) = \lceil \log \binom{m+n}{n} \rceil \tag{2.8}$$

$$\overline{ITM}(m,n) = \lfloor \log \binom{m+n}{n} \rfloor + 2 - \frac{2^{\lfloor \log \binom{m+n}{n} \rfloor + 1}}{\binom{m+n}{n}} \tag{2.9}$$

By theorems 2.5 and 2.6, we have the information theoretic lower bounds for a sort/merge algorithm. Some typical questions one may ask are :

i) Does there exist an algorithm which achieves such a bound?

ii) If no algorithm achieves such a bound, what is the lowest bound which can be achieved?

The concept of optimality of an algorithm addresses itself to these questions.

An optimal bound for a sort/merge problem is a bound which satisfies the following criteria :

i) There exist some algorithms which achieve this bound.

ii) Any other algorithm has to use at least the same number of comparisons dictated by the optimal bound.

We have two bounds corresponding to the two cases : the worst case and the average case. In the problem of sorting, the optimal bounds are denoted by $S(t)$, and $\overline{S}(t)$, for the worst case and the average case, respectively. In the problem of merging, the optimal bounds are denoted by $M(m,n)$, $\overline{M}(m,n)$ with similar definitions.

Of course, we have the following :

$$S(t) \geqslant ITS(t)$$

$$\overline{S}(t) \geqslant \overline{ITS}(t)$$

$$M(m,n) \geqslant ITM(m,n)$$

$$\overline{M}(m,n) \geqslant \overline{ITM}(m,n) .$$

2.4 Binary Insertion

The simplest form of the merge problem is the (1,n) problem, that is, insert one element into an ordered list B(1:n) with the assumption that the inserted item is distinct from those of list B. The best and well-known algorithm for this problem is the binary insertion which can be described as follows:

Algorithm binary insertion(bi) :

Step 1 : $\ell \leftarrow 1$; $u \leftarrow n$.

Step 2 : If $\ell+1 < u$ then stop ; output .

Step 3 : $j \leftarrow \left\lceil \frac{\ell+u}{2} \right\rceil$

Step 4 : Compare $(A^{1}:B^{j})$

If $A_1 < B_j$ then $u \leftarrow j-1$;

else $\ell \leftarrow j+1$; go to step 2.

In the format of a merge tree, we can represent this algorithm by :

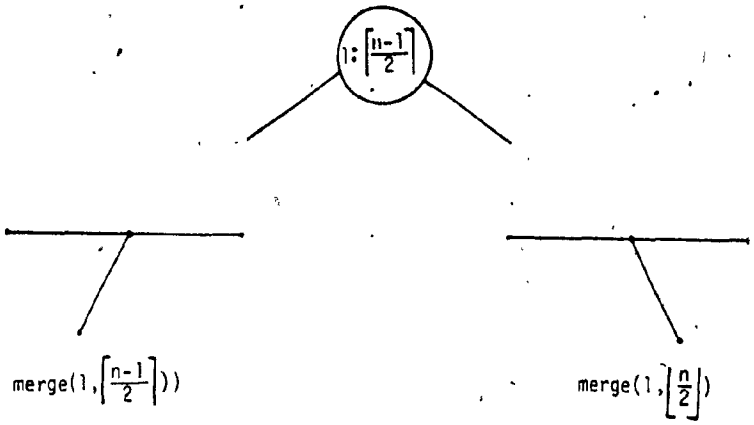


Figure 2.3

Theorem 2.7.

In the wosrt case, the number of comparisons required by the binary insertion is :

$$M_{bi}(1,n) = \lfloor \log\ n \rfloor + 1$$

and in the average case is

$$\bar{M}_{bi}(1,n) = \lfloor \log(n+1) \rfloor + 2 - \frac{2^{\lfloor \log(n+1) \rfloor + 1}}{n+1}$$

Proof :

From figure 2.5, one can easily get the equations :

$$M_{bi}(1,n) = 1 + \max \{M_{bi}(1,\lceil \tfrac{n-1}{2} \rceil), M_{bi}(1,\lceil \tfrac{n}{2} \rceil)\}$$

$$= 1 + M_{bi}(1,\lceil \tfrac{n}{2} \rceil)$$

$$= 1 + 1 + M_{bi}(1,\lceil \tfrac{n}{2} \rceil)$$

$$= 1 + 1 + \ldots + M_{bi}(1, \lceil \frac{n}{2} \rceil)$$

$$= \lfloor \log n \rfloor + M(1,1)$$

$$= \lfloor \log n \rfloor + 1.$$

Hence, we get

$$\bar{M}_{bi}(1,n) = 1 + \lfloor \log n \rfloor$$

$$= \lceil \log(n+1) \rceil$$

$$= \overline{ITM}(1,n).$$

As for the average case, we prove that the binary insertion tree has its leaves on the last two levels. The proof is by induction on n. If $n = 2^k - 1$, then the number of outcomes is 2^k. Using the comparison $(1:2^{k-1})$, we have the two subproblems merge$(1, 2^{k-1}-1)$. Then by induction hypothesis we have all the leaves on the last level. If n+1 is not a power of 2, then the t subtrees have their leaves on the last two levels d and d-1. Hence, the tree has its leaves on the last two levels. By Lemma 2.3 the tree has minimum epl.

Therefore,

$$\bar{M}_{bi}(1,n) = \overline{ITM}(1,n)$$

$$= \lfloor \log(n+1) \rfloor + 2 - \frac{2^{\lfloor \log(n+1) \rfloor + 1}}{n+1}$$

Q.E.D.

Both the wosrt-case and average-case information theoretical lower bounds are achieved. Thus, the binary insertion is both optimal in the worst case and in the average case. A merging algorithm is called M-optimal if it

is optimal in the worst case. Similarly, a merging algorithm is $\bar{M}$-optimal if it is optimal in the average case.

One interesting question that could be raised here is the following : In how many different ways we can insert one element into a list B(1:n) to achieve the M-optimal bound (or the M-optimal bound). That is how many different trees are there which achieve the M-optimal bound (or the $\bar{M}$-optimal bound) ?

Theorem 2.8.

An M-optimal insertion algorithm is also $\bar{M}$-optimal.

Proof :

If an M-optimal algorithm tree achieves the information theoretical lower bound, then it achieves the minimum depth. Therefore, it is also M-optimal.

Q.E.D.

Theorem 2.9

When inserting an element A_1 into a list B(1:n), the M-optimal algorithm must compare the element A_1 with the element B_x of list B, where x must satisfy the condition :

$$(2.10) \qquad n - 2^{\lfloor \log n \rfloor} + 1 \leqslant x \leqslant 2^{\lfloor \log n \rfloor} .$$

and similarly for the $\bar{M}$-optimal algorithm : $(A_1:B_y)$ where y must satisfy :

$$(2.11) \quad \max\{2^{\lfloor \log n \rfloor - 1}, n-2^{\lfloor \log n \rfloor}+1\} \leqslant y \leqslant \min\{2^{\lfloor \log n \rfloor}, n-2^{\lfloor \log n \rfloor - 1}+1\}$$

Proof :

The M-optimal tree has the depth $1+\lfloor \log n \rfloor$ then the left and the right subtrees can not have the depth exceeding $\lfloor \log n \rfloor$. Therefore, if we compare $(\bar{A}_1 : Bx)$ (see figure 2.4),

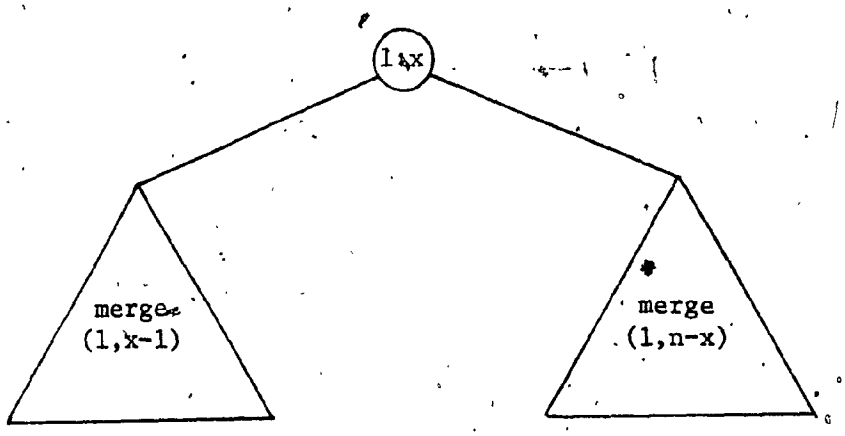


Figure 2.4

then we get

$\lceil \log x \rceil \leqslant \lfloor \log n \rfloor$ implies $x \leqslant 2^{\lfloor \log n \rfloor}$,

and

$\lceil \log (n-x+1) \rceil \leqslant \lfloor \log n \rfloor$ implies $n-x+1 \leqslant 2^{\lfloor \log n \rfloor}$.

So we have $x \geqslant n - 2^{\lfloor \log n \rfloor} + 1$.

In the $\bar{M}$-optimal tree, the leaves must appear on the last two levels, then the number of outcomes of the left subtree must satisfy the following :

Let y be the number of outcomes in the left subtree, then n-y+1 is the number of outcomes in the right subtree.

Hence, we must have :

$$2^{\lfloor \log n \rfloor - 1} \leqslant y \leqslant 2^{\lfloor \log n \rfloor}$$

$$2^{\lfloor \log n \rfloor - 1} \leqslant n-y+1 \leqslant 2^{\lfloor \log n \rfloor}.$$

Combine these inequalities, y must satisfy (2.11).

Now, we denote

T(n) to be the number of different insertion(1,n) trees which are M-optimal, and

$\bar{T}(n)$ to be the number of differnent insertion(1,n) trees which are M-optimal.

By theorem 2.8, we have

$$\bar{T}(n) \leqslant T(n)$$

and by theorem 2.9, $\bar{T}(n)$ and T(n) satisfy the recurrence formulas :

$$(2.12) \qquad T(n) = \sum_{x=m_1}^{m_1'} T(x-1) \cdot T(n-x)$$

where $m_1 = n - 2^{\lfloor \log n \rfloor} + 1$,

$m_1' = 2^{\lfloor \log n \rfloor}$,

and :

$$(2.13) \qquad \bar{T}(n) = \sum_{x=m_0}^{m_0'} \bar{T}(x-1) \cdot \bar{T}(n-x)$$

where $m_0 = \max\{2^{\lfloor \log n \rfloor - 1}, n - 2^{\lfloor \log n \rfloor} + 1\}$

$m_0' = \min\{2^{\lfloor \log n \rfloor}, n - 2^{\lfloor \log n \rfloor - 1} + 1\}$

and the initial values :

$T(1) = \bar{T}(1) = 1.$

A partial solution for (2.12) and (2.13) is giiven below :

$T(2^k - 1) = 1$; $\qquad \bar{T}(2^{k-1} - 1) = 1$;

$T(2^k - 2) = 2^{k-1}$; $\qquad \bar{T}(2^k) = 2^k$;

For more complete solutions, see appendix A.

Theorem 2.8 was proved for the binary insertion algorithm. The merge(1,n) is the only problem which achieves the theoretic lower bounds in both cases (average case as well wosrt case) for any $n \in N$. For $m=2,3,\ldots$ these theoretical bounds could not be achieved for any value of n. however, it is also conjectured that theorem 2.8 is also true for $m \geq 2$.

CHAPTER 3

MERGE ALGORITHMS : CASE m LARGE

3.1 Definition.

Merge algorithm is the process of determining the linear ordering of two ordered lists.

Suppose we are given two linear ordered lists A and B :

$$A_1 < A_2 < \ldots < A_m,$$

and

$$B_1 < B_2 < \ldots < B_n.$$

We want to merge them by means of a sequence of pairwise comparisons between an element of A and an element of B. Since the merge process is symmetric in the sense that merging A with B is the same as merging B with A, we can assume without loosing the generality that $m \leq n$. Also, we assume that the m+n elements are totally distinct.

Given an algorithm s for merging(m,n), we define :

$M_s(m,n)$ = maximum number of comparisons by algorithm s for all possible ordering of A U B.

$\bar{M}_s(m,n)$ = average number of comparisons required by algorithm s.

Using this notation, the optimal bounds defined in chapter 2 can be rewritten as :

$$M(m,n) = \min M_S(m,n)$$

$$\bar{M}(m,n) = \min \bar{M}_S(m,n)$$

where the min is taken over all algorithms.

The two functions $M(m,n)$, $\bar{M}(m,n)$ are not known in general. An algorithm s is called M-optimal if

$$M_S(m,n) = M(m,n)$$

Similarly, an algorithm u is called M-optimal if

$$\bar{M}_U(m,n) = \bar{M}(m,n)$$

From chapter 2, we know that $M(m,n) \geq ITM(m,n) = \lceil \log \binom{m+n}{m} \rceil$. Since $M(m,n)$ is not known in general, it is desirable to derive another lower bound for the $M(m,n)$ rather than the $ITM(m,n)$ bound. In the next section, we will discuss some methods for deriving "better" lower bounds.

3.2 Splitting methods

The M-optimal merging problem may be viewed as a two-person game with perfect information in which the first player chooses the comparisons, while the second player chooses (consistently) the results of these comparisons. The first player tries to choose the comparisons to minimize the total number of comparisons, while the second player tries to choose the outcome of each comparison to maximize the total number of comparisons required. The M-optimal lower bounds $M(m,n)$ are just the min-max value of the game.

Whereas a given strategy of the first player (i.e. an algorithm) determines an upper bound on optimal merging, a given counterstrategy of the second player determines a lower bound on optimal merging. The terminologies strategy and counterstrategy are self-explanatory since the first player tries to meerge(m,n) in as few comparisons as possible, while the second player tries to delay the achievement of this task as long as possible.

Consider the merge problem(m,n). Suppose the first comparison in the merging process is $(A_i:B_j)$ (see figure 3.1)

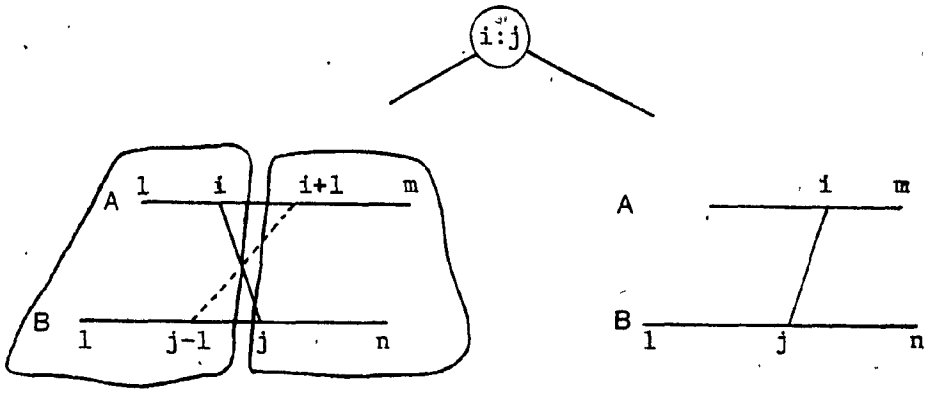


Figure 3.1

A possible splitting after the choice of the answer $A_i < B_j$ is to divide the problem into two subproblems :

- merge $A_1,\ldots,A_i$ togetther with $B_1,\ldots,B_{j-1}$, this is the problem of merge(i,j-1).
- merge $A_{i+1},\ldots,A_m$ together with $B_j,\ldots,B_n$, this is the problem of merge(m-i,n-j+1).

In this case we say that the problem of merge(m,n) is

splitted between the pairs of elements (A_i,A_{i+1}) and (B_{j-1},B_j). This kind of splitting is known as the ordinary splitting method (or simple splitting strategy), since it is straight-forward after the comparison $(A_i:B_j)$.

In [06], Christen used this ordinary splitting method together with the divide-and-conquer approach to obtain the recurrence condition :

$$M(m,n) \geqslant \min_{\substack{1\leqslant i\leqslant m \\ 1\leqslant j\leqslant n}} \max_{(h,k)\in P} \{(1+M(h,k)+M(m-h,n-k))\}$$

where

$$P = \{(h,k) \mid (m\geqslant h\geqslant i \text{ and } o\leqslant k<j) \text{ or } (o\leqslant h<i \text{ and } j\leqslant k\leqslant n)\}$$

However, in some cases the ordinary method is not helpful, since it derives the lower bound which is worse than the information-theoretic lower bound.

A more refined splitting method was suggested by Knuth [16]: Splitting at an element instead of between two elements. When splitting at an element B_k, (respectively at A_h), the counterstrategy must later respect the condition $A_h<B_k$ and $B_k<A_{h+1}$ (respectively $B_k<A_h$ and $A_h<B_{k+1}$), in order to insure the independence of the resulting problems. This requires the consideration of diverse variants of the merging game, in which the second player is restricted by the condition on extreme elements of the sequences. After the comparison $(A_i:B_j)$, consider the outcome $(A_i:B_j)$ shown on figure 3.2.

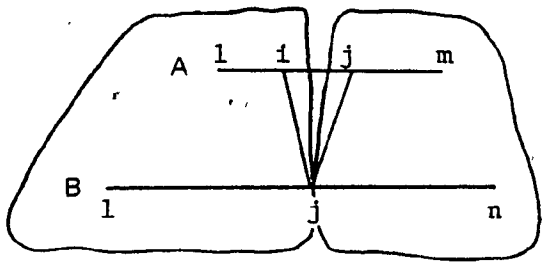


Figure 3.2

Thus, we have the relation :

$$M(m,n) \geq 1 + M\backslash(i,j) + /M(m-i,n-j+1).$$

The case $A_i < B_j < A_{j+1}$ is considered, where the notations /M and \M are defined as follows :

/Merge(m,n) = problem of merge(m,n) with $B_1 < A_1$ but this information is not known to the merger.

Merge\(m,n) = similar definition but with $B_n > A_m$.

/Merge\(m,n) = similar with $B_1 < A_1$ and $B_n > A_m$.

and from these definitions, we denote :

/M(m,n) = maximum number of comparisons to solve the /Merge(m,n) problem.

Similar notations for M (m,n) and /M\(m,n).

This method is known as oracular splitting method (or the commplex splitting strategy). Using the divide-and-conquer approach again, for the /Merge\ problem(m,n), Christen obtained the recurrence condition in this case which is given below :

$$/M\backslash(m,n) \geqslant \min_{\substack{1\leqslant i\leqslant m \\ 1\leqslant j\leqslant n}} \max_{(h,k)\in R} \left(1 + /M\backslash(h,k) + /M\backslash(m-h,n-k+1) \right)$$

where

$$R = \{(h,k) \mid (i\leqslant h\leqslant m \text{ and } 1<k<j) \text{ or } (o\leqslant h<i \text{ and } j<k<n)\}.$$

The usefulness of these splitting methods is twofold. First, it is used to prove the optimality of a given algorithm. Second, it is used to derive some lower bounds which are higher than the information-theoretical lower bounds. We discus below some interesting results using splitting methods.

Lemma 3.1 [25].

i) $/M\backslash(m,n) \leqslant /M(m,n) \leqslant M(m,n)$.

ii) $/M(m+1,n+1) \geqslant /M(m,n) + 2$.

iii) $/M\backslash(m+1,n+1) \geqslant /M\backslash(m,n) + 2$.

Proof :

i) Obviously since the /Merge\(m,n) problem is more constrained than the /Merge(m,n) problem which is again more constrained than the Merge(m,n) problem.

ii) The oracle for /M(m+1,n+1) is the problem where $A_1<B_1$. consider the outcome : $A_1<B_1<A_2<B_2$. This forces $A_2,A_3,\ldots,A_{m+1}$ to be merged with $B_2,B_3,\ldots,B_{n+1}$ and the two comparisons $(B_1:A_1)$, $(B_1:A_2)$ cannot be avoided.

iii) Similar proof with the same outcome.

Q.E.D.

Theorem 3.2 [25].

(3.1) a) $M(m,m+d) \geq 2m + d - 1$ for $m \geq 2d-2$;

b) $M(m,m+d) \geq 2m + d - 1$ for $m \geq 2d-1$;

c) $M(m,m+d+2) \geq 2m + d$ for $m \geq 2d - 1$.

Proof :

If b) and c) are true for $m=2d-1$, then they are also true for $m > 2d-1$ by applying the Lemma 3.1(i) and (ii). Also, if b) is true for $m \geq 2d-1$ then the Lemma 3.1(i) implies that a) is also true for $m \geq 2d-1$. Thus, it is sufficient to prove a) for $m = 2d-2$, b) and c) for $m = 2d-1$, that is,

$$M(2d-2,3d-2) \geq 5d-5,$$

$$M(2d-1,3d-1) \geq 5d-3,$$

and $$M(2d-1,3d+1) \geq 5d-2.$$

The proof is by induction on d. The starting values for $1 \leq d < 3$ are given in [16].

Part a)

Suppose an algorithm begins by comparison $(A_i:B_j)$ where $i=2k-1$, $j \leq 3k-2$ for some $1 \leq k < d$. Consider the case $A_i > B_j$ and use the ordinary splitting method, we get the outcome as shown on figure 3.3.

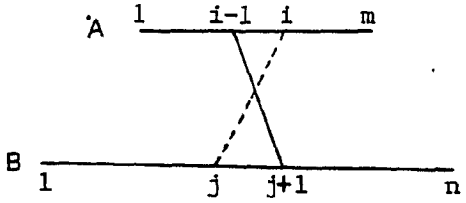


Figure 3.3

$$M(2d-2,3d-2) \geq 1 + M(2k-2,3k-2) + M(2d-2k,3d-3k)$$
$$\geq 1 + (5k-5) + (5(d-k)-1)$$
$$= 5d-5.$$

If $i=2k-1$ and $j \geq 3k-1$, the oracular strategy announces that $A_i < B_j$ (see figure 3.4) and uses B_{3k-2} in both subproblems. This leads to

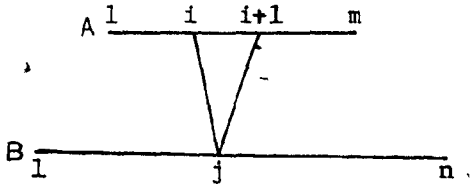


Figure 3.4

$$M(2d-2,3d-2) \geq 1+M(2k-1,3k-2) + M(2(d-k)-1,3(d-k)+1)$$
$$\geq 1+M(2k-1,3k-2)+M(2(d-k)-1,3(d-k)+1)$$
$$\geq 1+(5k-4)+(5(d-k)-2)$$
$$= 5d-5.$$

This settles the case where i is odd. Reversing the order of the elements in A and B maps all points of A with even subscripts onto those with odd. Thus by symmetry we

have handled the even case as well.

Part b).

Suppose the first comparison of an algorithm is $(A_i : B_j)$ with $i=2k-1$ and $j \leqslant 3k-2$, where $1 \leqslant k \leqslant d$. Consider the case $A_i > B_j$. We have by the oracular splitting method, with B_{3k-1} in both subproblems :

$$\begin{aligned} /M(2d-1,3d-1) &\geqslant 1 + /M(2k-2,3k-1) + /M(2(d-k)+1,3(d-k)+1) \\ &\geqslant 1 + (5k-5) + (5(d-k)+1) \\ &= 5d-3. \end{aligned}$$

If $i=2k-1$ and $j \geqslant 3k-1$, the case $A_i < B_j$ is considered. The ordinary strategy yields

$$\begin{aligned} /M(2d-1,3d-1) &\geqslant 1 + /M(2k+1,3k-2) + M(2(d-k),3(d-k)+1) \\ &\geqslant 1 + (5k-4) + 5(d-k) \\ &= 5d-3. \end{aligned}$$

Now, suppose $i=2k$ and $j \leqslant 3k$, with $1 \leqslant k < d$. Choosing $A_i > B_j$, the oracular strategy leads to

$$\begin{aligned} /M(2d-1,3d-1) &\geqslant 1 + /M(2k-1,3k+1) + /M(2(d-k),3(d-k)-1) \\ &\geqslant 1 + (5k-2) + (5(d-k)-2) \\ &= 5d-3. \end{aligned}$$

Otherwise, if $i=2k$ and $j \geqslant 3k+1$, the simple strategy with $A_i < B_j$ produces

$$\begin{aligned} /M(2d-1,3d-1) &\geqslant 1 + /M(2k,3k) + M(2(d-k)-1,3(d-k)-1) \\ &\geqslant 1 + (5k-1) + (5(d-k)-3) \\ &= 5d-3. \end{aligned}$$

Part c).

Assume an algorithm begins with comparison $(A_i:B_j)$ with $i=2k-1$ and $j \leqslant 3k-1$, where $1 \leqslant k < d$. Choosing $A_i < B_j$, the simple strategy yields

$$\begin{aligned} /M\backslash(2d-1,3d+1) &\geqslant 1+/M(2k-2,3k-1)+M\backslash(2(d-k)+1,3(d-k)+2) \\ &\geqslant 1+/M\backslash(2k-2,3k-1)+/M(2(d-k)+1,3(d-k)+2) \\ &\geqslant 1+(5k-5)+(5(d-k)+2) \\ &= 5d-2. \end{aligned}$$

The case $i=2k-1$ and $j \geqslant 3k$ is the mirror image of this case.

If $i=2k$ and $j<3k+1$, with $1<k<d$, the ordinary strategy works again. Choosing $A_i < B_j$, we have

$$\begin{aligned} /M\backslash(2d-1,3d+1) &\geqslant 1 +/M(2k-1,3k+1)+M\backslash(2(d-k),3(d-k)) \\ &\geqslant 1 +/M\backslash(2k-1,3k+1)+/M(2(d-k),3(d-k)) \\ &\geqslant 1+(5k-2)+(5(d-k)-1) \\ &= 5d-2. \end{aligned}$$

Finally, the case $i=2k$ and $j \geqslant 3k+1$ is the mirror image of this case.

Q.E.D.

Using the same techniques of proving, Christen derived the following results:

Theorem 3.3 [06].

i) $/M\backslash(m,m+1+2d) \geqslant 2m+d \quad (0 \leqslant d < 2m)$

and ii) $/M\backslash(m,5m+3) \geqslant 4m$.

Proof :

The proof is by induction on m and on d.

Obviously, the recurrence condition insures the monocity of /M\. Thus, for fixed positive m, the assertion (i) and (ii) entail :

(iii) $/M\backslash(m,m+3+2d) \geq 2m+d; \quad 0 \leq d \leq 2m.$

This auxiliary assertion is used to unify the argumentation.

For m=0, assertion (i) void, whereas assertions (ii) and (iii) are trivial. For m>0, assertion (i) is proved by induction on d, using further (i) and (iii) for smaller values of m. Assertion (ii) is then proved, using (ii) for smaller values of m, and (i) up to m.

Proof of (i) :

For d=0, the assertion holds, because (as note by Graham and Karp[15]) every merging method requires 2m comparisons in the interwoven case

$$B_1<A_1<B_2<\ldots<B_m<A_m<B_{m+1}.$$

So suppose 0<d<2m, and let the counterstrategy reply $B_j<A_i$, when $j \leq i-1+2\min(d,2i-1)$, and $A_i<B_j$ otherwise.

In the first case, let h=i-1 and k=i+2min(d,2i-1). the counterstrategy forces then

$1+/M\backslash(i-1,i+2\min(d,2i-1))+/M\backslash(m-i-1,m-i+2+2\max(0,d-2i+1))$

comparions. By the induction hypothesis for (iii), the second summand is at least 2i-3+min(h,2i-1). By the induction hypothesis for (i), because d<2m, the third summand is at least 2m-2i+2+max(0,h-2i+1). Thus at least 2m+d comparisons are required in this case.

In the second case, let h=i and k=i-1+2min(d,2i-1).

Because $0<d, k>1$, the counterstrategy forces

$1 + M(i, i-1+2\min(d, 2i-1)) + M(m-i, m-i+3+2\max(0, d-2i+1))$

comparisons. By the induction hypothesis for (i), the second summand is at least $2i+\min(d, 2i-1)-1$. By the induction hypothesis for (ii), again because $d<2m$, the third summand is at least $2m-2i+\max(0, d-2i+1)$. In this case too, $2m+d$ comparisons are required; therefore (i) holds for this m.

Proof of (ii) :

Let the counterstrategy reply $B_j<A_i$ when $j \leq 5i-1$, and $A_i<B_j$ otherwise. In the first case, let $h=i-1$ and $k=5i$. The counter strategy requires then $1+M(i-1, 5i)+M(m-i+1, 5m-5i+4)$ comparisons. By the induction hypothesis on (i), the third summand is at least $4m-4i+3$. thus at least $4m$ comparisons are required in this case.

In the second case, let $h=i$ and $k=5i-1$. The counterstrategy requires then $1+M(i, 5i-1)+M(m-i, 5m-5i+5)$ comparisons. By the induction hypothesis on (i), the second summand is at least $4i-1$. By the induction hypothesis on (ii), the third summand is at least $4m-4i$. Thus again $4m$ comparisons at least are required. This completes the proof of (ii) and therefore that of the theorem.

Q.E.D.

Theorem 3.4 [06].

i) For $e,m \geq 0$, $0 < d \leq \lfloor m/2 \rfloor$:

(3.2) $M(m, m(3.2^{e+1}-1) - 2^{e+1}+1+2^{e+2}d) \geq (e+4)m+d-1.$

ii) For $e \geq 0$ and $p \geq 0$:

(3.3) $M(2p+1, (2p+1)(2^{e+3}-1)+2^{e+1}+1) \geq (2p+1)e+9p+4.$

iii) For $e \geq 0$, $p \geq 0$ and $0 \leq d \leq p$:

(3.4) $M(2p+1, (2p+1)(2^{e+3}-1)+2^{e+3}(d+1)+1) \geq (2p+1)e+9p+$

iv) For $e \geq 0$, $p \geq 1$ and $0 \leq d \leq p$:

(3.5) $M(2p, 2p(2^{e+3}-1)+2^{e+2}+2^{e+3}d+1) \geq 2pe+9p+d.$

Proof :

The proof is by triple induction first on the number m of elements of the shorter sequence, then on the exponent e and finally on the displacement d.

For e=d=0 and any m, use inequalities in Theorem 3.3 as induction basis. Set

$$f(d,e,i) = (3.2^{e+1}-1)i - 3.2^{e+2}+1+2^{e+2}\min(d, \lfloor (i-1)/2 \rfloor$$

$$g(d,e,2i) = (2^{e+3}-1)2i+1+2^{e+3}\min(i-1,d),$$

$$g(d,e,2i+1) = (2^{e+3}-1)(2i+1)+1-2^{e+2}+2^{e+3}\min(i,d).$$

To prove assertion (i), let the counterstrategy replies $A_i < B_j$, if $f(d,e,i) < j$ and $B_j < A_i$ otherwise. In the first case, put h=i and k=f(d,e,i); in the second case, put h=i-1 and k=f(d,e,i)+1. For three other assertions, use g instead of f.

The rest of the argument is similar to that used in the proof of Theorem 3.3 and is therefore omitted.

Q.E.D.

We have discussed above the splitting methods and

presented some results on the lower bound for the M-optimmal merge problem. In the next sections, we present various algorithms which produce an upper bound for M(m,n).

3.3 The tape linear merge algorithm

The commonly used algorithm for merging is the tape linear merge algorithm. Given two lists, A(1:m) and B(1:n), to be merged ($m \leq n$), this algorithm consists of comparing the two largest elements (initially A_m and B_n) and the larger of these is deleted from its list and placed on an output list. The process is repeated until one list is exhausted. The tape linear merge algorithm can be described as given below:

Algorithm Tape linear merging(tl) :

Step 1. If m=0 or n=0 then stop.

Step 2. Compare (A_m:B_n).

Step 3. If ($A_m > B_n$), set n=n-1; go to step 1.

Step 4. If ($A_m < B_n$), set m=m-1; go to step 1.

Another representation of the tape linear algorithm is given in the format of a binary tree on figure 3.5.

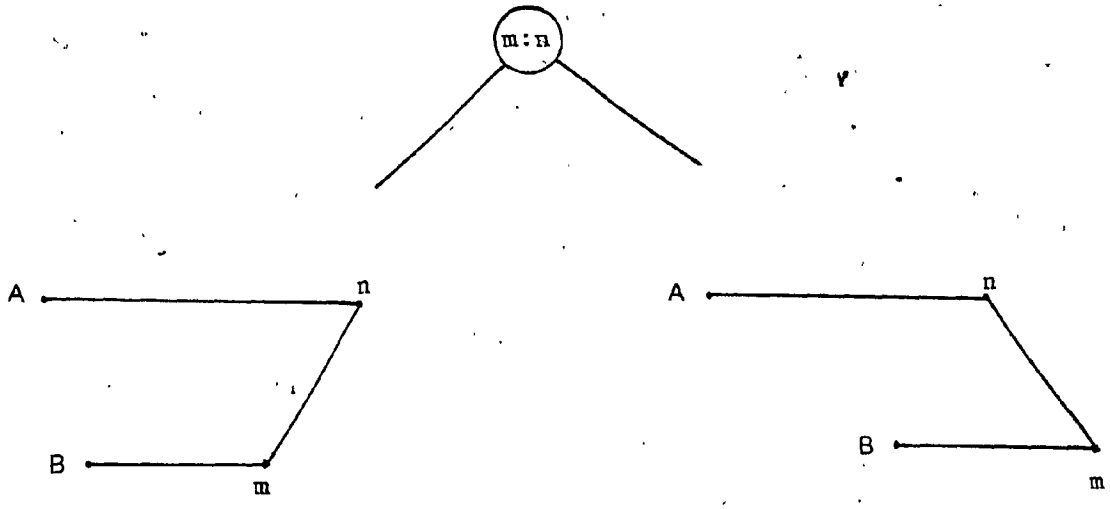


Figure 3.5

where the two branches represent the two subproblems of the linear merge (m-1,n) and (m,n-1). The maximum and average number of comparisons can be derived in Theorem 3.5.

Theorem 3.5.

$$M_{tl}(m,n) = m+n-1,$$

$$M_{tl}(m,n) = m+n - \frac{m}{n+1} - \frac{n}{m+1}$$

where tl denotes the tape linear merging algorithm.

Proof :

Each comparison in thee tape lineear merge algorithm can output only one element either from list A or list B, except for the last comparison : the smallest of one list is outputed together with the rest of the other list. Thus, for the wosrt data which involves the case $A_1<B_1<A_2$, the last comparison is $(A_1:B_1)$ and outputs 2 elements. So, we have :

(3.6) $$M_{tl}(m,n) = m+n-1.$$

As for the average analysis, from the binary tree shown on figure 3.5 we have the equation :

(3.7) $$\bar{M}_{tl}(m,n) = p(1+\bar{M}_{tl}(m,n-1)) + p'(1+\bar{M}_{tl}(m-1,n))$$

where

p = probability that $A_m < B_n$

$$= \frac{\binom{m+n-1}{n-1}}{\binom{m+n}{n}} = \frac{n}{m+n},$$

p' = probability that $A_m > B_n$

$$= 1 - p$$

$$= \frac{m}{m+n},$$

so (3.7) becomes :

(3.8) $$\bar{M}_{tl}(m,n) = 1 + \frac{m}{m+n}\bar{M}_{tl}(m-1,n) + \frac{n}{m+n}\bar{M}_{tl}(m,n-1).$$

By solving this recurrence equation[13], we have :

(3.9) $$M_{tl}(m,n) = m+n - \frac{m}{n+1} - \frac{n}{m+1}.$$

Q.E.D.

By theorem 3.2, we see that

$$M(m,n) = M_{tl}(m,n) \quad \text{when} \quad m \leq n \leq \left\lceil \frac{3m+1}{2} \right\rceil$$

That is, the tape linear merge algorithm is M-optimal in the range $[m,(3m+1)/2]$. It has been proved[12] that :

$$M(m,2m) \leq 3m-2 \quad \text{whereas} \quad M_{tl}(m,2m) = 3m-1.$$

Therefore, the tape linear algorithm is not M-optimal for $n=2m$. The exact value ℓ for which $M(m,\ell) = M_{tl}(m,\ell)$ has not been determined.

The distinctive feature of this algorithm is its simplicity and straightforward access. However, in general it is quite inefficient in the sense that it requires a lot more comparisons than the optimal bound. The difference $M_{tl}(m,n) - M(m,n)$ may be very large when n tends to infinity. In the next section, we will present an algorithm which matches the tape linear algorithm in simplicity and is more efficient.

3.4 The Hwang-Lin binary merging algorithm.

The Hwang-Lin algorithm is the combination of the binary insertion and the tape linear merging algorithm. This algorithm is used to solve the merge(m,n) problem with any $m,n \geq 0$. The main idea behind the algorithm is this : divide the longer list into m sublists of n/m each, then use binary search on the sublists. Each element of the smaller list is tentatively inserted into the corresponding sublist of the larger list. (See figure 3.6)

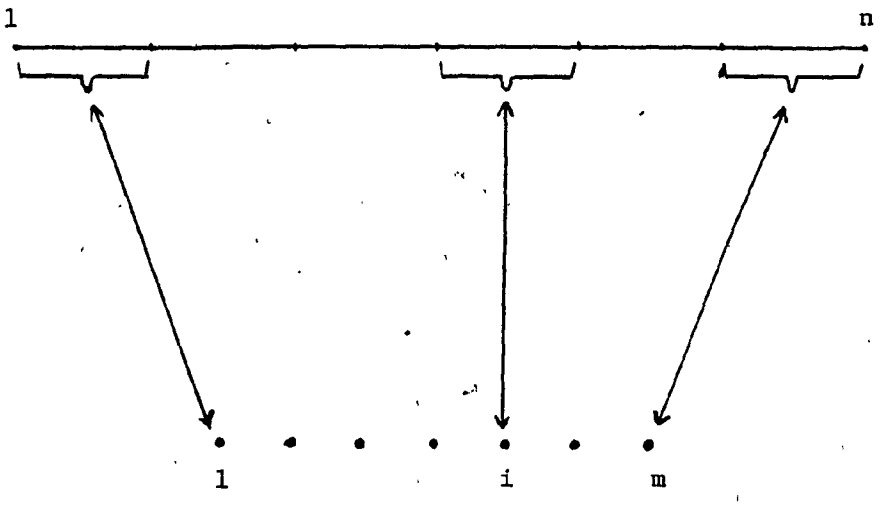


Figure 3.6

The binary insertion is efficient when the blocks in the larger list have the size of 2^t-1 $(t>0)$. If the ratio n/m is exactly a power of 2, then the longer list can be broken into m blocks of size 2^t each. Now, if n/m is not a power of 2, then we should divide the longer list in such a

way that we have blocks of size 2^t (except possibly one). So the larger list should be broken into blocks of size $2^t = 2^{\lfloor \log(n/m) \rfloor}$. The last element of the smaller list, A_m, is compared with B_{n-2^t+1}. If A_m is smaller, then 2^t elements are annexed from the larger list, i.e. removed from the larger list. If A_m is larger, then A_m is inserted into $B(n-2^t+2:n)$ by the binary insertion algorithm using t comparisons. The same process is iterated until one list is exhausted. Formally, the Hwang-Lin binary merging algorithm is described as :

Algorithm Hwang-Lin (hl) :

Step 1. If m=0 or n=0 then exit.

Step 2. Compute $t = \lfloor \log(n/m) \rfloor$.

Step 3. Comparre $(A_m : B_{n-2^t+1})$.

Step 4. If $A_m < B_{n-2^t+1}$ then annex 2^t elements of B; $n \leftarrow n-2^t$; go to step 1.

Step 5. If $A_m > B_{n-2^t+1}$ then insert A_m into $B(n-2^t+2:n)$. $n \leftarrow n-q$; $m \leftarrow m-1$; where q is defined by $B_q < A_m < B_{q+1}$.

The Hwang-Lin merging algorithm is represented by the tree as shown on figure 3.7.

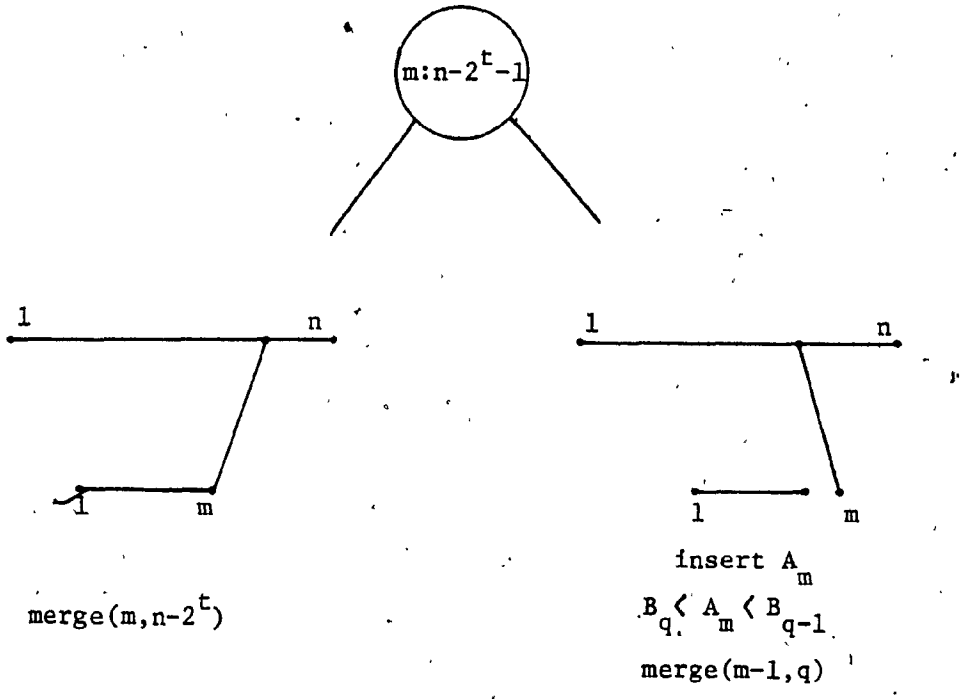


Figure 3.7

The worst case analysis of the Hwang-Lin algorithm is easy to perform. The idea is that neither the shorter nor the longer list should be exhausted prematurely.

The wosrt case analysis yields[16] :

(3.10) $M_{hl}(m,2n+e) = M_{hl}(m,n)+m$ where $m \leq n$; $e=0$ or 1

with the boundary condition :

(3.11) $M_{hl}(m,n) = m+n-1; \quad m \leq n < 2m.$

Theorem 3.6.

For $m \leq n$ and t such that $2^t.m \leq n < 2^{t+1}.m$:

(3.12) $M_{hl}(m,n) = (t+1)m + \lfloor n/2^t \rfloor - 1.$

Since $t = \lfloor \log(n/m) \rfloor$, we have :

$$M_{hl}(m,n) \sim m(\log(n/m)+2).$$

Proof :

The proof is by induction on n.

For $n \leqslant n < 2m$, (3.12) reduces to (3.11) because t=0.

Suppose $n \geqslant 2m$, then for some $p \geqslant m$ and e=0 or 1, n=2p+e.

Using (3.10) and the induction hypothesis, we have :

$$\begin{aligned} M_{hl}(m,n) &= M_{hl}(m,2p+e) \\ &= M_{hl}(m,p) + m \\ &= (t'+1)m + \left\lfloor \frac{p}{2^{t'}} \right\rfloor - 1 + m \end{aligned}$$

where $t' = \lfloor \log(p/m) \rfloor = t-1$.

Since

$$\left\lfloor \frac{n}{2^t} \right\rfloor = \left\lfloor \frac{p}{2^{t-1}} \right\rfloor + \left\lfloor \frac{e}{2^t} \right\rfloor = \left\lfloor \frac{p}{2^{t-1}} \right\rfloor = \left\lfloor \frac{p}{2^{t'}} \right\rfloor$$

we have :

$$M_{hl}(m,n) = (t+1)m + \left\lfloor n/2^t \right\rfloor - 1$$

Q.E.D.

When the ratio n/m is a power of 2, $n = 2^d.m$, (3.12) becomes

(3.13) $$M_{hl}(m, 2^d.m) = (d+2)m - 1.$$

Hwang and Lin[13] have shown that :

$$M_{hl}(m,n) < \lceil \log(\tbinom{m+n}{n}) \rceil + m.$$

However, a more powerful inequality is derived from theorem 3.4 which can be stated as follow :

Corollary 3.7.

For $n \geq 5m-3$, the Hwang-Lin binary merging algorithm does not require more than $M(m,n) + \lceil m/2 \rceil$ comparisons. In other words, we have :

$$M_{hl}(m,n) \leq M(m,n) + \lceil m/2 \rceil \quad \text{for } n \geq 5m-3.$$

The Hwang-Lin binary merging algorithm has another version which is simpler. This version can be described as:

Algorithm stahl(sh) :

Step 1. Compute $t = \lfloor \log(n/m) \rfloor$.

Step 2. If $n < 2^t$, go to step 6.

Step 3. Compare $(A_m : B_{n-2^t+1})$.

Step 4. If $A_m < B_{n-2^t+1}$ then annex 2^t elements of B; $n \leftarrow n-2^t$; go to step 2.

Step 5. If $A_m > B_{n-2^t+1}$ then insert A_m into $B(n-2^t+2:n)$; Once $B_q < A_m < B_{q+1}$ is known : $n \leftarrow n-q$; $m \leftarrow m-1$; go to step 2.

Step 6. Insert the elements of list A into list B, one by one using binary insertion.

This version of the Hwang-Lin algorithm is known as the static version[21], since the factor $t = \lfloor \log(n/m) \rfloor$ is computed only once in the first step of the algorithm.

The Hwang-Lin merging algorithm is improved by Manacher[20], and further improved by us[19]. We represent a brief sketch of the latter. In this improved algorithm, basically, the length to insert is computed to save the

comparisons as much as possible. This improved algorithm can be expressed as :

Algorithm improved hl(hb) :

Step 1. Compute $d = \lfloor \log(n/m) \rfloor$

$c_1 = n-2^d$; $c_2 = n-\lfloor (17/14)2^d \rfloor$;

$c_3 = n+1-\lfloor (12/7)2^d \rfloor$; $c_4 = n+1-\lfloor (41/28)2^d \rfloor$;

Step 2. Compare $(A_m : B_{c_1})$

If $A_m < B_{c_1}$ then $n \leftarrow c_1-1$; go to step 1

else go to step 3.

Step 3. Compare $(A_{m-1} : B_{c_2})$.

If $A_{m-1} < B_{c_2}$ then insert A_m into $B(c_1+1:n)$;

$n \leftarrow c_2-1$; $m \leftarrow m-1$; go to step 1

else go to step 4.

Step 4. Compare $(A_{m-2} : B_{c_3})$.

If $A_{m-2} < B_{c_3}$ then merge $A(m-1:m)$ with $B(c_2+1:n)$; $n \leftarrow c_3-1$; $m \leftarrow m-3$; go to step 1

else go to step 6.

Step 6. Merge $A(m-3:m)$ with $B(c_4:n)$; $(B_q < A_{m-4} < B_{q+1})$

$n \leftarrow q$; $m \leftarrow m-4$; go to step 1.

This algorithm can be explained by the following merge tree on figure 3.8 :

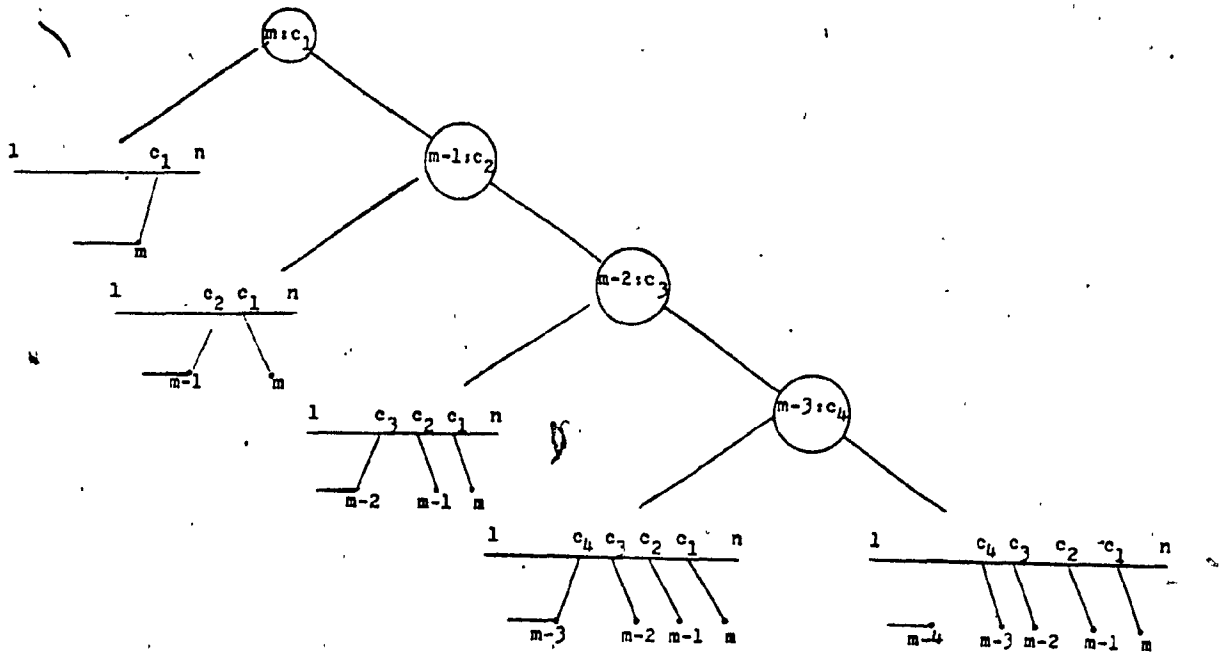


Figure 3.8

This improved algorithm is known as the dynamic version of the binary merge algorithm since the size of the shorter list (to be inserted into the longer list) is changed according to the ratio of n/m. However, the dynamic algorithm is better than the static one only when $n/m = 2^d$ (see [21]).

The worst case of this dynamic version yieldds :

$$M_{hb}(m,m.2^d) = (d + 2 - 13/84)m - 1$$

$$= (d + 155/84)m - 1. \quad (3.14)$$

It is improved by a factor of 13/84[19].

As for the average analysis, from figure 3.7 we can derive the recurrence formula for the Hwang-Lin binary merge algorithm as follow :

$$\bar{M}_{hl}(m,n) = p\bar{M}_{hl}(m,n-2^t) + \sum_{q=n-2^t+1}^{n} p_q \bar{M}_{hl}(m-1,n-q)$$

where

p = propability that $A_m < B_{n-2^t+1}$

$$= \frac{\binom{m+n-2^t}{m}}{\binom{m+n}{m}}$$

and p_q = probability that $B_q < A_m < B_{q+1}$

$$= \frac{\binom{n-q+m-1}{m-1}}{\binom{m+n}{m}}$$

The Hwang-Lin algorithm has the property that the element at the end of the shorter list (A_m of list A) is always compared first, and this end-element is handled (i.e. inserted into the longer list) before the task of dividing the problem into subproblems. This property is the characteristic of the R-class. (Note that the tape linear merge algorithm also belongs to this R-class but the Forward-Testing-Backward-Insertion algorithm (see section 3.5) does not). A modification of the Hwang-Lin binary merge algorithm was designed by Tanner[26]. Instead of treating the end-element first, he treated first the element

in the middle of the shorter list. Here, we present only the algorithm and its results. We refer to [26] for further details. This algorithm is known as the fractile algorithm.

Algorithm Fractile Insertion (fi) :

Step 1. Compute $f = \lceil m/2 \rceil$; $k_1 = \left\lceil n.\left(\frac{f}{m+1}\right)\right\rceil$

$\alpha = \left\lfloor \frac{1}{2}\log(n(1+n/m)) - 1.3 \right\rfloor$; $\Delta = 2^{\alpha}$

Step 2. Compare $(A_f : B_{k_1})$;

Step 3. If $A_f > B_k$ then $k_1 \leftarrow k_1 + \Delta$;

Repeat step 3 until either $k_1 > n$ or $A_f < B_k$

Step 4. If $A_f > B_k$ then $k_1 \leftarrow k_1 - \Delta$; Compare $(A_f : B_k)$.

Repeat step 4 until either $k_1 < 1$ or $A_f > B_k$

Step 5. Once an interval $[B_k, B_{k'}]$ is found in which A_f belongs to; insert A_f into $[B_k, B_{k'}]$ by binary insertion.

Step 6. After handling A_f, divide the problem into 2 subproblems (see figure 3.9).

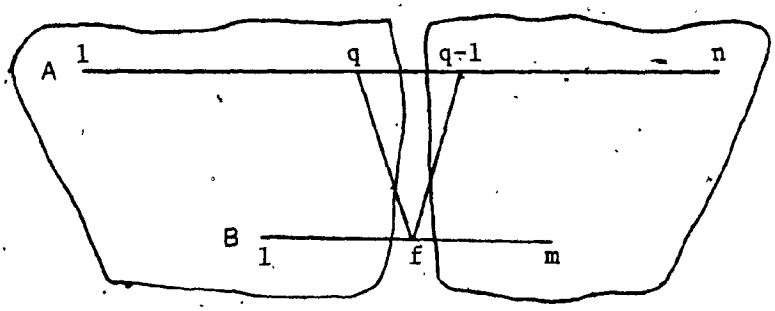


Figure 3.9

Merge (f-1,q) and merge (m-f-1,n-q). Solve these two subproblems by the fractile insertion algorithm.

Thus fractile insertion algorithm chooses an element of A ($A_f = A_{\lceil m/2 \rceil}$); it settles this element first besore dividing the problem into two subproblems.

Tanner[26] has shown that :

(3.15) $$\bar{M}_{fq}(m,n) < 1.06\log\binom{m+n}{n}$$

The fractile insertion algorithm provides a highly efficient technique for decomposing a large problem into many seperately processable ssmaller problems. As such it may make possible efficient parallellism in merging. However, the bound $1.06\overline{ITM}(m,n)$ is not really a good bound, and therefore the number of comparisons in the worst case may exceed that of other algorithms.

In the next section, we discuss an algorithm which is not in the R-class : the Forward-Testing-Backward-Insertion algorithm.

3.5. The Forward-Testing-Backward-Insertion algorithm.

This algorithm was developed in 1978 by Christen[06]. The algorithm has the basic idea, as its name suggests, that it keeps going on comparing the elements of the shorter list A, with the corresponding element of the longer list B until one element of A is found larger than B or the shorter list is exhausted. In figure 3.10 we assume that $A_{x+1} > B_i$.

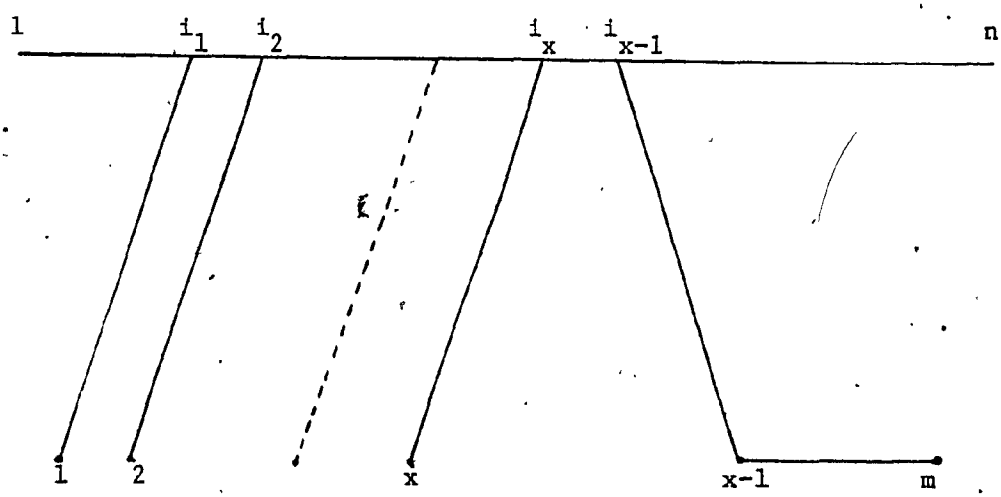


Figure 3.10

The next step is to insert the elements A_i for $i=1,\ldots,x$ by the binary insertion and the method itself. The fine points of this method can be found in [06]. Formally, the Forward Testing Backward Insertion can be expressed as below :

Algorithm Forward-Testing-Backward Insertion (ftbi) :

Step 1. Compute $j=\lfloor \log_4(1+n/m) \rfloor$; $k=\lfloor \log(n/m) \rfloor - 2j$.
$T(i,j) = 4^{j+1} + (i+1)*(4^{j+1}-1)/3$.

Step 2. If $j=0$, use algorithm h1, else go to step

Step 3. $i = 1$.

Step 4. Compare $(A_i : B_{2^k * T(j,i)})$.
If $A_i < B_{2^k * T(j,i)}$ then $i \leftarrow i+1$; go to step 4
else go to step 5.

Step 5. $i \leftarrow x$.

Step 6. Compare $(A_i : B_{2^k * T(j,i) - 2^{2j+k+1}})$.
If A_i is larger then binary insert A_i into $B(2^k * T(i,j) - 2^{2j+k+1} : 2^k * T(i,j))$; $i \leftarrow i-1$; go to step 6
else go to step 7.

Step 7. Recursively merge the first i elements of A with the first $2^k * T(j,i) - 2^{2j+k+1} - 1$ elements of B, and the last $(m-x)$ elements of A with the last $n - 2^k * T(j,i)$ elements of B.

The worst case bound of the Forward Testing Backward Insertion algorithm (ftbi) is computed as follows :

i) For $3m \leqslant n < 4m$:

$$M_{ftbi}(m,n) \leqslant 3m + \left\lceil \frac{n-m-2}{4} \right\rceil - 1 \qquad (3.16)$$

ii) For $2^{k+2} \leqslant n/m < 2^{k+3}$, and $k>0$:

$$M_{ftbi}(m,n) \leqslant (k+3)m + \left\lceil \frac{n-2^k(m+3)+1}{2^{k+2}} \right\rceil - 1 \qquad (3.17)$$

iii) For $2^{2j+k+2} \leqslant n/m < 2^{2j+k+3}$, where

$j = \lfloor \log_4(1+n/m) \rfloor$ and $k = \lfloor \log(n/m) \rfloor - 2\lfloor \log_4(1+n/m) \rfloor$

(3.18) $$M_{ftbi}(m,n) \leqslant (2j+k+3)m + \left\lceil \frac{n-T(j,m)2^{k+1}}{2^{k+2+2j}} \right\rceil - 1$$

where $T(j,m) = 4^{j+1} + \left\lceil \frac{(m-1)(4^{j+1}-1)}{3} \right\rceil$.

Note that case (ii) is a generalization of case (i) with j=0, and case (iii) is a generalization of case (ii) with k=0. Christen[06] proved that the ftbi algorithm saves at least $\sum_{J=1}^{k} \lfloor (m-1)/4^J \rfloor$ comparisons over the binary merge algorithm forr $n \geqslant 4^k.m$. Thus asymptotically, when n/m increases to infinity, at least m/3 comparisons are saved.

From Theorem 3.4, we can derive the following corollary:

Corollary 3.8.

i) The ftbi algorithm is M-optimal for $5m-3 \leqslant n \leqslant 7m$.

ii) For $n>7m$, we have

$$M_{ftbi}(m,n) \leqslant ITM(m,n) + \lceil m/6 \rceil .$$

The average performance analysis of the Forward Testing Backward Insertion has not been investigated. It is suspected that the task involves lengthly mathematical computations.

Note that the ftbi algorithm is better than the Hwang-Lin merging algorithm(both static and dynamic version)

in the worst-case performance. This fact suggests that the ftbi could be used to improve the sorting problem. Indeed, this idea leads us to chapter 5 on sorting algorithms. We will use the ftbi algorithm to derive various sorting algorithms which improve the S-optimal sort bounds.

We have exhibited various algorithms for merging(m,n) in the general cases. These algorithms are used for the general cases. Some have been known for a long time (tape linear), some are only known recently (ftbi,fractile). Except for some particular ranges, these algorithms are not optimal in general. The M-optimal merge(m,n) in general $m \geq 0$, $n \geq 0$ remains an open problem.

CHAPTER 4

MERGE ALGORITHMS : CASE $1 < m < 5$

In chapter three, we have discussed various algorithms for the general problem of merging(m,n) where $m \geq 0$. None of these algorithms is M-optimal, except for some special cases. In this chapter, we consider the cases where m is small ($1 \leq m \leq 5$). The problem of merge(m,n) using comparisons can be viewed from another view point : Instead of considering M(m,n), the minimum number of comparisons required for the merge(m,n) problem, we consider the largest length into which m elements could be merged using exactly k comparisons. We denote f(m,k) the largest length such that $M(m,n) \leq k$. Therefore, for a fixed m, M(m,n) can also be determined for every n N, if and only if f(m,k) is known for every $k \in N$. Similar to our previous notations, for an algorithm s, we denote

$$f_{s(m,k)} = \max\{n \in N \mid M_s(m,n) \leq k\}$$

and from this we have :

$$f(m,k) = \max \{f_{s(m,k)}\}$$

where the maximum is taken over all algorithms.

4.1. Results for m=1,2,3.

The binary insertion algorithm, or the merge(1,n) algorithm described in chapter 2 produces :

(4.1) $f(1,k) = 2^k - 1$ for $k \in N$.

Hwang and Lin[14] have shown that :

(4.2) $f(2,k) = \lfloor 17/14.2^k \rfloor - 1$ for $k=2r$,

(4.3) $= \lfloor 12/14.2^k \rfloor - 1$ for $k=2r-1$.

Hence :

$f(2,x) \sim (17/14).2^{x/2} = 1.2143\ 2^{x/2}$ if x is even,

$f(2,x) \sim (12\ 2/14).2^{x/2} = 1.2122\ 2^{x/2}$ if x is odd.

Combining these, one can derive the approximation :

(4.4) $f(2,x) \sim \sqrt{\frac{17}{14}.\frac{12}{14}.\sqrt{2}}.2^{x/2}$

$\sim 1.2132.2^{x/2}$ for any $x \in N$.

From (4.2) and (4.3), we get :

(4.5) $M(2,n) = \lceil \log_{\frac{14}{17}}(n+1) \rceil + \lceil \log_{\frac{7}{12}}(n+1) \rceil$.

In the case of merging(3,n), Hwang[11] has shown that :

$f(3,k) = (0,1,1,2,3,4,6,8)$ for $k \leq 8$,

and for $r \geq 3$

(4.6) $f(3,3r) = \lfloor (43/7)2^{r-2} \rfloor - 2$,

(4.7) $f(3,3r+1) = \lfloor (107/7)2^{r-3} \rfloor - 2$,

(4.8) $f(3,3r+2) = \lfloor (17.2^r - 6)/7 \rfloor - 1$

Or

$f(3,x) \sim (43/28)2^{x/3} \sim 1.5357\ 2^{x/3}$ if $x=0 \pmod 3$,

$f(3,x) \sim (107/(56\sqrt[3]{2}))2^{x/3} \sim 1.5165\ 2^{x/3}$ if $x=1 \pmod 3$,

$f(3,x) \sim (17/(7\sqrt[3]{2^2}))2^{x/3} \sim 1.5299\ 2^{x/3}$ if $x=2 \pmod 3$.

Combining these equations, we have the approximation :

$$f(3,x) \sim \sqrt[3]{\frac{43}{28}\cdot\frac{107}{56\sqrt[3]{2}}\cdot\frac{17}{7\sqrt[3]{2^2}}}\cdot 2^{x/3}$$

$$(4.9) \qquad \sim 1.5274\ 2^{x/3}.$$

From (4.6), (4.7) and (4.8), we have :

$$(4.10)\quad M(3,n) = \left\lceil \log_{\frac{28}{43}}(n+1) \right\rceil + \left\lceil \log_{\frac{56}{107}}(n+1) \right\rceil + \left\lceil \log_{\frac{7}{17}}(n+13) \right\rceil .$$

4.2. Some useful lemmas.

In [11], Hwang gave a strong conjecture :

$$f(m,mr+i) \sim 2f(m,m(r-1)+i).$$

This is quite reasonable since, with a fixed m, this conjecture tell us that the maximum length n to be merged with m can be approximately doubled if the number of comparisons is increased by m. We suggest more powerful inequalities which generalize Hwang's conjecture as follows:

$$2f(m,k) + 1 \leqslant f(m,k+m) \leqslant 2f(m,k) + m.$$

The first inequality is proved by the following lemma and the second one, unfortunately, can be only proved partially. However, it is strongly believed that the sketch of the proof given here may lead to the complete proof of our conjecture.

Lemma 4.1.

$$f(m,k+m) \geqslant 2f(m,k)+1 \qquad \text{for all } m,k \geqslant 0.$$

Proof :

We give a constructive proof by giving an algorithm to

merge $(m,2f(m,k)+1)$ using at most $k+m$ comparisons.

Consider the list $B(1:2f(m,k)+1)$, we pick out the even indexed elements $B_2, B_4, \ldots, B_{2f(m,k)}$. These elements constitute a list of length $f(m,k)$. By definition, we can merge m elements of list A(1:m) with these $f(m,k)$ elements using at most k comparisons.

After this merging process, every element of list A is known either to lie between some consecutive even elements of list B, or at two ends of list B(i.e. $< B_2$ or $> B_{2f(m,k)}$). To complete the merge problem $(m,2f(m,k)+1)$, we simply compare these m elements of list A with at most m corresponding odd elements of list B, and it is obvious that at most m more comparisons are required.

This completes the proof since we have merged $(m,2f(m,k)+1)$ using at most $k+m$ comparisons.

Q.E.D.

Conjecture

$$f(m,k+m) \leqslant 2f(m,k) + m.$$

Partial proof of the conjecture :

The proof is a double induction on (m,k). The basis of induction is proved as follows :

We have

$$f(m,2m) = m+1.$$

Thus, let $k=2m$, we have to show that :

$$f(m,k+m) \leqslant 2f(m,k) + m,$$

or

$$f(m,3m) \leqslant 2(m+1) + m = 3m+2.$$

This is true due to Theorem 3.3(i) since :

$$M(m,3m+2) \geqslant M(m,3m+1) \geqslant /M\backslash(m,3m+1)$$
$$\geqslant 2m + m = 3m.$$

Hence,

$$f(m,3m) \leqslant 3m+2.$$

The usefulness of the inequalities

$$2f(m,k) + 1 \leqslant f(m,k+m) \leqslant 2f(m,k) + m$$

can be expressed by the following lemma :

Lemma 4.2.

If $f(m,k+m) \leqslant 2f(m,k)+m$ then

$f(m,X) = A$ implies that $(A+1)2^k \leqslant f(m,X+mk) \leqslant (A+m)2^k$.

Proof :

By applying the above inequalities repeatedly, we get the result of the lemma.

Q.E.D.

This lemma gives an upper bound as well as a lower bound for $f(m,k)$. The following example illustrates the usefulness of the lemma.

Example.

Suppose we know that $f(2,10) = 37$, then without knowing the merge(2,n) algorithm nor the $f(2,k)$ bound, we can derive that

$$38.2^k \leq f(2,10+2k) \leq 39.2^k$$

or

$$(38/32)2^k \leq f(2,2k) \leq (39/32)2^k$$

$$1.1875\ 2^k \leq f(2,2k) \leq 1.21875\ 2^k \qquad (4.11)$$

and indeed we have

$$f(2,2k) \sim (17/14)2^k \sim 1.2142\ 2^k.$$

The following lemma will be used in the next section :

Lemma 4.3[10].

In the algorithm of merging two lists A(1:m) and B(1:n), suppose for a fixed i neither of the following two cases can be done in k comparisons :

i) The first comparison is $(A_i:Bx)$ and results in $A_i<Bx$.

ii) The first comparison is $(A_i:Bx-1)$ and results in $Ai>Bx-1$. then any algorithm whose first comparison involves A_i requires more than k comparisons.

Proof :

Suppose an algorithm starts with the first comparison $(A_i:B_y)$. If $y>x$ then with the outcome $A_i<B_y$ (see figure 4.1) we have to use more comparisons than stated in (i) of the lemma.

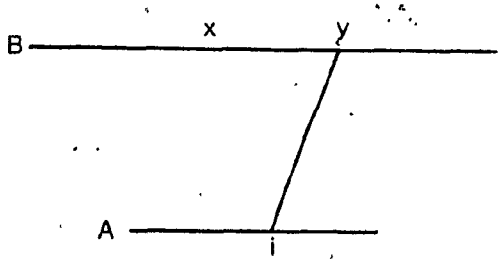


Figure 4.1

If $y<x$, or $y\leq x-1$ then with the outcome $A_i>B_y$ (see figure 4.2)

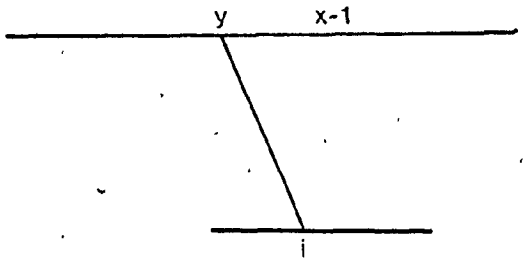


Figure 4.2

we have to use more comparisons than stated in (ii) of the lemma.

In either case, at least one outcome will result in more than k comparisons and the proof hence is complete.

Q.E.D.

Lemma 4.4.

In the process of merging, if more than one task can be done independently, then the order of doing these tasks does not affect either the worst-case or the average-case bounds of the whole problem.

Proof :

It is sufficient to prove the case of two tasks which

could be done independently, since if there are more than two tasks, we repeat the same process in a finite numbeer of steps.

Let A,B denote the trees for tasks a,b respectively.

Figure 4.3

If we perform task a first, the resulting tree will be as shown in figure 4.4.

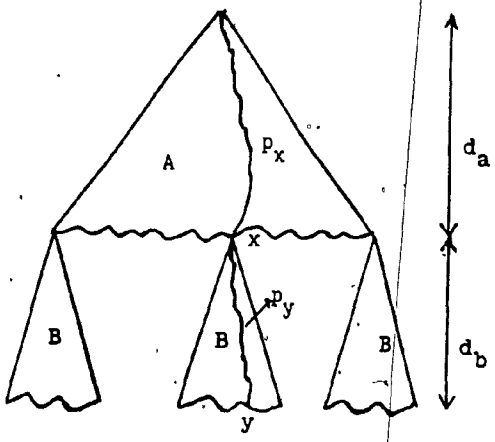


Figure 4.4

If we perform task b first, the resulting tree will be as shown in figure 4.5.

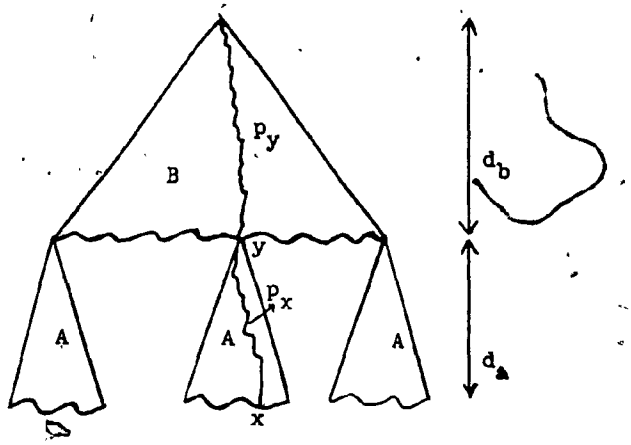


Figure 4.5

In the worst case, the depths of both trees are :

$$d_a + d_b = d_a + d_b$$

where d_a, d_b are the depthd of tree A and tree B, respectively.

In the average case, the external path length for the first resulting tree is

$$\sum_{x \in out_a} \sum_{y \in out_b} (p_y + p_x)$$

and for the second one is :

$$\sum_{y \in out_b} \sum_{x \in out_a} (p_x + p_y)$$

where x,y denote the leaves of tree A,B respectively,

P_x, p_y denote the path length from the root downto x,y respectively, and

out_a, out_b are the set of leaves of tree A,B respectively.

Hence, they are equal since out_a, out_b are finite.

The epl is computed as follows :

$$epl = epl(A)*|out_b| + epl(B)*|out_a|$$

where $epl(A)$: external path length of tree A,

and $|out_a|$: number of leaves of A.

Q.E.D.

4.3 A Top-Down approach for the merge (4,n) problem :

In [12] and [11], the merge(2,n) and (3,n) problems are solved by giving the algorithms together with the proof of their optimality. This approach is known as the heuristic approach.

In this section, we introduce a top-down approach to solve the (4,n) problem. This approach analyses and chooses the best possible pairs of objects form the two merged lists A and B for comparison of every level in the merge tree. One distinct advantage of our approach here over the heuristic ones is that efficiency is inherent in our method of exhaustive search, abandonment and acceptance of proper elements for comparison at each level in the merge tree. Our method is no less complex than the heuristic ones; however, our method explains the why and the how of each choice for comparison .

Before giving an algorithm, conventions and notaions should be made to facitilate the presentation.

We will present a merging algorithm in a format of a rooted binary merge tree where each internal node is associated with a comparison of two elements, the first is an element from the shorter list A(1:4), and the second is an element from the longer list B(1:n). If the first element is smaller, then we follow the left branch. Otherwise, we follow the right branch. These two branches lead to further comparisons which are represented by an internal nodes at the low level of the tree. At a given

stage of the merge process, a certain number of comparisons have been made and partial knowledge on the relations between A_i's and B_j's has been accumulated. We use a diagraph, called a configuration, to characterize the known relations at the current stage. In a configuration, each node represents an element and a link x y indicates the relation that x is less than y. The integer under one element or a group of elements of the shorter list denotes the required comparisons to insert or to merge these elements or to insert this element into the longer list B. Figure 4.6 gives an example of a configuration :

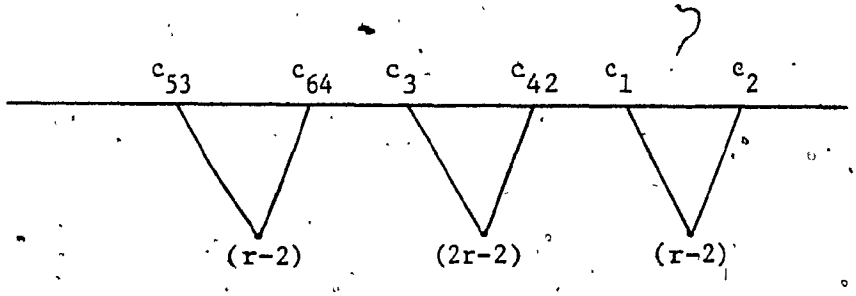


Figure 4.6

Configuration in figure 4.6 indicates :

$c_{53} < A_1 < c_{64}$; $c_3 < A_2$; $A_3 < c_{42}$; $c_{11} < A_4 < c_2$,

A_1 is inserted into the sublist $B(c_{53}:c_{64})$ in at most r-2 comparisons,

(A_2, A_3) are merged with the sublist $B(c_3:c_{64})$ in at most 2r-2 comparisons,

A_4 is inserted into the sublist $B(c_1:c_2)$ in at most r-2 comparisons.

A configuration X is said to dominate a configuration Y if every relation in X is implied by a relation in Y. Note that X dominates Y implies that the number of comparisons required to do X cannot exceed the number of comparisons to do Y. The first comparison in the merge process is associated with the root of the merge. The terminal nodes (leaves) of the tree are associated with the final configuration which can be readily analyzed, i.e., can be used to compute the number of comparissons required. In addition, we let c_{ij} denote the jth node (counted from left to right) in the ith level of the tree. If there is only one node in a level, we omit the second subscript j in c_{ij}.

The values f(4,k) for $k \leq 10$ are known[15]. We have :

f(4,i) = (0,0,1,1,2,3,4,5,6,8)

where i = (1,2,...,10)

Next we will show :

f(4,i) = (10,12,15,18,22,26)

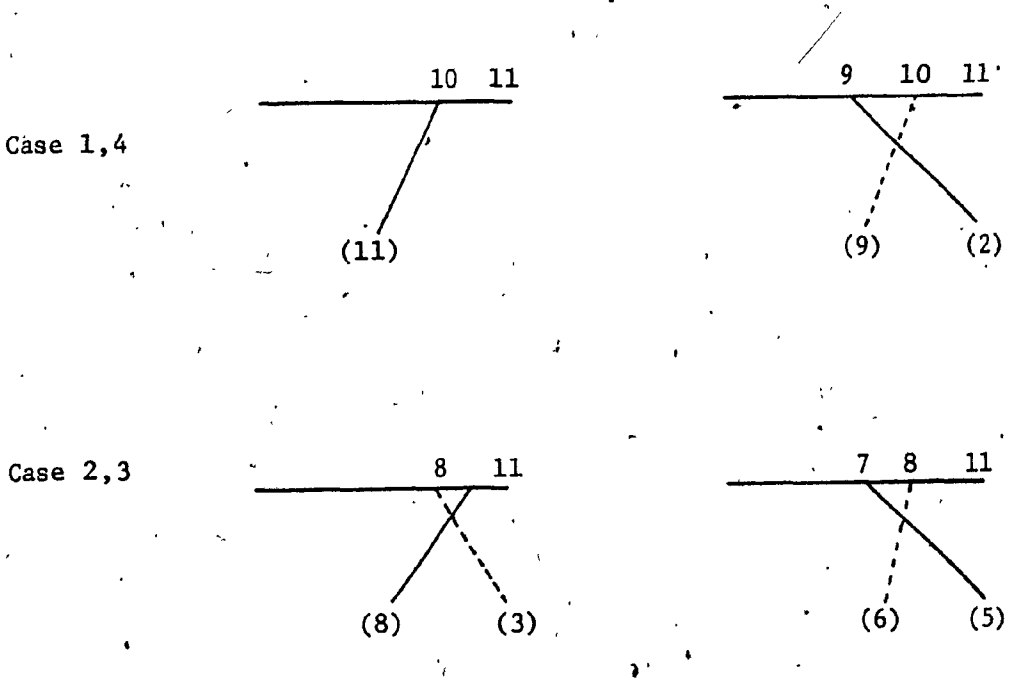
where

i = (11,12,13,14,15,16)

We use Lemma 4.4 to establish the optimality in these cases.

Proof of f(4,11) :

We compare $(A_1:B_2)$. If A_1 is smaller then we need one more comparison to insert A_1 with 9 comparisons left and we can merge 3 elements of A to at most 10 elements of B. If A_1 is greater then we merge 4 elements of A to 8 elements of B in 10 comparisons; in this case the length of list B is also 10. In order to establish the optimality it is sufficient to show M(4,11) > 12. The following diagram prove the claim by means of Lemma 4.3 :



Case i for i = 1,..,4 implies that the first comparison involves the element A_i.

Figure 4.7

The proofs of f(4,12), f(4,13) are similar to the proof of f(4,11).

Thus we give only the figures for the proofs. These figures are self-explanatory.

Proof of f(4,12) :

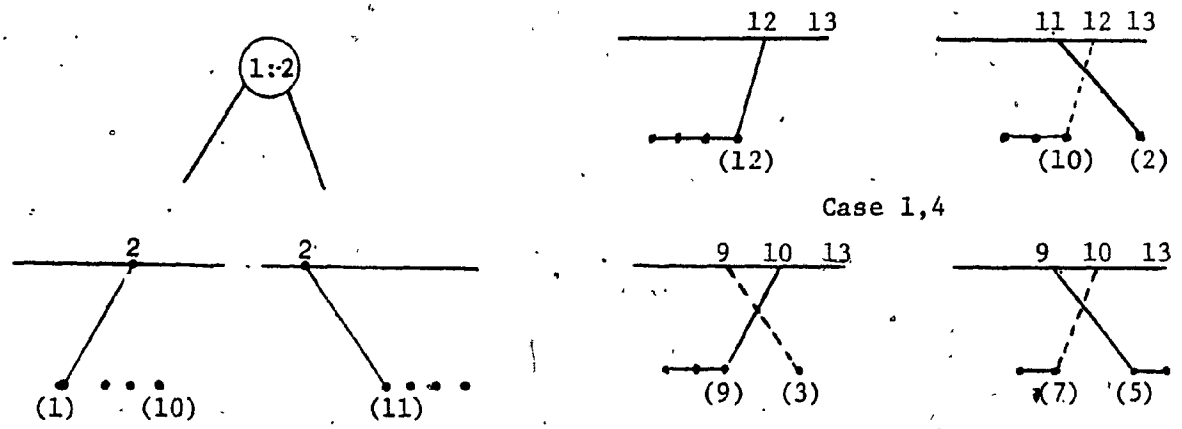
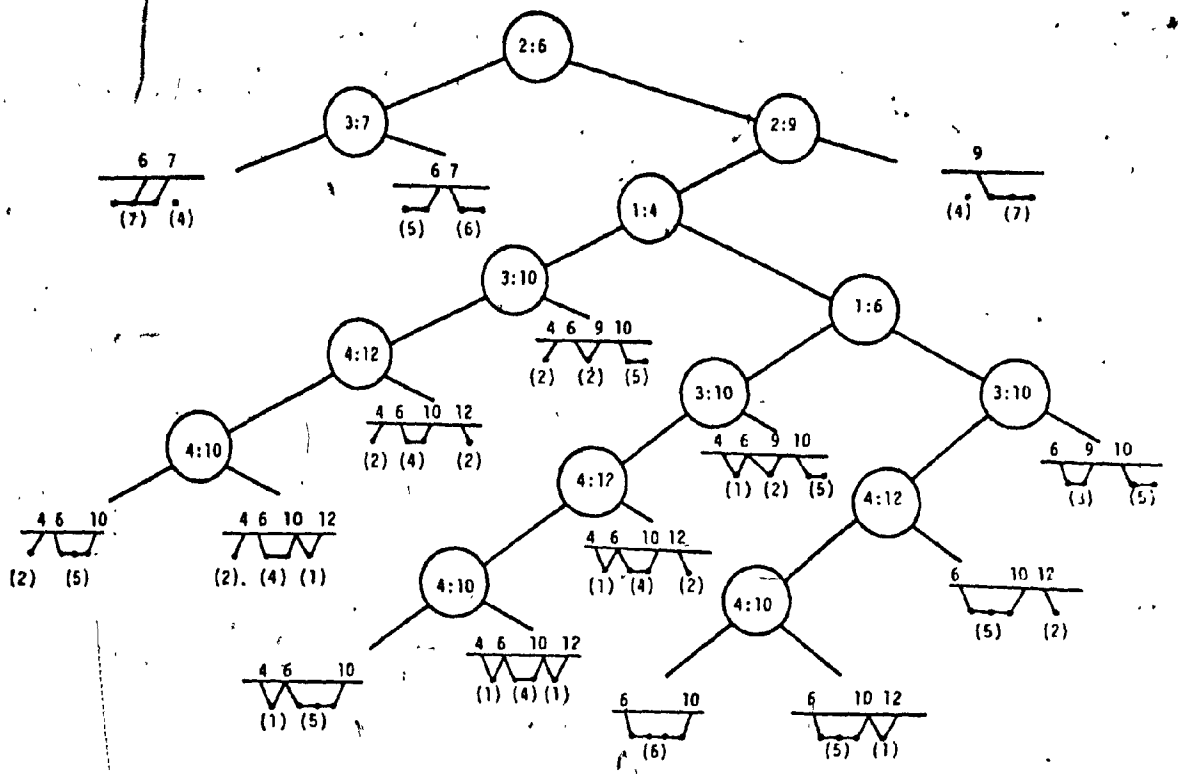
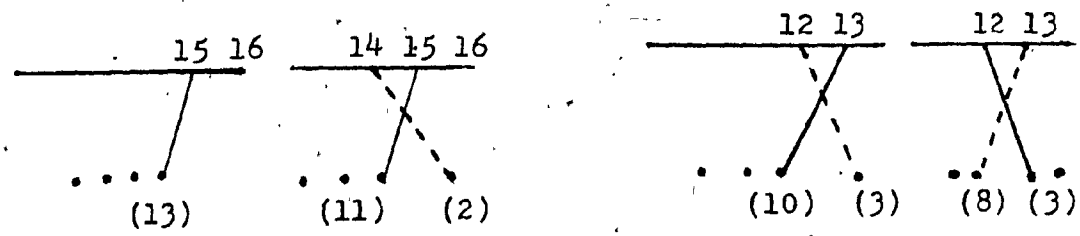


Figure 4.8

Proof of f(4,13) :



Merge tree



Case 1,4

Case 2,3

Optimality

Figure 4.9

Proof of f(4,14):

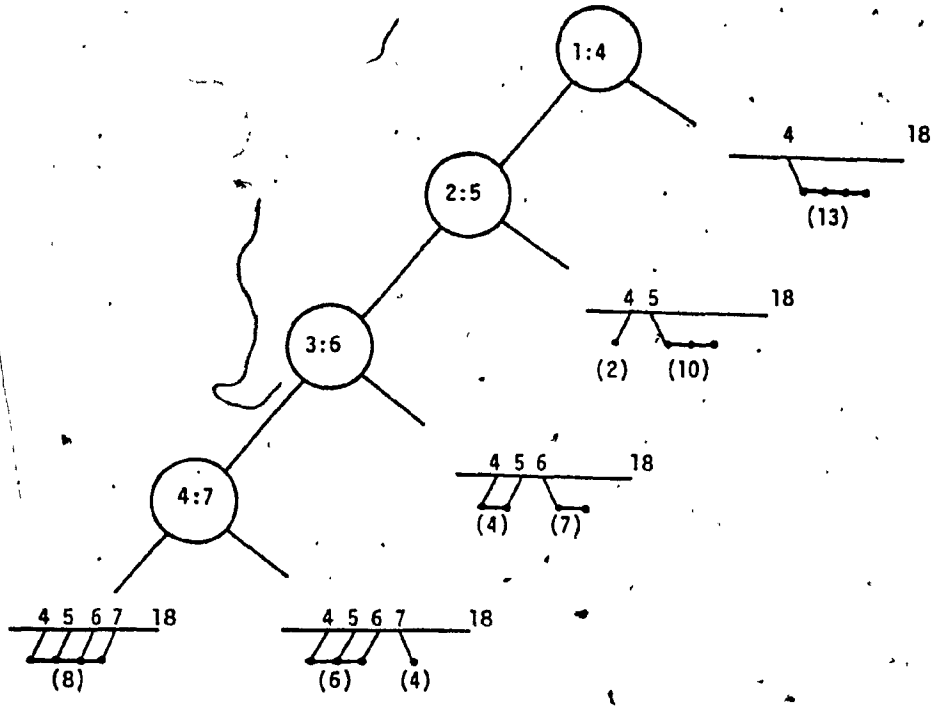


Figure 4.10

The optimality for f(4,14) and also for f(4,15), f(4,16) follows from the fact that the Forward Testing Backward Insertion algorithm is optimal for $5m-3 \leq n \leq 7m$ (Corollary 3.8, page 51). Hence, we give here the merge tree for f(4,15) and f(4,16).

Proof of f(4,15) :

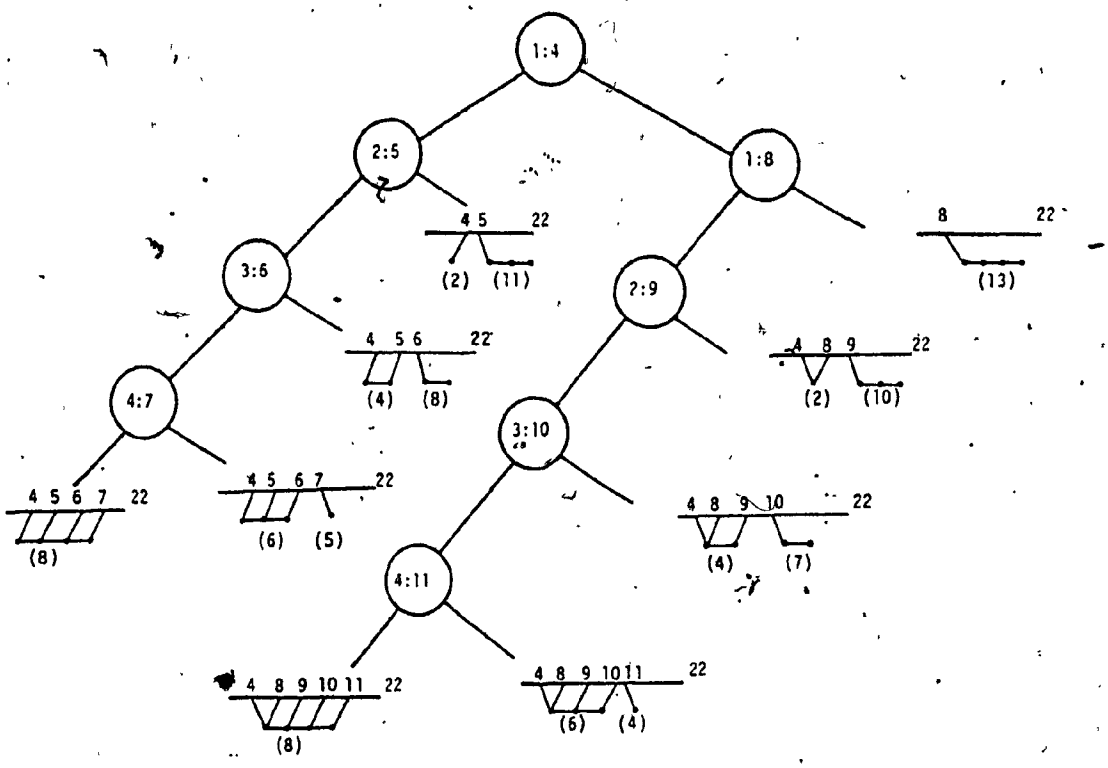


Figure 4.11

The merge tree for f(4,16) is exactly the same as figure 4.11. However, the second number of comparisons(written in brackets) in each terminal configuration is increased by one. Hence the total length is 26.

Now we set up the merge tree for the general cases :

Our task is to determine f(4,k) by a merge tree. Following Hwang's M-optimal merge trees in the case (3,n) merging. We divide the task into four cases : k=4r, k=4r+1, k=4r+2 and k=4r+3. The four problems according to these four cases are intimately related together. The longer length of f(4,4r) will lengthen the lengths f(4,4r+1), f(4,4r+2) and f(4,4r+3) and vice versa. Thus, it is sufficient to use the top-down approach for only one case, says k=4r, the other cases will be based on this case.

4.2.1 Merge tree(4,f(4,4r)) :

Our objective is to find f(4,4r). Let $c_\emptyset = f(4,4r) = c.2^r$. In fact, $1 < c < \sqrt[4]{24}$ must hold. Our aim is to maximize f(4,4r) or equivalently to maximize $c_\emptyset$.

To set up the merge tree, first we start at the root of the tree, find a suitable pair of element $(A_i : B_j)$ to be compared. Once we found such a pair, we repeat the same process at every level of the merge tree until we reach the terminal nodes which are the configurations ready to analyze. These configurations are reduced to either the cases merge(2,n) or merge(3,n) or the case merge(4,n) itself recursively. (The reason we call the approach the top-down approach is because we start from the root downto the leaves of the tree).

Choice at level 1 (the root) :

Since comparing A_1 with some element in $[1,c_\emptyset]$ is

symmetrical to comparing A_4 with some other element in $[1,c_\emptyset]$. The symmetry also happens for A_2 versus A_3. This allows us discuss only two cases (A_2 and A_4) and chooses the better of the two without missing any argument.

Suppose we choose A_4 to compare with some element B_t We will have the merge tree as follow :

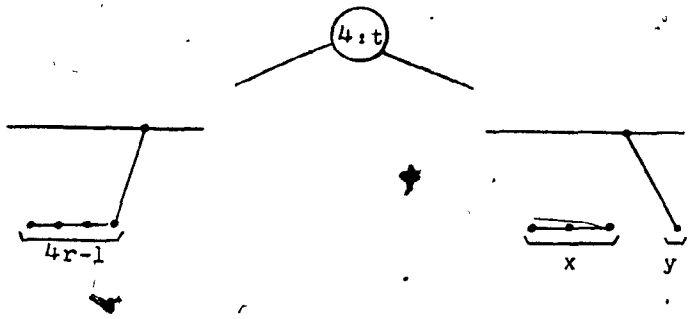


Figure 4.12

From the left hand configuration we must have $t=f(4,4r-1)$. If $f(4,4r)=c.2r$ then the right side is algebraically equivalent to

$$f(4,4r) = f(4,4r-1)+2^y-1$$

$$f(4,4r) = f(3,x),$$

where $x+y = 4r-1$. Permitting nonintegral solutions for x and y we obtain $c\sim1.8293$, where we have made use of the approximate results $f(4,4r-1)\sim c\ 2^r/\sqrt[4]{2}$ and $f(3,x)\sim1.5274\ 2^{x/3}$.

Now, suppose we choose 2 to compare with an element t' of the longer list, we have the following :

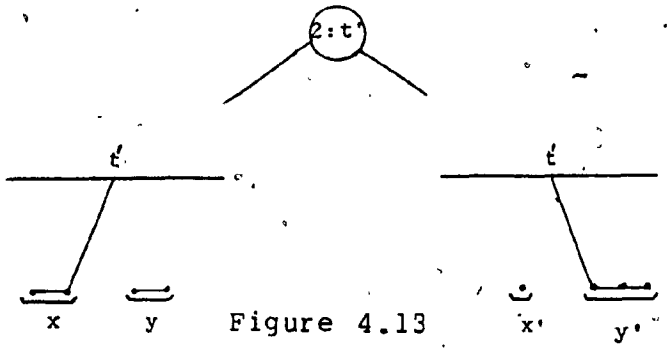


Figure 4.13

For the left side configuration we have

(4.12) $t'=f(2,x)+1; \quad f(4,4r)=f(2,y); \quad x+y=4r-1.$

Once again using approximations(see equation 4.4), we have

(4.13) $$t'= \frac{(1.2132)^4.2^{2r-1}}{c^2}$$

For the ~~right~~ side configuration we have

(4.14) $f(4,4r) - t' = f(3,y'); \quad f(4,4r) = f(1,x');$

$x'+y' = 4r-1$

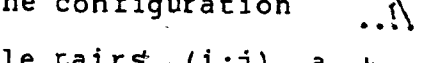
Substituting for t' and eliminating x',y' we get

(4.15) $$c(c-(1.2132)^2/c\sqrt{2})^3 = 1.7815.$$

The only real root of this equation gives $c \sim 1.6541$.

From the foregoing discussion we conclude that we must compare A_4 with B_t, $t=f(4,4r-1)$, at level 1. Note that t is fixed only relatively and hence is not obvious whether the left side of the configuration in level 1 should be developed or not.

Choice at level 2 :

Next we choose to develop the configuration  and we are faced with several possible pairs, (i:j), $a_i:b_j$. We shall systematically discuss all the possible choices.

Case 1 :

We have the configurations :

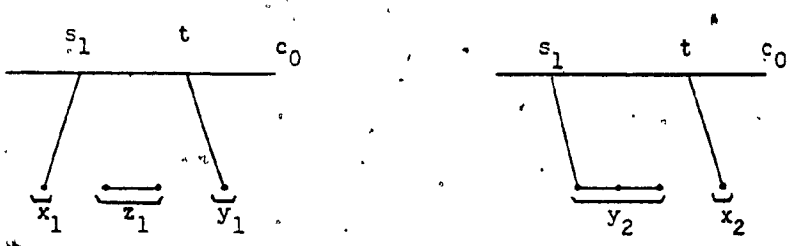


Figure 4.14

For the left hand side, we have $s_1 = 2^{x_1}$, $c_0 - t = 2^{y_1}$ and $c_0 = f(2,z_1)$ where $x_1+y_1+z_1=4r-2$. We write $f(2,z_1)=1.2132*2^{z_1/2}$ and eliminate x_1,y_1,z_1 to get $s_1=2^{r-2}(1.2132)^2/(0.1591)c^3$

For the right hand side, we have

$c_0-t = 2^{x_2}$, $c_0-s_1=f(3,y_2)$, $x_2+y_2=4r-2$.

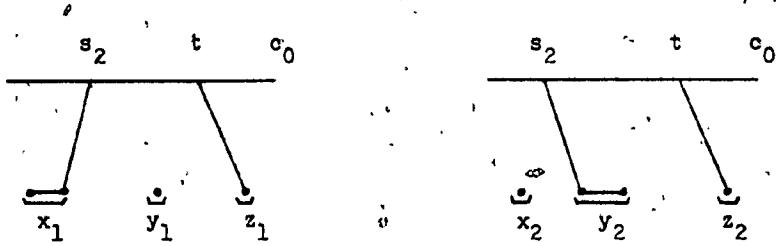
Substituting for s_1 and eliminate x_2,y_2 we get the equation :

$$c(c-2.3128)c^3)^3 = 5.5987,$$

and hence c = 1.8295.

Case 2 : We shall compare $(2:s_2)$ and consider two subcases corresponding to $s_2 < t$ and $s_2 > t$.

$s_2 < t$: We have the configuration



For the left hand side we have $f(2,x) = s_2$, $c_\emptyset - t = 2^{z_1}$ and $c_\emptyset = 2^{y_1}$ with $x_1 + y_1 + z_1 = 4r-2$. For the right hand

side we have $c_\emptyset = 2^{x_2}$,

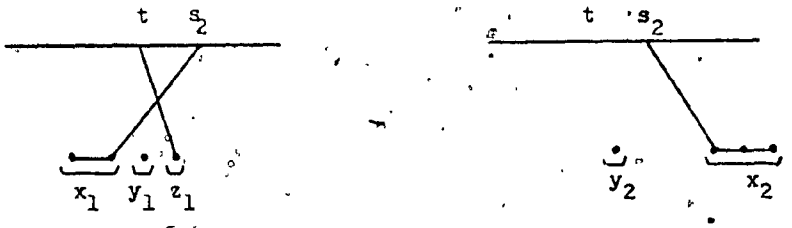
$c_\emptyset - s_2 = 2_y{}^{y_2}$ and $c_\emptyset - t = 2^{w_2}$ with $x_2+y_2+z_2+w_2 = 4r-2$.

From these we derive

$$s_2 = 0.5c_0$$

Using this and eliminating x_2, y_2, z_2 we have $c=1.7440$

$s_2 > t$: We have the configuration :



The equations for the left side configuration will be the same as in the previous case. For the right hand side we will have $f(3,x_2) = c_0 - s_2$, $c_0 = 2^{y_2}$ with $x_2+y_2 = 4r-2$. A routine and simple algebraic elimination gives $c = 1.7004$.

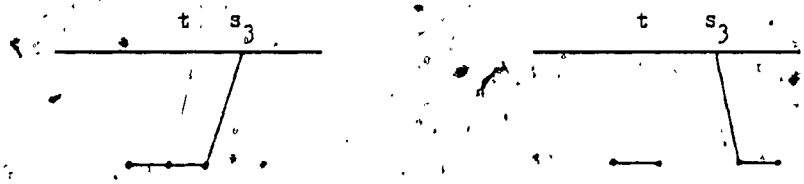
Thus far our analysis suggests that it is better to choose 1 to compare with the s determined in our Case 1 analysis. We should discuss two more cases. Because our discussions are similar, for each case below we show the configurations and the choice which maximizes c.

Case 3 : Compare 3 with s_3.

For $s_3 < t$, we have :

we get $s_3 = (1.7556\ 2^r)/\sqrt[3]{c}$ $c = 1.8259$

For $s_3 > t$ we have



$c = 1.8162$ and $s_3 = (c - 2.3128/c^3)2^r$

Case 4 : Compare 4 with s_4 :

When $s_4 < t$ we do not get any more information than we had in level 1. Hence we must choose $s_4 > t$. We have the configurations

and the equations :

$s_4 - t = 2^{x_1}$ $\quad s_4 = f(3, y_1)$ $\quad x_1 + y_1 = 4r-2$

for the left side ; and $c_0 - s_4 = 2^{y_2}$, $c_0 = f(3, x_2)$, $x_2 + y_2 = 4r-2$ for the right hand side . we have $(s_4 - t)s_4^3 = (c_0 - s_4)^3 c_0^3$

Let $c_0/s_4 = \lambda$, $c_0 = c\, 2^r$ and $t = c\, 2^r/\sqrt[4]{2}$, we have the equation :

$$\lambda^4 - \lambda^3 + 0.8409\lambda - 1 = 0.$$

$\lambda = 1.0762$ is an approximate root of this equation. Moreover we have $c = 1.8627$ We also have $s_4 - t = (0.1731)2^r$ for this value of c and this implies that we need $r-2$ comparisons to merge a_4 in the interval of length $s_4 - t$. Thus the point t is fixed relative to s_4. Observe that if the right side configuration is to be a terminal node we must have $y_2 = r-3$ and $x_2 = 3r+1$. In other words s_4 is fixed with respect to $c_\emptyset$ and t is now fixed with respect to s_4 . With this choice of s_4 we conclude :

i) The left branch of level 1 can be a terminal configuration and

ii) The choice of s_4 (at a distance 2^{r-3} from the right end of list B) to be compared with 4 is the best amond all possible choices of pairs for comparisons

Hence our partial merge tree is given in figure 4.15

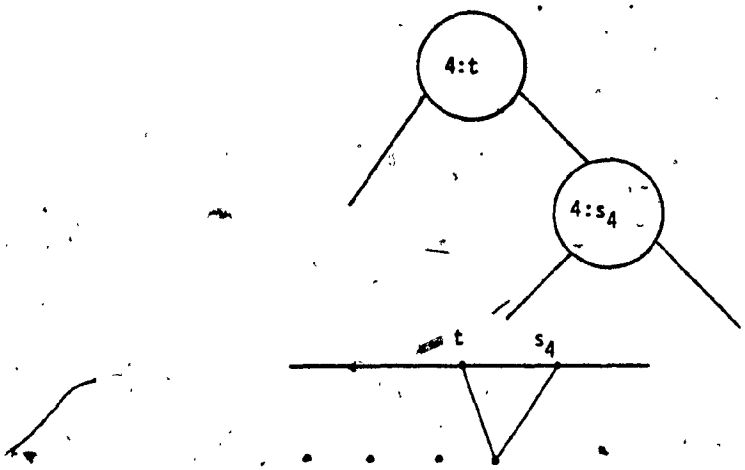


Figure 4.15

where $s_4 \le t = 2^{r-2}$, $c_0 - s_4 = 2^{r-3}$, $t = f(4,4r-1)$.

Since the position s_4 is fixed from the right we need to consider the merging of $(3,s_4)$ problem .

Here, we can continue the top-down approach to go to further levels. However, the cases to be analysed are exponentially increased with the level of the tree. For instances, if we have a merge tree of 10 levels, we have to analyse more than $2^{10} = 1024$ cases (which again may contain more subcases). It is desirable to avoid it . Now, the remaining problem is to handle the element A_4, and merge problem$(3,s_4)$. The merge(3,n) problem has been solved completely , we could adapt this merge tree to our case and therefore reduces the complexity of the cases to be analysed. Using this adaption, we get the following merge tree as in figure 4.16

The results we obtaiin from this adaption, together with three other cases (k=4r+1, k=4r+2 and k=4r+3) are as

follows :

(4.16a) $f(4,4r) = \lfloor (7/4).2^r \rfloor - 4; \quad r \geq 5$

(4.16b) $f(4,4r+1) = \lfloor (115/56).2^r \rfloor - 2; \quad r \geq 4$

(4.16c) $f(4,4r+2) = \lfloor (17/7).2^r \rfloor - 1; \quad r \geq 4$

(4.16d) $f(4,4r+3) = \lfloor (41/14).2^r \rfloor - 1; \quad r \geq 4$

Combining equations (4.16a-d), we have an upper bound for M(4,n) as follows:

$$M(4,n) = \left\lceil \log_{\frac{4}{7}}(n+1) \right\rceil + \left\lceil \log_{\frac{56}{115}}(n+\tfrac{13}{7}) \right\rceil + \left\lceil \log_{\frac{7}{17}}(n+\tfrac{13}{7}) \right\rceil + \left\lceil \log_{\frac{14}{41}}(n+\tfrac{13}{7}) \right\rceil$$

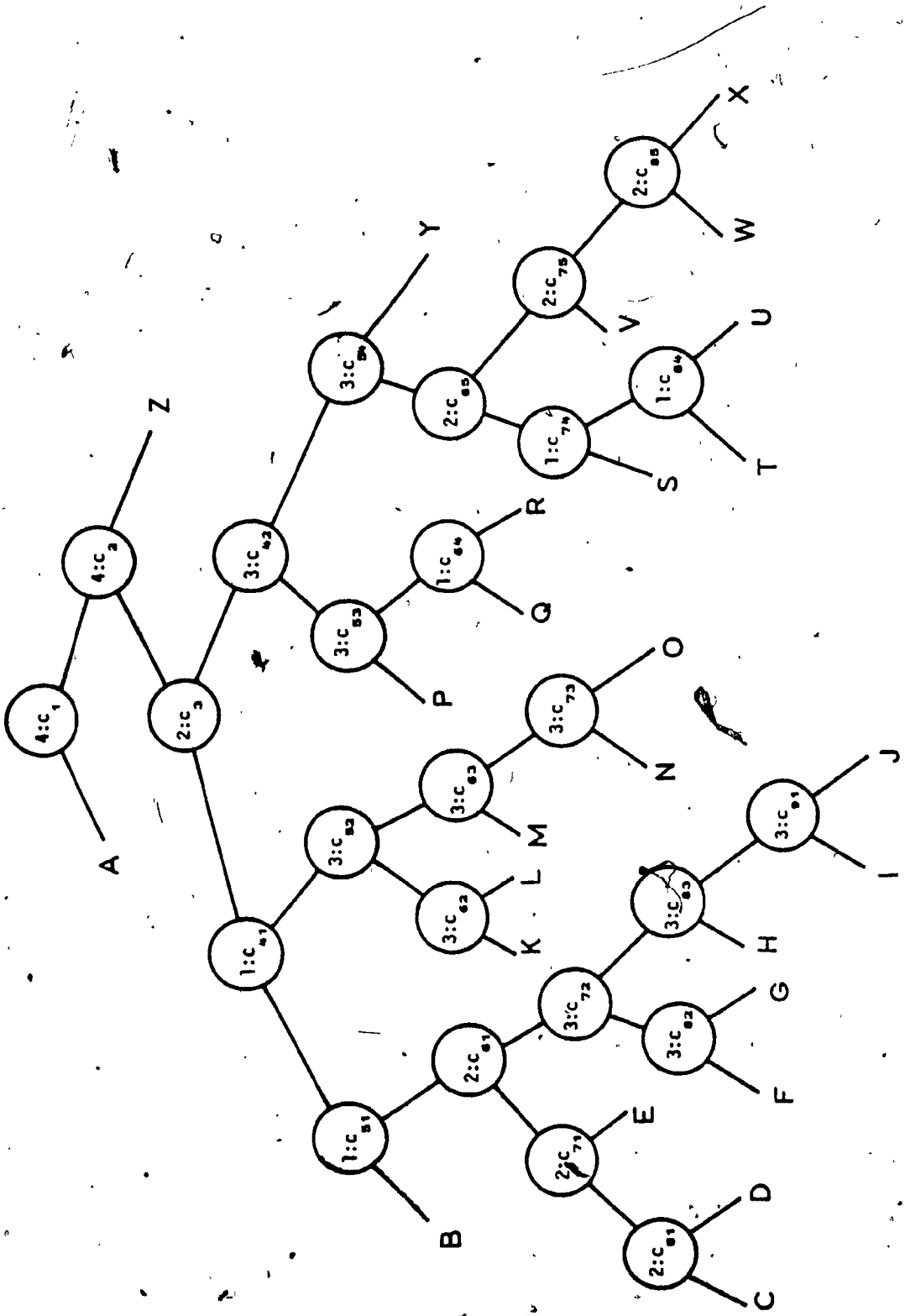

4:c1
4:c3
2:c3
1:c81
3:c82
3:c83
1:c84
3:c85
2:c85
1:c71
2:c75
3:c72
3:c73
2:c71
3:c81
3:c83
3:c82
2:c81
A
Z
B
Y
R
Q
P
X
W
V
U
T
S
O
N
M
L
K
J
I
H
G
F
E
D
C

Figure 4.16

The definitions for the notations used in Figure 4.16 are defined below :

$$c_0 = f(3,3r) + f(2,2r-5)$$

$$c_2 = c_0 - 2^{r-3} + 1.$$

$$c_1 = c_2 - 2^{r-2}$$

$$c_3 = f(3,3r-3)$$

$$c_{42} = f(3,3r-4) + 2^{r-1} + 2^{r-2}$$

$$c_{54} = f(3,3r) - 2^{r-4}$$

$$c_{53}$$

$$c_{64} = c_{84} = 2^{r-1} + 2^{r-2}$$

$$c_{65} = f(2,2r-3) + 2^{r-1} + 2^{r-2}$$

$$c_{75} = c_{65} + 2^{r-3}$$

$$c_{85} = c_{75} + 2^{r-4}$$

$$c_{41} = f(3,3r) - c_{42}$$

$$c_{51} = 2^{r-4}$$

$$c_{61} = f(3,3r) - c_{65}$$

$$c_{71} = f(3,3r) - c_{75} = f(3,3r) - c_{65} + 2^{r-3}$$

$$c_{81} = f(3,3r) - c_{85}$$

$$c_{52} = c_{72} = f(3,3r) - c_{74} = f(3,3r) - 2^{r-1}$$

$$c_{82} = c_{72} - 2^{r-2} = f(3,3r) - 2^{r-1} - 2^{r-2}$$

$$c_{63} = c_{83} = c_{52} + 2^{r-2} = f(3,3r) - 2^{r-1} + 2^{r-2}$$

$$c_{62} = c_{52} - 2^{r-2} = f(3,3r) - 2^{r-1} - 2^{r-2}$$

$$c_{73} = c_{91} = f(3,3r) - 2^{r-1} + 2^{r-2} + 2^{r-3} = c_{83} + 2^{r-3}$$

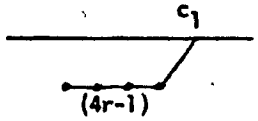
And from this tree we get,

$$f(4,4r) = f(3,3r) + f(2,2r-5)$$

$$= 7.2^{r-2} - 4.$$

The remaining tasks now are to analyze terminal configurations A,B,...,Z in the shown figure.

Node A : The configuration at this terminal node is :



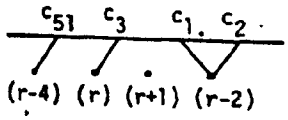
Here, we merge all 4 elements of A using 4r-1 comparisons. The length L_o of list B for this configuration is

$$L_o = f(4,4r-1) + 2^{r-2} + 2^{r-3}$$

$$= f(3,3r-1) + 2^{r-1} + 2^{r-3}$$

$$L_o = 1.8392.2^r + O(1).$$

Node B :

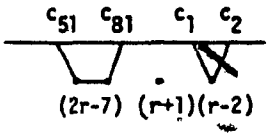


Here, the merging can be done independently for each element of A.

$$L_o = \min(c_0, 2^{r+1}) = c_0$$

$$L_o = 1.75.2^r + O(1).$$

Node C :



We have :

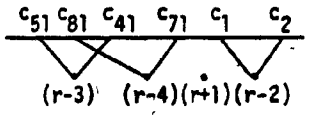
$$c_{81} - c_{51} = f(3,3r) - f(2,2r-3) - 2^{r-1} - 2^{r-2} - 2^{r-3} - 2^{r-4} - 2^{r-4}$$

Since $f(3,3r) = 2^r + f(2,2r-3) + f(2,2r-7) + 1$

$$c_{81} - c_{51} = f(2,2r-7) + 1.$$

Therefore, the first two elements of A can be merged in 2r-7 comparisons, and the length is also c_0.

Node D :

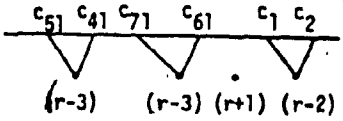


Here, we have

$$c_{71} - c_{81} = 2^{r-4}$$

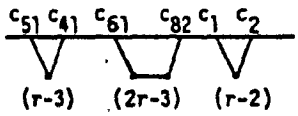
and $$c_{41} - c_{51} = f(3,3r) - f(3,3r-4) - 2^{r-1} - 2^{r-2} - 2^{r-4} - 1$$
$$= 0.1161.2^r + O(1)$$
$$< 2^{r-3} = 0.1250.2^r.$$

Node E :



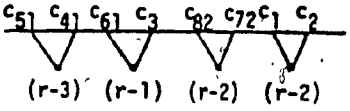
All elements of list A can be inserted independently. The length in this case is again c_0 .

Node F :



$c_{82} - c_{61} = f(2,2r-3)$. The length is also c_0 .

Node G :



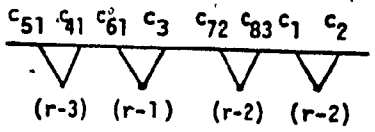
$$c_3 - c_{61} = f(3,3r-3)+f(3,3r)+f(2,2r-3)+2^{r-1}+2^{r-1}+2^{r-2}$$

$$= f(2,2r-5)+f(2,2r-9)-f(2,2r-7)+2^{r-2}$$

$$= 0.4107.2^r < 2^{r-1}$$

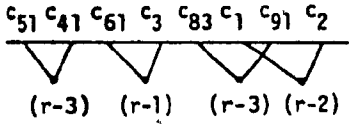
The length for this node is c_0.

Node H :



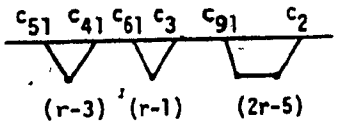
All elements are merged independently. The length is c_0.

Node I :



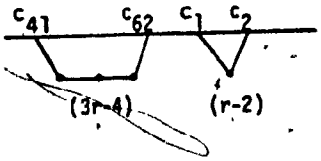
The total length is again c_0.

Node J :



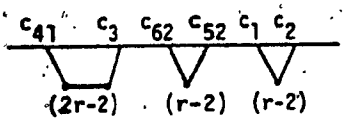
$c_2 - c_{91} = f(2,2r-5) + 1$. Again, we have $L_o = c_0$.

Node K :



$c_{62} - c_{41} = f(3,3r-4) + 1$; $L_o = c_0$

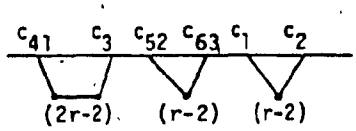
Node L :



$$c_3 - c_{41} = f(3,3r-3) - f(3,3r) + f(3,3r-4) + 2^{r-1} + 2^{r-2}$$
$$= 0.5893.2^r + O(1)$$

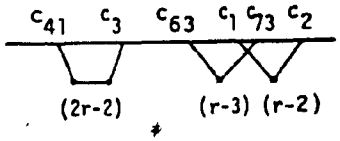
and $f(2,2r-2) = 0.6071.2^r + O(1)$, therefore the first two elements can be merged in 2r-2 comparisons. The last two elements can be merged independently. The length is c_0.

Node M :



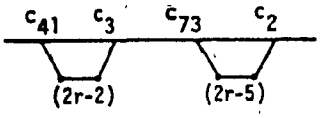
This is similar to node L. The length is again c_0.

Node N :



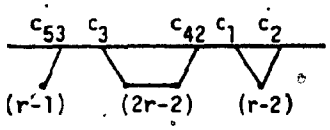
The merging process here is similar to that of node M.

Node O :



$c_2 - c_{73} = f(2,2r-5)+1$. The length is c_0

Node P :

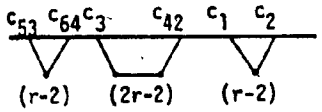


$$c_{42} - c_3 = f(3,3r-4) + 2^{r-1} + 2^{r-2} - f(3,3r-3)$$
$$= 0.589285.2^r + O(1)$$

$f(2,2r-2) = 0.6071.2^r + O(1)$. Therefore, the second and the

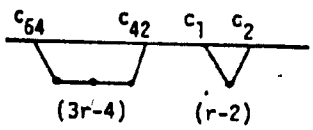
third elements can be merged together in 2r-2 comparisons. The first and the fourth are inserted independently. The length is c_0.

Node Q :



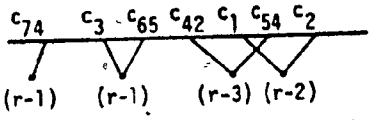
This is similar to node P, and $L_o = c_0$.

Node R :



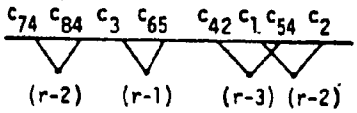
$c_{42} - c_{64} = f(3,3r-4) + 1$. The length is c_0.

Node S :



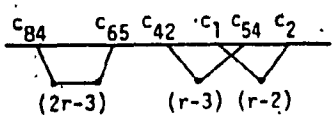
$$c_{65} - c_3 = f(2,2r-3) - f(3,3r-3) + 2^{r-1} + 2^{r-2}$$
$$= 0.42857.2^r + O(1) < 2^{r-1}$$

Node T :



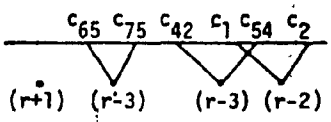
This is similar to Node S, and $L_o = c_0$

Node U :



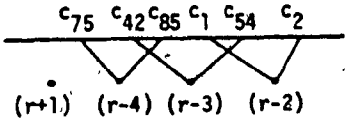
$c_{65} - c_{84} = f(2,2r-3)$; $L_0 = c_0$.

Node V :



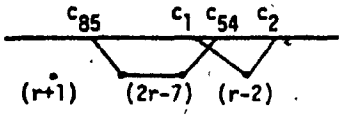
The merging process is obvious in this node. The length is c_0.

Node W :



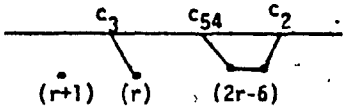
$$c_{54} - c_{42} = f(3,3r) - f(3,3r-4) = 2$$
$$= 0.116.2^r < 2^{r-3}.$$

Node X :



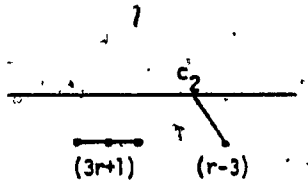
$c_{54} - c_{85} = f(2,2r-7) + 1$. The second and third elements can be merged in 2r-7 comparisons. The length is again c_0.

Node Y :



$c_2 - c_{54} = f(2,2r-5) + 2^{r-4} - 2^{r-3} + 1 = f(2,2r-6) + 1$. The length here is also c_0.

Node Z :



The length here is equal to $f(3,3r+1) = 1.9107*2^r + O(1)$

Thus we have shown that o configuration at a terminal node can have the length smaller than $f(3,3r) + f(2,2r-5)$ which has the asymptotic value $1.75.2^r + O(1)$. In fact except at nodes A and Z, we have $f(4,4r) = c_0$ as defined.

4.3.2. Merge tree(4,f(4r+1)) :

The algorithm is described by Figure 4.17.

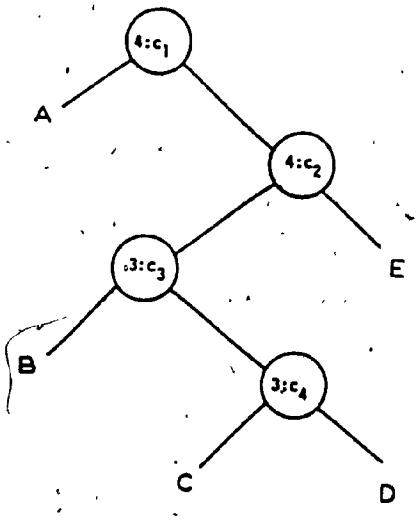


Figure 4.17

Notations used in figure 4.17 are defined as :

$c_0 = f(4,4r+1) = f(2,2r-5) + 2^{r-3} + 1$ for $r \geq 4$

$c_2 = c_0 - 2^{r-3} + 1$

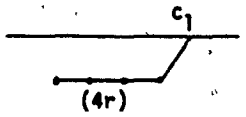
$c_1 = c_2 - 2^{r-2} + 2^{r-4}$

$c_4 = f(2,2r+1) + 1$

$c_3 = c_4 - 2^{r-2}$

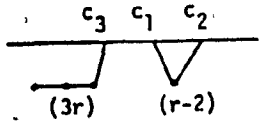
The analysis of the terminal nodes is as follows :

Node A :



$L_o = f(4,4r) + 2^{r-2} + 2^{r-3} - 2^{r-4} = 2.0625\ 2^r + O(1)$.

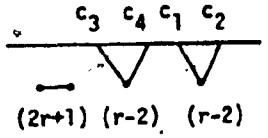
Node B :



$c_3 = f(2,2r+1) - 2^{r-2} + 1 = 1.4643\ 2^r + O(1)$

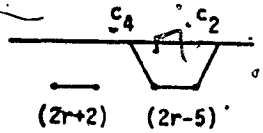
$f(3,3r) = 1.5357\ 2^r + O(1)$

Node C :



The third and fourth elements are inserted independently. The first two elements are merged to the section below c_4, and the length is c_0 .

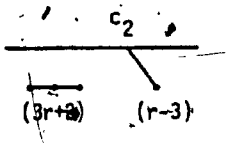
Node D :



$c_2 - c_4 = f(2,2r-5)+1$.

The length is $L_o = \min\{c_0, f(2,2r+2)\} = c_0$.

Node E :



$L_0 = f(3,3r+2) = 2.4285 + O(1)$.

Therefore, $f(4,4r+1) = c_0$, which has an asymptotic value of $1.7268.2^{k/4} + O(1)$, k= 4r+1.

4.3.3 Merge tree(4,f(4,4r+2)) :

The algorithm is given in figure 4.18. This case has been discussed in [19].

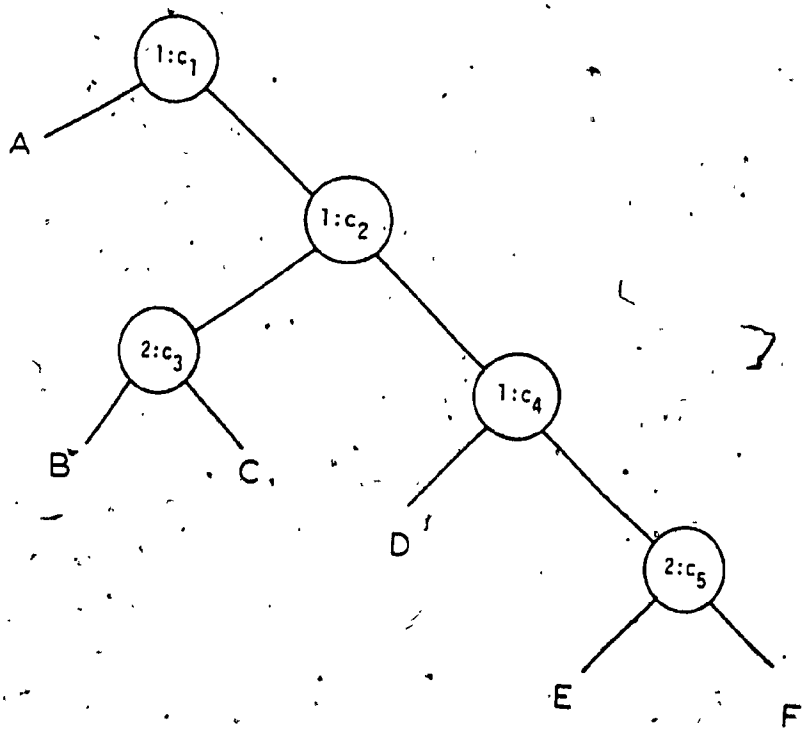


Figure 4

$c_0 = f(4r+2) = f(3,3r+2)$ for $r \geq 4$.

$c_1 = 2^{r-1}$ $c_2 = c_1 + 2^{r-1}$

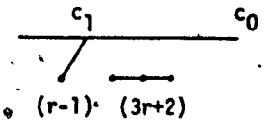
$c_3 = c_1 + f(2,2r-3) + 1$

$c_4 = c_2 + 2^{r-1}$

$c_5 = c_4 + f(2,2r-2) + 1$

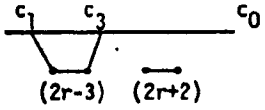
Analysis of figure 4.18 :

Node A :



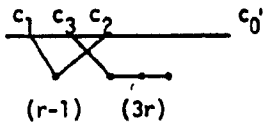
$L_o = f(3,3r+2) = 2.4286\ 2^r + O(1)$.

Node B :



$L_o = c_1 + f(2,2r+1) = 2.9286\ 2^r + O(1)$.

Node C :



Insert the first element, then merge the last three elements to the section larger than c_3.

$c_0 - c_3 = f(3,3r+2) - f(2,2r-3) - 2^{r-2} - 1 = 1.5\ 2^r + O(1)$

$f(3,3r) = 1.5357\ 2^r + O(1)$.

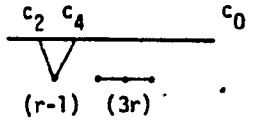
Therefore, the merging of the last three elements is possible.

$L_o = c_3 + f(3,3r)$

$$= 2^{r-1} + f(2,2r-3) + f(3,3r) + 1$$

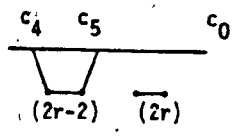
$$= 2.4637.2^r + O(1).$$

Node D :



$$L = c_2 + f(3,3r) = 2.5357.2^r + O(1)$$

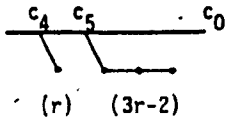
Node E :



$$L = c_4 + f(2,2r) = 2^r + 2^{r-1} + f(2,2r)$$

$$= 2.7143\ 2^r + O(1)$$

Node F :



$$c_0 - c_5 = f(3,3r+2) - 2^r - 2^{r-1} - f(2,2r-2) - 1$$

$$= 0.3214\ 2^r + O(1)$$

$$f(3,3r-2) = 0.9553\ 2^r + O(1)$$

Thus the last three elements can be merged successfully.

$$L = \min\{c_4+2^r,\ c_5+f(3,3r-2)\}$$

$$= c_4 + 2^r = 2.50\ 2^r$$

For this case, $f(4,4r+2) = f(3,3r+2)$, which has an asymtotic value of $1.7175.2^{k/4} + O(1)$; $k=4r+2$.

4.3.4 Merge tree(4,f(4,4r+3)) :

$c_0 = f(4,4r+3) = f(3,3r+2) + 2^{r-1}$ for r>4

$c_1 = 2^{r-1}$

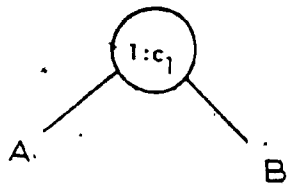
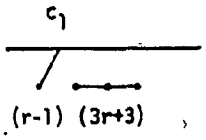


Figure 4.19

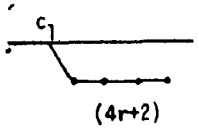
Analysis of figure 4.19 :

Node A :



$L = f(3,3r+3) = 3.074\ 2^r + O(1)$

Node B :



$L = c_1 + f(4,4r+2) = 2.9285 + O(1)$

Therefore, $f(4,4r+2) = c_1 + f(4,4r+2) = f(3,3r+2) + 2^{r-1}$ by using the result for the case k=4r+2. The asymtotic value for f(4,4r+3) is $1.7413\ 2^{k/4} + O(1)$ for k=4r+3. This concludes the analysis of the algorithms for merging 4 elements to n others elements.

This section has discussed the merge(4,n) problem. The

Top-Down method which was presented here is a new approach to develop the M(m,n) bound. The algorithms for all cases $k = 4r, 4r+1, 4r+2$ and $4r+3$ have asymptotic lengths varying from $1.7175\ 2^{k/4}$ to $1.75\ 2^{k/4}$ which agree with the conjecture in [10]. Also, Hwang's conjecture that $f(4,4r+2) = f(3,3r+2)$ (see [10]) has been proved in section 4.3.3.

The distinct feature this method differs from the heuristic approaches is that it is more systematic and algorithmic, and hence it lends itseft to a solution of the general merge(m,n) problem.

4.4 Results for m=5.

The merge(5,n) problem has been analyzed recently by Monting[21]. Monting's results for f(5,k) are summarized here :

$$f(5,5r) = \lfloor 118\ 2^r/224 \rfloor + \lfloor 96\ 2^r/224 \rfloor + 2\lfloor 136\ 2^r/224 \rfloor,$$
$$f(5,5r+1) = \lfloor 40\ 2^r/112 \rfloor + \lfloor 24\ 2^r/112 \rfloor + \lfloor 118\ 2^r/112 \rfloor + \lfloor 96\ 2^r/112 \rfloor,$$
$$f(5,5r+2) = \lfloor 59\ 2^r/112 \rfloor + \lfloor 48\ 2^r/112 \rfloor + \lfloor 136\ 2^r/112 \rfloor + \lfloor 76\ 2^r/112 \rfloor,$$
$$f(5,5r+3) = \lfloor 38\ 2^r/112 \rfloor + \lfloor 61\ 2^r/112 \rfloor + 2\lfloor 136\ 2^r/112 \rfloor,$$
$$f(5,5r+4) = \lfloor 94\ 2^r/112 \rfloor + \lfloor 61\ 2^r/112 \rfloor + 2\lfloor 136\ 2^r/112 \rfloor.$$

The optimality of these lengths is not known. However, Monting conjectured that these lengths are indeed optimal.

4.5 Average analysis of the M-optimal merge algorithms:

The M-optimal problems for merging (m,n) where $1 \leq m \leq 4$ are completely solved. However, the $\bar{M}$-optimal problem for these cases is not solved except the case m=1 (binary insertion). One reason that these problems are not of interest is that one does not want to solve a particular problem. But one does want to design algorithms for the general case rather than the particular cases of m=2,m=3,etc....The fractile algorithm described in section 3.5 is an example of the general case. Another reason is that the $\bar{M}$-optimal for the merge problem does not differ very much from the M-optimal one, for the same value of m. For example, if M(3,100) = 19 then $18 \leq \bar{M}(3,100) < 19$.

In this section, we shall develop the average analysis for the M-optimal merge(2,n) algorithms of Hwang and Lin[14]. We shall also analyse the average performance of the merge(3,n) algorithms given by Hwang[11]. Our objective is to produce the M-optimal bounds for the case (2,n) merging and to derive a lower bound as well as an upper bound for the $\bar{M}$-optimal bound of the merge (3,n) problem. These bounds may be helpful for designing an M-optimal algorithm which is not known at the present time.

4.5.1 Case $m=2$:

In [14], Hwang and Lin gave an M-optimal algorithm to solve tthe (2,n) problem. The M-optimal bounds for merging(2,n) are given in the equations (4.2) and (4.3).

At the point $n=f(2,k)$, the merge (2,n) merge algorithm is very efficient in the sense that it is both M-optimal and $\bar{M}$-optimal. Moreover, the information theoretical lower bound $\overline{ITM}(2,n)$ is achieved.(see [14])

At these points, $f(2,k)$, the M-optimal algorithm is represented by a merge tree as given in [14]. Next, we will analyse the merge tree for the points others than $f(2,k)$, $k \in N$.

The average comparisons of a merge algorithm is simply the average path length of the corresponding merge tree, which is the external path length divided by the total number of outcomes which is fixed for a fixed n. To minimize the number of comparisons of the average case is equivalent to minimize the external path length. Among the merge trees with the same number of leaves(outcomes), the one is more balanced,i.e., the difference between the number of outcomes of the left subtree and that of the right subtree is smallest, will have shorter external path length. When a merge tree achieves the minimum(theoretic) external path length (see Lemma 2.3 of section 2.2), every comparison in the tree must satisfy the following criteria which we called the balanced criteria :

A comparison(internal node) c in a merge tree is called to satisfy the balanced criteria if

$$(4.17) \quad 2^{\lfloor \log n \rfloor - 1} \leqslant n_\ell \leqslant 2^{\lfloor \log n \rfloor}$$

$$(4.18) \quad 2^{\lfloor \log n \rfloor - 1} \leqslant n_r \leqslant 2^{\lfloor \log n \rfloor}$$

where n : the number of outcomes before making the comparison c.

n_ℓ : the number of outcomes corresding to the left subtree.

n_r : the number of outcomes corresponding to the right subtree.

Given a size t of list B, $f(2,k) < t < f(2,k+1)$, it is not always true that we can find a comparison which splits the merge tree and satisfies the balanced criteria. A counter example will illustrate this :

Example : Merge(2,4) problem.

Comparison	n	n_ℓ	n_d
$A_2:B_1$	15	1	14
$A_2:B_2$	15	3	12
$A_2:B_3$	15	6	9
$A_2:B_4$	15	10	5

The balanced criteria are satisfied if and only if $4 \leqslant n_\ell, n_r \leqslant 8$. Hence, the balanced criteria are violated by all comparisons in this example. Therefore, the $\overline{\text{ITM}}(2,4)$ cannot be achieved.

Now, given a merge(2,t) problem, the merge tree whose the first comparison in (2:x) is given in figure 4.19

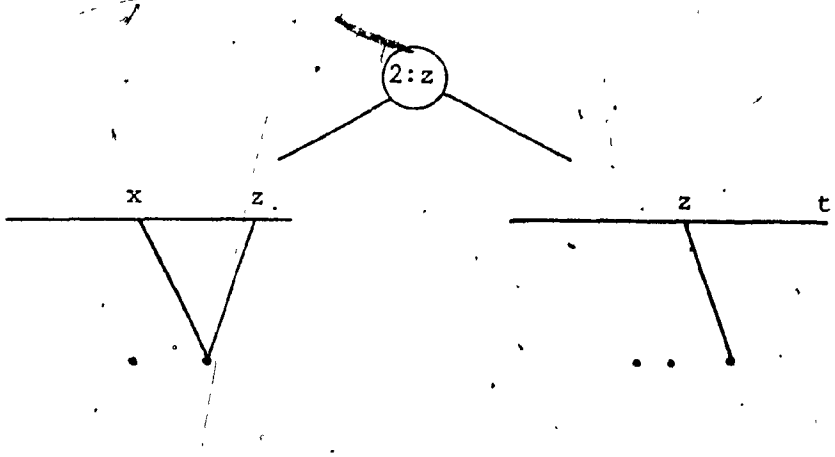


Figure 4.19

The external path length for this tree is

$$epl(C(t)) \leqslant epl(C(x-1)) + epl(H(x,t)) + \binom{t+2}{2}$$

where C(t) is the problem of merge(2,t),

and H(x,t) is the merge(2,t) problem in which $(A_2 > B_x)$ is known.

Therefore, the external path length for the merge tree(2,t) can be given by :

$$epl(C(t)) = \min_{1 \leqslant x \leqslant t} \{epl(C(x-1)) + epl(H(x,t)) + \binom{t+2}{2}\}$$

$$(4.19) \qquad = \min_{1 \leqslant x \leqslant t} \{epl(C(x-1)) + epl(H(x,t))\} + \binom{t+2}{2}$$

However, it is not necessary to consider the exhaustive search for the whole length (1:t). We need only to consider the case sucth that the M-optimality of the merge tree is satisfied.

Since t lies between f(2,k-1) and f(2,k), the merge tree has to be adapted to the merge tree of either

(2,f(2,k-1)) or (2,f(2,k)). The first comparison in the merge tree (2,f(2,k-1)) is $(A_2:B_{f(2,k-2)+1})$ and in the merge tree (2,f(2,k)) is $(A_2:B_{f(2,k-1)+1})$. So, the first comparison of the merge tree (2,t) should be (2:x) where

$$f(2,k-2)+1 \leqslant x \leqslant f(2,k-1)+1$$

Beyond these limits, either the left or the right subtree is too large (its depth exceeds the M-optimal bound, hence its external path length is large). These limits can be used to reduce the computations in equation (4.19).

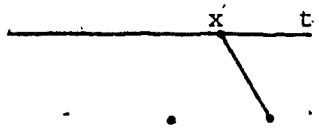
Equation (4.19) can now be rewritten as :

(4.20) $$epl(C(t)) = \min\{epl(C(x-1))+epl(H(x,t))\} + \binom{t+2}{2}$$

where $f(2,k-2)+1 \leqslant t \leqslant f(2,k-1)+1$,

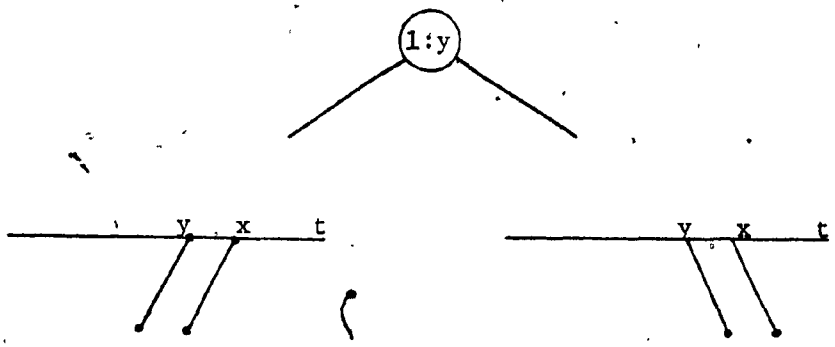
and k satisfies $f(2,k-1) < t < f(2,k)$.

Now, we consider the optimal external path length for the configuration H(x,t) which represents the merge(2,t) problem in which we know $A_2 > B_x$.



We have two choices :

Either we compare (1:y), 1<y<x+1 :



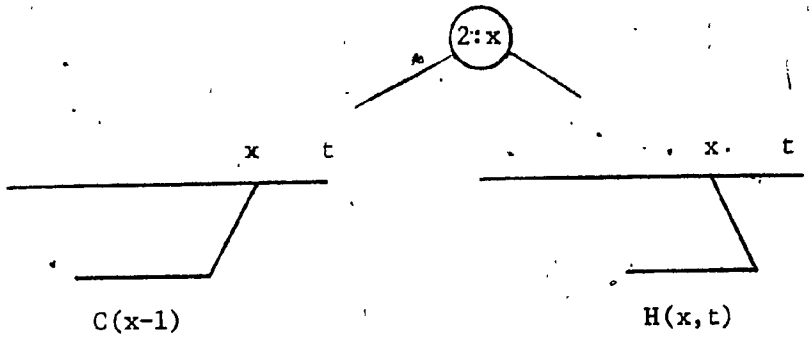
(4.21) $epl(H(x,t)) = \min\{epl(Ind(y-1,t-x)+epl(H(t-y,t-x))\}$

where

$Ind(y-1,t-x)$ = work to be done by 2 independent tasks

insert(1,y-1) + insert(1,t-x).

or we compare (2:z), $x \leqslant z \leqslant t$:



(4.22) $epl(H(x,t)) \leqslant \min_{x \leqslant z \leqslant t}\{epl(H(z,t))+epl(H(x,z))\}$

Combining (4.21) and (4.22), we get :

(4.23) epl(H(x,t)) = min(formula(4.21),formula(4.22)).

From equations (4.20) and (4.23), an algorithm can be written to compute the M(2,n) bounds as follows :

```
Algorithm Compute(2,t)
    If t=f(2,k), k∈N  then
       epl := (t+2 over 2)*TM(2,t)
    else
      Ilow ← f(2,k-2); Iup ← f(2,k-1);
      ( f(2,k-1)<t<f(2,k) )
      For x ← Ilow  to  Iup do :
       If  epl<Compute(2,x-1)+H(x,t)   then
          epl ← Compute(2,x-1)+H(x,t)
       endif;
      M(2,t) = epl/(t+2 over 2)
    End.
    Procedure H(x,t)
     epl ← ∞
     For i ← 1 to x do
       If epl<Ind(i-1,x)+H(t-i,x-i) then
         epl ← Ind(i-1,x)+H(t-i,x-i)
     For z ← x to t do
       If epl<H(x,z)+H(z,t) then
         epl ← H(x,t)+H(z,t)
     Return epl;
    End.
    Ind(x,y) = (x+1)*epl(y) + (y+1)*epl(x)
                    (see Lemma 4.4)
```

We have written programs to analyse all these cases for n=1 to 100. The results are given in table 4.1.

Table 4.1 Average-case optimal bound M(2,n); $1 \le n \le 100$

n	M(2,n)	n	M(2,n)
1	1.6667	51	10.5138
2	2.6667	52	10.5688
3	3.4000	53	10.6209
4	4.0000	54	10.6701
5	4.4762	55	10.7168
6	4.8929	56	10.7610
7	5.2500	57	10.8030
8	5.5778	58	10.8429
9	5.8545	59	10.8809
10	6.1061	60	10.9170
11	6.3590	61	10.9514
12	6.5934	62	10.9886
13	6.7810	63	11.0365
14	6.9500	64	11.0904
15	7.1397	65	11.1474
16	7.3268	66	11.2019
17	7.5029	67	11.2540
18	7.6526	68	11.3039
19	7.7810	69	11.3517
20	7.8918	70	11.3975
21	8.0158	71	11.4414
22	8.1558	72	11.4835
23	8.2933	73	11.5240
24	8.4246	74	11.5628
25	8.5413	75	11.6001
26	8.6455	76	11.6360
27	8.7389	77	11.6706
28	8.8230	78	11.7038
29	8.8989	79	11.7358
30	8.9819	80	11.7666
31	9.0739	81	11.7964
32	9.1747	82	11.8250
33	9.2790	83	11.8527
34	9.3746	84	11.8793
35	9.4625	85	11.9051
36	9.5434	86	11.9300
37	9.6181	87	11.9540
38	9.6872	88	11.9783
39	9.7512	89	12.0095
40	9.8107	90	12.0447
41	9.8660	91	12.0851
42	9.9175	92	12.1258
43	9.9729	93	12.1653
44	10.0329	94	12.2035
45	10.1055	95	12.2405
46	10.1844	96	12.2765
47	10.2585	97	12.3113
48	10.3282	98	12.3451
49	10.3937	99	12.3778
50	10.4555	100	12.4096

4.5.2 Case m=3 :

We continue the average-case analysis for the merge(3,n) problem by considering the merge trees given in [11]. We refer the reader to this reference for the trees.

Whereas the merge tree(2,f(2,k)) trees obtain the theoretical lower bounds $\overline{ITM}(2,f(2,k))$; ($k \in N$) the merge (3,f(3,k)) trees, unfortunately, do not. Therefore, first step is to compute the average number of comparisons of the merge trees(3,n) where n=f(3,k).

Four subcases are considered :

a) k=3r :

The merge tree for this case is given in figure 4.20. This is actually figure 2 of [11], but adapted to our conventions for consistency. For this case, the merge tree given in [11] does not achieve the $\overline{ITM}(3,n)$ bound. This can be easily verified by considering the terminal nodes B and C in which the task of merge($2,c_2-c_1-1$) is done independently (see Lemma 4.4) where $c_2-c_1-1 \neq f(2,2r)$ with any $r \in N$.

If we use the strategy of computing the external path length for all possible comparisons, then the computations grow exponentially. Therefore, instead of computing the optimal average bounds by examining all possible outcomes, we compute the upper bound and the lower bound of the merge tree as follows:

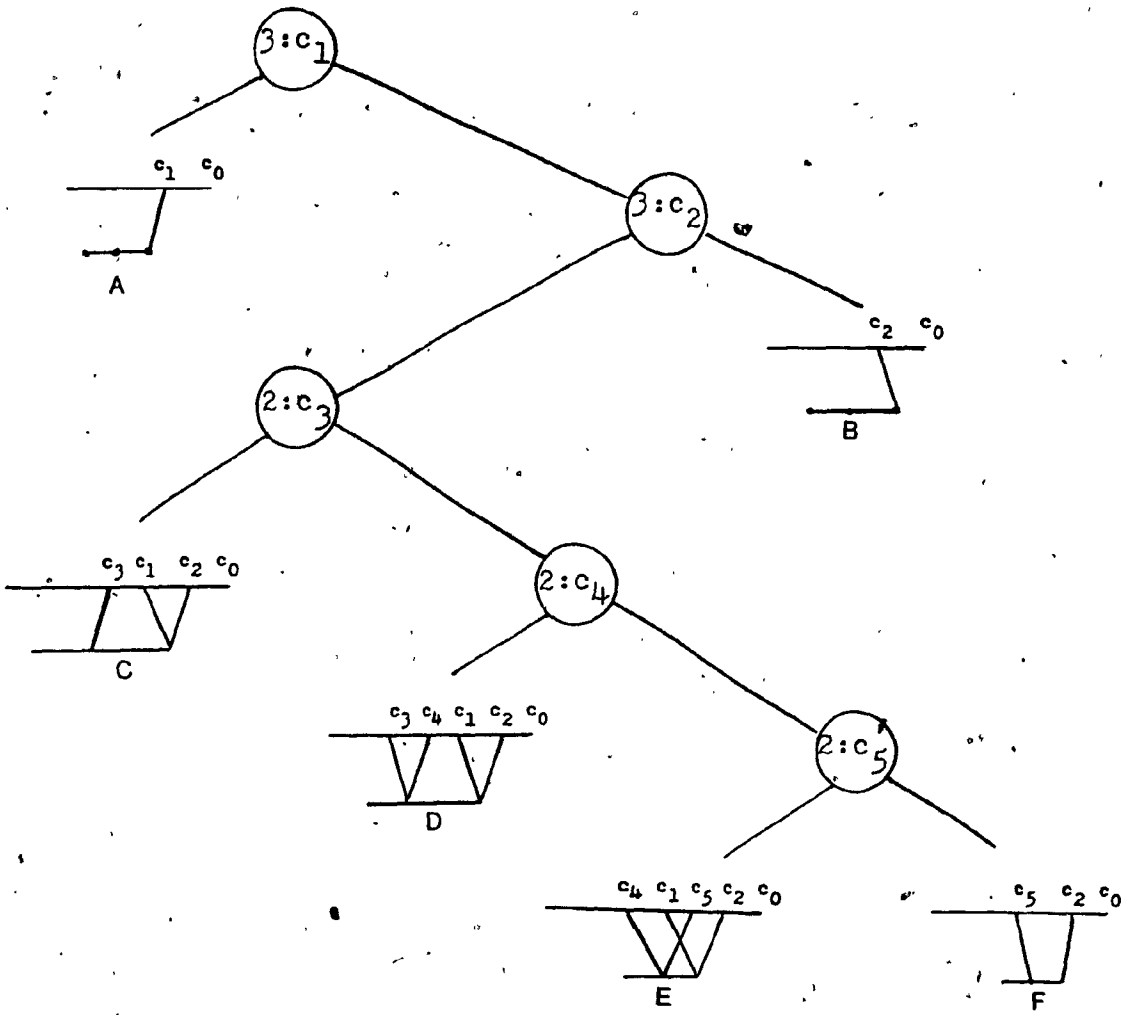

3:c1
c1 c0
A
3:c2
c2 c0
B
2:c3
c3 c1 c2 c0
C
2:c4
c3 c4 c1 c2 c0
D
2:c5
c4 c1 c5 c2 c0
E
c5 c2 c0
F

Figure 4.20

At nodes where the problem is reduced to the previous cases (merge(1,n), merge(2,n),etc..) and the tasks can be performed independently, get the external path length from the previous cases and by means of Lemma 4.4. This class of nodes consists of nodes B,C,F,G,H and I.

As for other nodes, the external path length is not computed directly. So, we compute the lower bound and the upper bound for the external path length(epl) as follows :

- Lower bound is the theoretical lower bound of the epl.
- Upper bound is the epl of the worst-case optimal merge tree (which is the product of the worst-case comparisons and the number of outcomes).

This class of nodes consists of D,E,J,K.

Node A (left subtree) is symmetrical to the right subtree, hence it is not necessary to analyse A.

Hence, for the whole tree, we have computed the external path length for every node; this completes the computation and produces the lower bound as well as the upper bound for the external path length.

b) k=3r+1 :

The merge tree for this case is given in figure 4.21 (see figure 3 of [11])

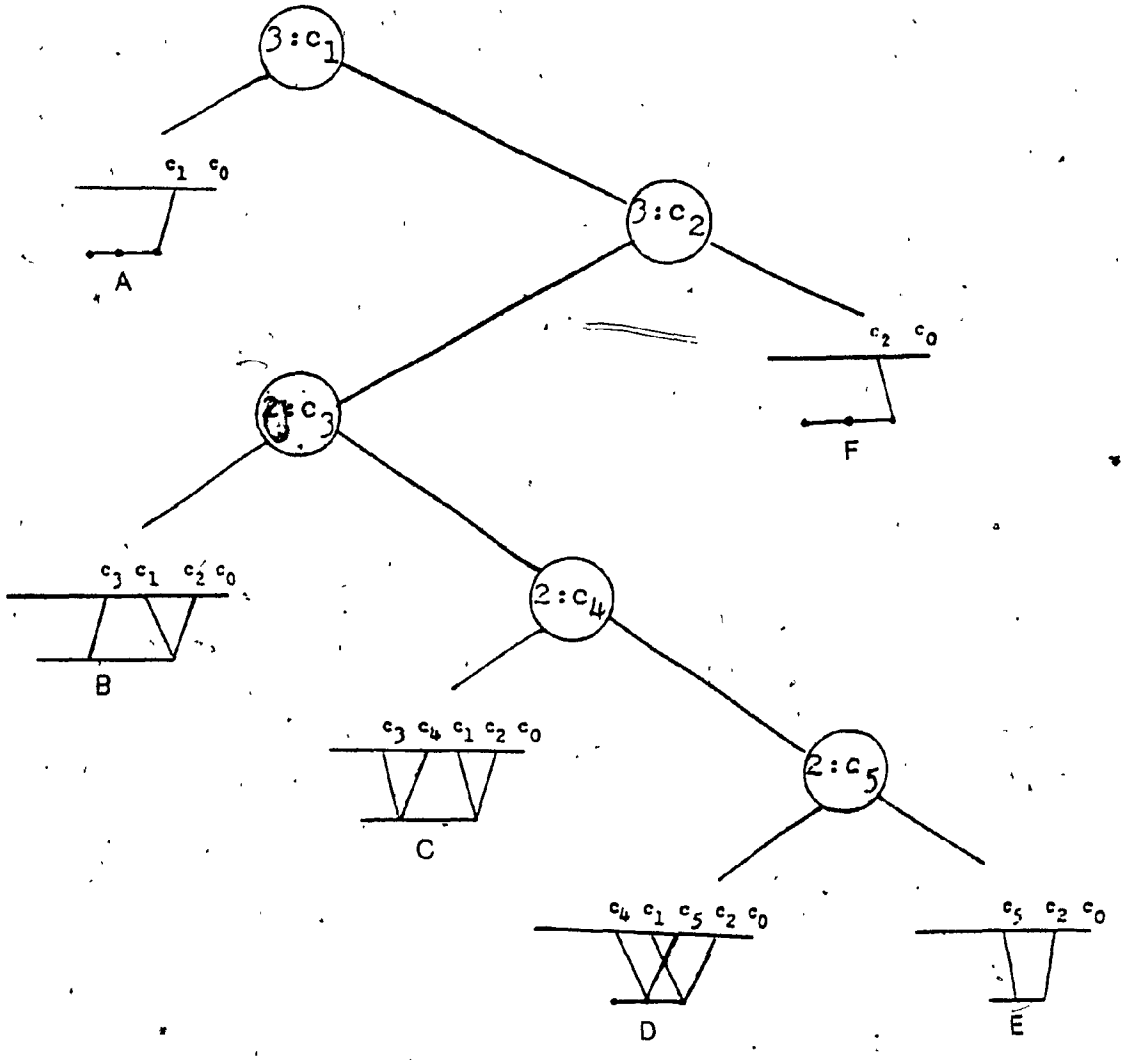

3:c1
c1 c0
A
3:c2
c2 c0
F
2:c3
c3 c1 c2 c0
B
2:c4
c3 c4 c1 c2 c0
C
2:c5
c4 c1 c5 c2 c0
D
c5 c2 c0
E

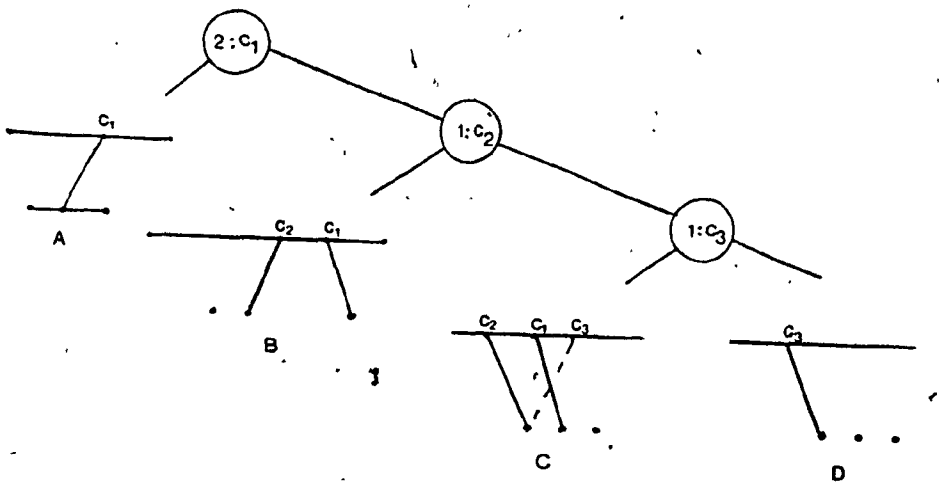
Figure 4.21

Node A can be analyzed as in case (a).

Nodes B and C can be computed by means of lemma 4.4 and previous results, since the tasks to be done in these nodes are independent. Nodes D,E and F are computed similarly to the nodes D,E,J,K in case (a).

c) k=3r-2:

The merge tree for this case is given in figure 4.22 (see figure 4 of [11])



The computations of the terminal nodes are given below

The left branch (node A) of the merge tree is symmetrical to the right branch. Therefore, we discuss only the computations for the right branch of this merge tree.

Node B is computed by means of Lemma 4.4 and previous results.

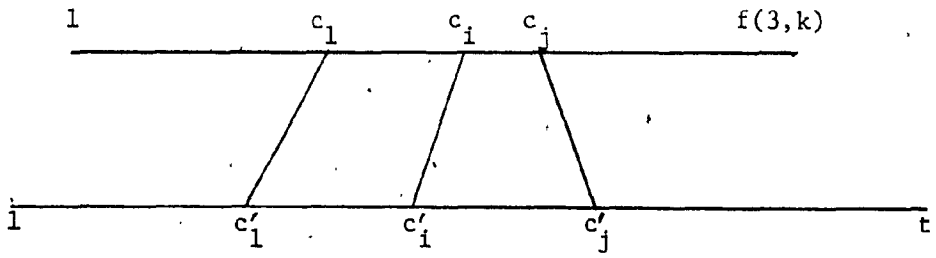
Node C is computed similarly to D,,E,J,K in case (a).

Node D is computed recursively.

d) $f(3,k) < t < f(3,k+1)$ for some k.

As in the merge(2,t) tree, if t lies between f(3,k) and f(3,k+1), the merge tree should be adapted to either the merge tree (3,f93,k)) or (3,f(3,k+1)).

To maintain the tree as balanced as possible, we use the concept of "proportionality" as described below::



We use the merge tree (3,f(3,x)) ,where x=k or x=k+1, for (3,t) in which the points c_i in the merge tree (3,f(3,x)) is replaced by $c'_i = c_i * r$ where r is a real constant ratio computed by :

$r = t/f(3,k)$ if we use the merge tree (3,f(3,k)), and

$r = f(3,k+1)/t$ if we use the merge tree f(3,f(3,k+1)).

Finally, the minimum of these bounds is taken.

Here, we only give the brief sketch for the computation of the average number of comparisons for the merge tree(3,n). We have written programs to analyse all these caes for n=1 to 100. The results are given in Table 4.2.

Table 4.2 Lower bound and upper bound for M(3,n), $1 \leq n \leq 100$

n	lower bound	upper bound	n	lower bound	upper bound
1	2.000	2.000	51	15.273	15.425
2	3.400	3.400	52	15.360	15.513
3	4.577	4.577	53	15.442	15.597
4	5.385	5.385	54	15.476	15.630
5	6.104	6.104	55	15.541	15.696
6	6.748	6.748	56	15.617	15.773
7	7.220	7.220	57	15.724	15.881
8	7.754	7.754	58	15.797	15.955
9	8.152	8.234	59	15.910	16.069
10	8.549	8.635	60	15.992	16.152
11	8.950	9.039	61	16.065	16.225
12	9.247	9.340	62	16.141	16.302
13	9.545	9.641	63	16.197	16.359
14	9.895	9.994	64	16.284	16.446
15	10.147	10.249	65	16.331	16.494
16	10.349	10.452	66	16.383	16.547
17	10.622	10.728	67	16.435	16.599
18	10.889	10.998	68	16.493	16.658
19	11.115	11.226	69	16.537	16.703
20	11.298	11.411	70	16.605	16.771
21	11.441	11.555	71	16.656	16.823
22	11.686	11.803	72	16.729	16.896
23	11.904	12.023	73	16.794	16.961
24	12.088	12.208	74	16.875	17.044
25	12.237	12.359	75	16.929	17.098
26	12.381	12.505	76	17.004	17.174
27	12.483	12.608	77	17.098	17.269
28	12.689	12.815	78	17.147	17.318
29	12.862	12.991	79	17.203	17.375
30	13.013	13.143	80	17.242	17.414
31	13.152	13.283	81	17.312	17.485
32	13.263	13.395	82	17.349	17.522
33	13.380	13.514	83	17.418	17.593
34	13.464	13.598	84	17.469	17.644
35	13.604	13.740	85	17.482	17.657
36	13.745	13.882	86	17.526	17.701
37	13.900	14.039	87	17.560	17.736
38	14.006	14.146	88	17.610	17.786
39	14.121	14.262	89	17.657	17.833
40	14.220	14.362	90	17.697	17.874
41	14.331	14.475	91	17.746	17.923
42	14.422	14.566	92	17.818	17.996
43	14.487	14.632	93	17.886	18.064
44	14.577	14.723	94	17.931	18.110
45	14.701	14.848	95	17.969	18.148
46	14.820	14.968	96	18.028	18.208
47	14.904	15.053	97	18.097	18.278
48	15.007	15.157	98	18.160	18.342
49	15.098	15.249	99	18.204	18.386
50	15.213	15.365	100	18.231	18.413

In this chapter, we have discussed the problems of merging. These problems are intimately related to the sort problems. Improvements on the bounds of one problem may entail the improvement of the other. In the next section, we present the sort algorithms which show that the improvement of the merge bound entails the improvement of the sort bound.

CHAPTER 5

SORT ALGORITHMS

In an article entitled " A Tournament Problem " which appeared in the American Mathematical Monthly [09], Ford and Johnson gave an algorithm for sorting t elements according to some transitive characteristics, by means of successive pairwise comparisons. The Ford and Johnson algorithm (fja) comes remarkably close to the information theoretical lower bound ITS(t) for infinitely many special values of t and is never far away. Hwang and Lin [15] showed, in this connection, that $S_{fja}^{*}(t) = t.\log(t) - k(t).t + O(\log(t))$, where k(t) is almost a constant satisfying $1.329 < k(t) < 1.415$. Since $ITS(t) = t.\log(t) - 1.44t + O(\log(t))$, it follows that a better algorithm than the fja can have little margin for improvement. In addition, it was known that $S(12) = S_{fja}(12) = 30$ whereas $ITS(12) = 29$, and it has been suspected for a long time that for many values of t (perhaps for all except a set of measure 0) S(t) may be larger than ITS(t). This suspicion leads to the prospect that the

---

* Note that we use similar definition for $S_u(t)$ as follow

$S_u(t)$ = max. number of comps. to sort t items using algorithm u.

$S(t) = \min\{S_u(t)$, min. is taken over all algorithms.

margin for possible improvement of the fja may be even less than the meager margin indicated by $S_{fja}(t) - ITS(t)$. For these reasons, it has been uncertain for more than 20 years whether the fja is optimal. To answer to these conjectures, Manacher[20] gave a constructive proof that this is not the case, and provided an algorithm which surpasses the fja for infinitely many t, $t \geq 189$. The set A of the values at which the fja is surpassed by Manacher's algorithms has a measure of index 1/8 with respect to N, the set of the natural integers, in the sense that $m(A)/m(N) = 1/8$, here m denotes the counting measure.

Another sorting algorithm which also surpasses the fja is that of Monting. He made use of the efficient optimal algorithm for merging(5,n). The set A of the values at which the fja is surpassed by the Monting algorithms has a measure of index 4.27/8 with respect to N.

In this chapter, we develop a hybrid method for sorting which significantly improves the S-optimal bound of sorting, and we also show that the set of values t at which the fja is not optimal has a measure of index 7/8. We believe, although we have not been able to prove, that our algorithm in this chapter is S-optimal, at least in the class of hybrid methods using merge algorithms for sorting. No attempt is made to analyze the average behaviour of this algorithm.

Let us first review the famous Ford-Johnson algorithm, Manacher's algorithm and Monting's algorithm.

5.1 The Ford-Johnson algorithm (fja).

To sort t items, the fja can be expressed inductively in three steps :

Algorithm Ford-Johnson(fja) :

Step 1. Group the t elements in $\lfloor t/2 \rfloor$ pairs and make pairwise comparisons. Leave one element out if t is odd.

Step 2. Sort the $\lfloor t/2 \rfloor$ larger elements found in step 1 by the fja inductively.

Step 3. This is the last step and also crucial step. The third step is best explained by a diagram. We adopt the following convention: two elements x and y are connected by a line when their relative ordering is known. If $x<y$ then x is shown to the left of y, otherwise x is shown to the right of y. Using this convention, we have at the end of step 2 the following diagram on figure 5.1 :

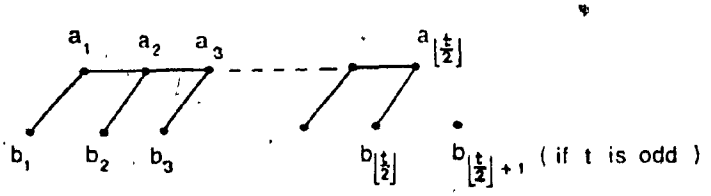


Figure 5.1

In this step, we insert the B's in sequences into the main chain $(B_1, A_1, A_2, \ldots, A_{\lfloor t/2 \rfloor})$ which is already sorted. The sequences to be inserted are (B_3, B_2), (B_5, B_4), $(B_{11}, B_{10}, \ldots, B_6), \ldots, (B_{t_k}, B_{t_k-1}, \ldots, B_{t_{k-1}+1}), \ldots$

The last sequence of B's may not be a complete set as defined.

We now give a brief analysis of the fja (see [15] for further details). Following Knuth, we define the sequence $(t_1, t_2, t_3, \ldots) = (1, 3, 5, \ldots)$ suth that each element of the sequence $(B_{t_k}, B_{t_k-1}, \ldots)$ can be inserted into the main chain in at most k comparisons. Knuth[15] has shown that :

$$t_k = \frac{2^{k+1} + (-1)^k}{3} \tag{5.1}$$

It is obvious that

$$S_{fja}(t) = \lfloor t/2 \rfloor + S_{fja}(\lfloor t/2 \rfloor) + G(\lceil t/2 \rceil) \tag{5.2}$$

where $G(\lceil t/2 \rceil)$ is the number of comparisons in step 3.

If $t_{k-1} \leq m \leq t_k$, then :

$$G(m) = \sum_{j=1}^{k} j(t_j - t_{j-1}) + k(m - t_{k-1})$$

$$= km - w_k \tag{5.3}$$

where

$$w_k = \sum_{i=1}^{k-1} t_i = \left\lfloor \frac{2^{k+1}}{3} \right\rfloor . \tag{5.4}$$

The closed form solution for $S_{fja}(t)$ was obtained by Knuth[15] :

$$S_{fja}(t) = t\lceil \log(3/4)t \rceil - \lfloor (1/3)2^{\lfloor \log 6t \rfloor} \rfloor + \lfloor (1/2)\log 6t \rfloor \tag{5.5}$$

Following Knuth[15], we make the following substitution :

(5.6) $$t = (4/3).2^{k+f}$$

where k is an integer and f is a real number : $0 \leqslant f < 1$; then $S_{fja}(t)$ can be rewritten as :

(5.7) $$S_{fja}(t) = t\log t - t(1+f+2^{1-f}-\log 3) + \tfrac{1}{2}\log t + O(1).$$

The coefficient of t :

(5.8) $$c(t) = -(1+f+2^{1-f}-\log 3)$$

is approximately to - 1.415 when f=0 or f=1 and raises to its peak at approximately -1.329 when $f = 1+\log(\ln 2) = 0.471$. In figure 5.2 we draw the function $S_{fja}(t)$ versus t, the minimum local points are the points $t = u_k = \lfloor (4/3)2^k \rfloor$ and the maximum local points are at $t = 2^k$.

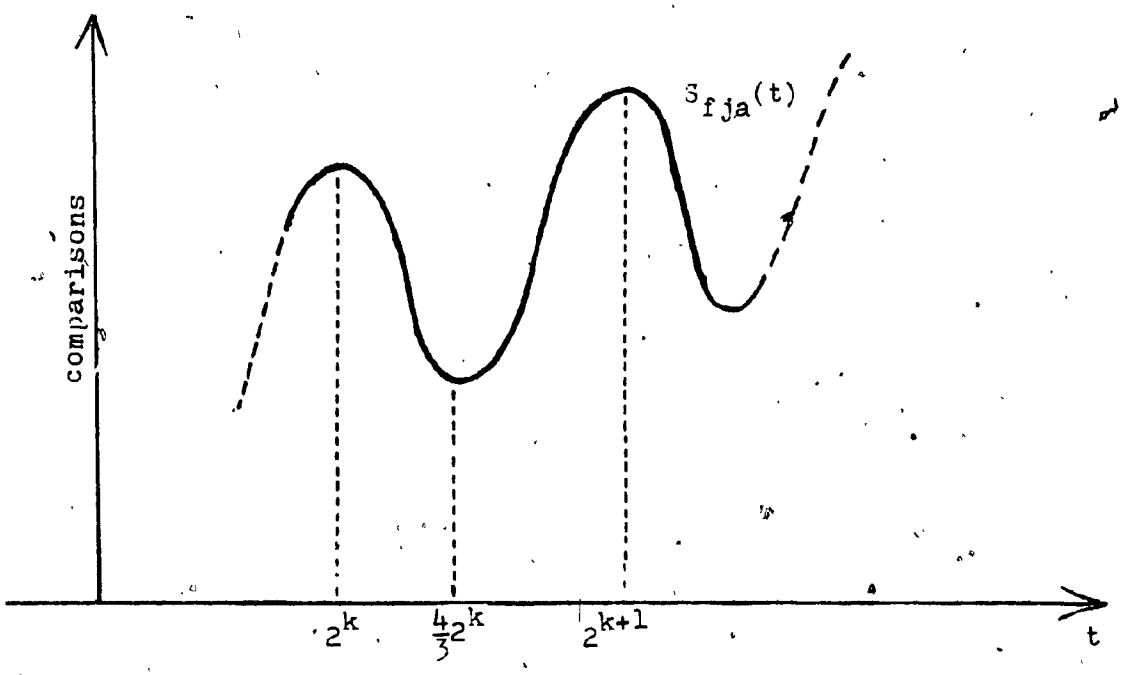


Figure 5.2

At the local minimum points $t = u_k = \lfloor (4/3).2^k \rfloor$, c(t) is remarkably close to the absolute lower bound -1.4427 (see

equation (2.7) of chapter 2), hence $S_{fja}(t)$ is close to the information theoretic lower bound ITS(t). Therefore, we can base on the assumption that the fja is very close to being optimal at local minima. Hence, we also sort t items by sorting subsets of t corresponding to these local minima, we then merge by an efficient algorithm.

Use this technique to sort, we can surpass the fja at other values of t, especially at the local maximum points.

5.2 Manacher's method.

Manacher makes use of the property that c(t) achieves the local minima value -1.415 at the points $u_k = \lfloor (4/3).2^k \rfloor$. He uses the fja to sort u_k and u_j seperately and merges them by his modified binary merge algorithm. Manacher's merging algorithm is based on Hwang-Lin's optimal algorithm of 2 elements to n [14], and some special cases for merging 3 and 4 elements to n. For $n/m \geq 8$, Manacher's merging algorithm requires $31/336m + O(1)$ less comparisons than the binary merge algorithm. Using this strategy, Manacher's algorithm to sort t items, where $t = \lfloor (4/3).2^k \rfloor$ and $0 < f < \log(9/8) \leq 0.17$ is as follows :

Algorithm manacher sort(ms) :

Step 1. Split t = n+m where n is the largest value of $u_k = \lfloor (4/3)2^k \rfloor$ which is smaller than t.

Step 2. Sort n and m elements by the fja.

Step 3. If (m,n) are special (either list has 1,2 or 3 elements) then use the appropriate optimal merging algorithm and stop.

Step 4. If $n \leq 7m$, perform one iteration of the binary merge algorithm
else go to step 5.

Step 5. Perform one iteration of Manacher's merge algorithm[19].

Step 6. If one list is exhausted then stop,
else go to step 3.

Manacher's algorithm is better than Hwang-Lin's binary merge for $n/m \geq 8$. It gets worse when n becomes smaller than 8m. For this reason, Manacher's algorithm beats the fja only for values of t which could be splitted into n+m where $n = u_k$ and m is near u_{k-3}. This fact actually restricts the lowest value of t for which the fja can be beaten. It also restricts the size of the bands of successive values of t for which $S(t) < S_{fja}(t)$. Within an interval $I_k = [2^k, 2^{k+1}]$, the values of t for which $S(t) < S_{fja}(t)$ are at most in the range $A_k = [2^k, 2^{k+1}]$, hence if we permit the measure of this set, at most we get $m(A_k)/m(I_k) = 1/8$; or $m(A)/m(N) = 1/8$. The constructive proof of Manacher shows that the set of values of t at which the fja is not optimal is of index 1/8.

5.3 Monting's algorithm.

Whereas Manacher used the nearly constant quotient of m/n, namely n=8m, to split the set of size t=m+n into two subsets of sizes m and n, Monting used another approach. His method consists of choosing a fixed size for the shorter list m rather than a constant quotient for n/m. This fixed size is equal to 5, i.e. m=5, since he could derive an efficient algorithm (which he believed to be optimal) for merging (5,n). (He also proved that the fixed size m=3 or m=4 are not good enough to beeat the Ford-Johnson algorithm).

The way he sorts t elements consists of choosing a largest size u_k such that $u_k < t < u_{k+1}$ and sort u_k elements. Next, he repeated the process of picking 5 elements, sort them and merge into the sorted list. This process is repeated in a finite number of steps in such a way that the number of comparisons is gained after each step. This is done until no further gain is possible, for the elements left over in the sorted list, he used the binary insertion to insert these elements, one by one. And he reused the process of sorting 5 elements, merging into the sorted list. His algorithm to sort t elements can be described as follows:

Algorithm monting sort(som) :

For $u_k+5 \leq t \leq 2^{k+1} + 5\left\lfloor \frac{1}{2}c_n \right\rfloor$.

Step 1. Sort u_k elements by the fja; $t_0 \leftarrow u_k$

Step 2. Compute $c_n = \left\lfloor \frac{1}{5}\left(\left\lfloor \frac{9.2^{k-4}}{7} \right\rfloor + \left\lfloor \frac{2^{k-5}}{3} \right\rfloor + 1 \right) \right\rfloor$

Step 3. If $t-t_0 < 5$; go to step 7
else sort 5 elements; merge (5,t) elements;
$t_0 \leftarrow t_0+5$.

Step 4. Repeat step 3 c_n times.

Step 5. Insert $(2^{k+1}-t_0)$ elements into the sorted list one by one, using the binary insertion;
$t_0 \leftarrow 2^{k+1}$.

Step 6. Repeat step 3 $\lfloor \frac{1}{2}c_n \rfloor$ times.

Step 7. Merge $(t-t_0, t_0)$, use the appropriate merge algorithm for $t-t_0 = 1,2,3$ or 4.

The range for which the fja is surpassed by Monting's algorithm is larger than that of Manacher's algorithm. However, the fact that he used the fixed size for the shorter list makes the gain lower and therefore restricts the size of the bands of successive values of t for which $S(t) < S_{fja}(t)$. Within an interval $I_k = [u_k, u_{k+1}]$, the maximum and minimum values of t at which Monting's algorithm surpasses the fja are :

(5.9) $m \sim 2^k(1 + 61/1344) \sim u_k(1 + 957/1792)$

(5.10) $m' \sim u_k + 5.$

Thus, this "band" is in the range $B_k = [u_k,(1+957/1792)u_k]$, hence if we permit the measure of this set, we get $m(B_k)/m(I_k) = 957/1792 = 4.272/8$.

In constrast to Manacher's annd Monting's method, our approach uses neither fixed ratio nor fixed length. Fisrt, we make use of the Forward-Testing-Backward-Insertion algorithm, which is, as stated in chapter three, the best known algorithm in the worst case. Next, the length of the shorter list to be inserted into the longer list is varied from time to time. At some stages of the sorting process, we have to merge the two lists of size m and n with the ratio n/m as great as 1/4 (note that this ratio in Manacher's method is 1/8). At other stages of the sort process, we insert the elements one by one using the binary insertion. The purpose of this strategy is to gain as much as possible the number of comparisons over the Ford-Johnson algorithm. Once, we reached this advantage, we try to release the gained comparisons as slow as possible.

5.4 Further improvement of the Ford-Johnson algorithm.

If we sort u_k and u_{k-2} elements seperately by the fja and merge them by the Fordward-Testing-Backward-Insertion algorithm (ftbi), then for lower values of k, we have the results shown in Table 5.1. The last two columns in this table are obtained by using equations (5.5) and (3.16-18).

Table 5.1

t	u_{k-2}	u_k	$S_{fja}(u_{k-2})$	$S_{fja}(u_k)$	$S_{fja}(u_k-u_{k-2})$	$S_{fja}(u_k)-S_{fja}(u_{k-2})$ $-M_{ftbi}(u_k,u_{k-2})$
1	0	0	0	0	0	0
2	0	2	0	1	1	1
6	1	5	0	7	10	10
12	2	10	1	22	30	30
26	5	21	7	66	91	91
52	10	42	22	171	231	230
106	21	85	66	429	576	573
212	42	170	171	1024	1360	1352
426	85	341	429	2392	3157	3139
852	170	682	1024	5461	7161	7122
1706	341	1365	2392	12291	16042	15961
3412	682	2730	5461	27306	35490	35324

Column 7 shows the values of $S_{fja}(u_k + u_{k-2})$. The last column of Table 5.1 shows the value of $S_{fja}(u_k) + S_{fja}(u_{k-2}) + M_{ftbi}(u_k, u_{k-2})$ which is obtained when u_k and u_{k-2} elements are sorted seperately by the fja then merged by the ftbi.

Table 5.1 gives 52 as the smallest value for which we have achieved a better sort than the fja. At this point, the fja requires 231 comparisons whereas our method requires 230 comparisons. It is now appropriate to see whether the fja can be surpassed at values of t whiich are not of the form $t = u_k + u_{k-2}$. In other words, we want to find the sequences of values of t for whiich the fja may be surpassed.

In the following, we will denote the following notations:

$$m \sim u_{k-2}, \tag{5.11a}$$

$$4m \sim u_k, \tag{5.11b}$$

$$4.5m \sim u_k + u_{k-3}. \tag{5.11c}$$

Hence,

$$5m \sim 4m + m = u_k + u_{k-2}, \tag{5.11d}$$

$$6m \sim 5m + m = (u_k + u_{k-2}) + u_{k-2}, \tag{5.11e}$$

$$7m \sim 6m + u_{k-2}, \tag{5.11f}$$

$$8m \sim u_{k+1}. \tag{5.11g}$$

Algorithm s1 :

For $t = n+m$, where $m = u_{k-2}$, and $5m \leq t < 6m$ (i.e. $4m \leq t < 5m$).

Step 1. Sort m and n elements seperately by the fja.

Step 2. Use the ftbi algorithm to merge these sorted lists in step 1.

Theorem 5.1.

In the worst case, algorithm s1 requires $m/4-(1/2)\log m+O(1)$ comparisons less than the fja for sorting 5m elements where $m = u_{k-2}$; $n = 4m \sim u_k$.

Proof :

Using equations (5.5) and (5.7), we have for step 1 :

$$S_{fja}(m) = m\log m - (3-\log 3)m + 1\log m + O(1),$$

(5.12) $$S_{fja}(4m) = 4m\log 4m-(3-\log 3)4m+1\log m+O(1),$$

and for step 2 :

(5.13) $$M_{ftbi}(m,4m) = 3m + (3/4)m + O(1).$$

Therefore the total number of comparisons required by algorithm s1 is :

(5.14) $$S_{fja}(m) + S_{fja}(4m) + M_{ftbi}(m,4m) = 5m\log m + 11.75m - 5m(3-\log 3) + \log m + O(1)$$

whereas

(5.15) $$S_{fja}(5m) = 5m\log 5m-5m(1+f+2^{1-f}-\log 3)+\frac{1}{2}\log 5m+O(1)$$

whereas $f = \log(5/4)$. Hence,

$$S_{fja}(5m) = 5m\log m - 5m(3-\log 3) + \frac{1}{2}.\log 5m + O(1)$$

(5.16) $$= 5m\log m + 12m - 5m(3-\log 3) + 1/2.\log m + O(1).$$

Equations (5.14) and (5.16) give :

(5.17) $$S_{fja}(5m) - S_{fja}(m) + S_{fja}(4m) + M_{ftbi}(m,4m)) = m/4 - \log m/2 + O(1).$$

Q.E.D.

The result of theorem 5.1 allows us to obtain the range of the values of t in which the fja can be surpassed. Recall that $t=n+m$, we now fix $m = u_{k-2}$ and vary n in the range $4m \leqslant n < 5m$, i.e. $5m \leqslant t < 6m$ (see equations (5.11)). Since $u_k = 4m$, then

(5.18) $$u_k < 5m \leqslant t < 6m < u_{k+1}$$

and in this range :

(5.19) $$S_{fja}(t+1) - S_{fja}(t) = k+1.$$

Equation (5.19) says that in the range $u_k < t < u_{k+1}$, each additional element requires (k+1) more comparisons if the fja is used to sort all t items. However, algorithm s1 requires (k+1+1/4) more comparisons per additional element. This is obvious since $S_{fja}(m)$ is unchanged, $S_{fja}(n+1) = S_{fja}(n)+k+1$ (since $u_k<n<u_{k+1}$) and $M_{ftbi}(m,n+1) = M_{ftbi}(m,n) + 1/4$ (see equation (3.16)). Therefore :

(5.20) $$S_{fja}(m) + S_{fja}(n+1) + M_{ftbi}(m,n+1) = S_{fja}(m) + S_{fja}(n) + M_{ftbi}(m,n) + k+1+1/4.$$

The result of theorem 5.1 says that at $t=5m=u_k+u_{k-2}$ elements, algorithm s1 gains $m/4-1\log m+O(1)$ comparisons compared to the fja. However, for each

additional element, algorithm s1 looses 1/4 comparisons compared to the fja (see equation (5.19) and (5.20)).

Therefore, algorithm s1 will break even with the fja at about

$$(5.21) \qquad \frac{(m/4 - \log m/2) + O(1)}{1/4} = m - 2\log m + O(1)$$

additional elements,

i.e. at $t = 5m+m-2\log m+O(1) = 6m-2\log m+O(1)$.

Collorary 5.2.

Algorithm s1 uses fewer comparisons than the fja in the range $5m \leqslant t \leqslant 6m-2\log m$; where $m = u_{k-2}$, 5m and 6m are defined by equation (5.11).

In the following, we will further extend the range of

Algorithm s2.

For $t=n+m$, $m=u_{k-2}$, and $6m \leqslant t < 7m$ (i.e. $5m \leqslant n < 6m$).

Step 1. Sort m elements by the fja.

Step 2. Sort n elements by the algorithm s1.

Step 3. Merge the two sorted lists by the ftbi algorithm.

Theorem 5.3

Algorithm s2 requires $m/4 - \log m + O(1)$ comparisons less than the fja for sorting 6m elements, where $m = u_{k-2}$ and $n=\overline{5m}=u_k+u_{k-2}$.

Proof :

The number of comparisons in step 1 is :

$S_{fja}(m) = m\log m - (3-\log 3)m + \frac{1}{2}\log m + O(1)$.

For step 2, equation (5.14) gives :

$5m\log m + 11.75m - 5m(3-\log 3) + \log m + O(1)$ commparisons

Step 3 requires :

$$M_{ftbi}(m,5m) = 3m + (5m-m)/4 + O(1)$$
$$= 4m + O(1).$$

Therefore, the total number of commparisons required by algorithm s2 to sort 6m elements is :

(5.22) $6m\log m - 6(3-\log 3)m + 15.75m + (3/2)\log m + O(1)$.

Whereas the number of comparisons required by the fja to sort 6m elements s :

$S_{fja}(6m) = 6m\log m - 6m(1+f+2^{1-f}-\log 3) + \log m/2 + O(1)$

where $f = \log(6/4)$, hence :

(5.23) $S_{fja}(6m) = 6m\log m - 6(3-\log 3)m + 16m + \frac{1}{2}\log m + O(1)$.

Comparing equations (5.22) and (5.23), it is clear that algorithm s2 requires $m/4 - \log m + O(1)$ comparisons less than the fja for sorting 6m elements.

Q.E.D.

Similar to the discussion following theorem 5.1, algorithm s2 requires 1/2 comparisons more than the fja for each additional element (1/4 comparisons for the merging in step 2 and 1/4 comparisons for merging in step 3).

At the point t=6m, the total gain of algorithm s2 is $m/4 - \log m + O(1)$ (by theorem 5.3). Therefore, algorithm s2 will break even with the fja at about $m/2 - 2\log m + O(1)$ additional elements, i.e. at

$t=6m+m/2-2\log m+O(1) = 6\frac{1}{2}m -2\log m+O(1)$.

Collorary 5.4.

Algorithm s2 uses fewer comparisons than the fja in the range $6m \leqslant t < 6\frac{1}{2}m-2\log m$.

The results of theorems 5.1 and 5.3 show that there are two "bands" of values of t between 5m and 7m (see equations (5.11)) in which the fja can be surpassed. In the range $7m<t<8m$, the fja can not be beaten by the strategy of algorithms s1 and s2. However, the same strategy can be used to extend the bands to between 4m and 5m. Since $4m=u_k$ and $8m=u_{k+1}$, the results of this sectio which are valid for $4m<t<8m$ are also applicable to all other values t outside this range. This is because the function $S_{fja}((t)$ has the pattern for $u_k<t<u_{k+1}$ for all k. For the range between 4m and 5m, we have the following lemma.

Lemma 5.5.

Let $m=u_{k-3}$, if we sort 4 m elements by seperately sorting 4m and m by the fja and merge the sorted lists by the Forward testing Backward Insertion (ftbi) algorithm, then the new sorting method requires m/8-logm/2 comparisons less than the fja to sort $4\frac{1}{2}m$ elements.

Proof :

Use equations (5.7) and (3.17), we have :

(5.24) $S_{fja}(4m) + S_{fja}(m) + M_{ftbi}(m,4m) =$

$$\frac{9}{2}m\log m - 9m(3-\log 3) + 9\frac{7}{8}m + \log m + O(1).$$

Equation (5.7) with $f = \log(9/8)$ gives :

(5.25) $$S_{fja}(4\,m) = \frac{9}{2}m\log m - \frac{9}{2}m(3-\log 3) + 10m + \log m + O(1).$$

Hence,

(5.26) $$S_{fja}(4\tfrac{1}{2}m) - (S_{fja}(4m) + S(\tfrac{1}{2}m) + M_{ftbi}(\tfrac{1}{2}m, 4m) = m - \log m +$$

Q.E.D.

In the range $4\frac{1}{2}m \leq t < 5m$, we write $t = n + m$ where $4m < n < 4\frac{1}{2}m$. Similar to lemma 5.5, we sort n and $\frac{1}{2}m$ by t he fja, then we merge two sorted lists by the ftbi algorithm. Since $8 \leq (n/m) < 16$, equation (3.17) is used for the ftbi algorithm which shows that each additional element requires 1/8 comarisons more for the merging process. Therefore, the range in which the fja is surpassed is between $4\frac{1}{2}m$ and

(5.27) $$4\frac{1}{2}m + \frac{m/8 - \log m/2 + O(1)}{1/8} = 5m - 4\log m + O(1)$$

where $5\frac{1}{2}m = u_k + u_{k-2} + u_{k-3}$ (see equation (5.11)).

However, at t=5m, algorithm s1 gives us a better result with the gain of $m/4 - \log m/2$ comparisons.

The result of this section show that for $u_k < t < u_{k+1}$ there are bands of values of t in which the fja is surpassed. Using the notations defined by equations (5.11), the three bands are :

$$4\tfrac{1}{2}m \leq t \leq \min\{5m, 5\tfrac{1}{2}m - 4\log m\}$$

(5.28) $$5m \leq t \leq 6m - 2\log m$$

$$6m \leqslant t \leqslant 6\tfrac{1}{2}m - 2\log m.$$

For example, for k=7, these bands are:

$$191 \leqslant t \leqslant 212$$

$$212 \leqslant t \leqslant 242$$

$$254 \leqslant t \leqslant 264 .$$

5.5. Still further improvements.

In figure 5.3, we plot the regions corresponding to the inequalities (5.28). For $k \geq 7$ and $m = \lfloor 4/3.2^k \rfloor$, we always have $5m \leq 5\frac{1}{2}m - 4\log m$. Therefore, there are two regions in which our algorithm will surpass the fja. The first is between $4\frac{1}{2}m$ to $6m\frac{1}{2}2\log m$ and the second is between 6m and $6\frac{1}{2}m-2\log m$. The number of comparisons gained by our method compared to the fja at the point $4\frac{1}{2}m$ is $m/8 - \log m/2 + O(1)$, this number decreases steadily as t increases. At $t=6m-2\log m$, the gain is zero. From $6m-2\log m$ to 6m, the algorithms in the previous section cannot beat the fja. At t=6m, the gain is $m/4 - \log m + O(1)$ comparisons, this number decreases steadily to zero at $t = 6\frac{1}{2}m - 2\log m$.

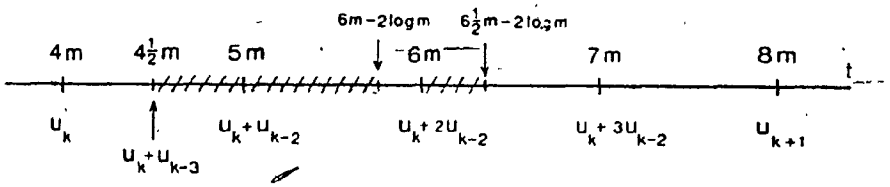


Figure 5.3

In figure 5.4, we plot the nummber of comparisons gained by the method of section (5.4) compared to the fja for k=7.

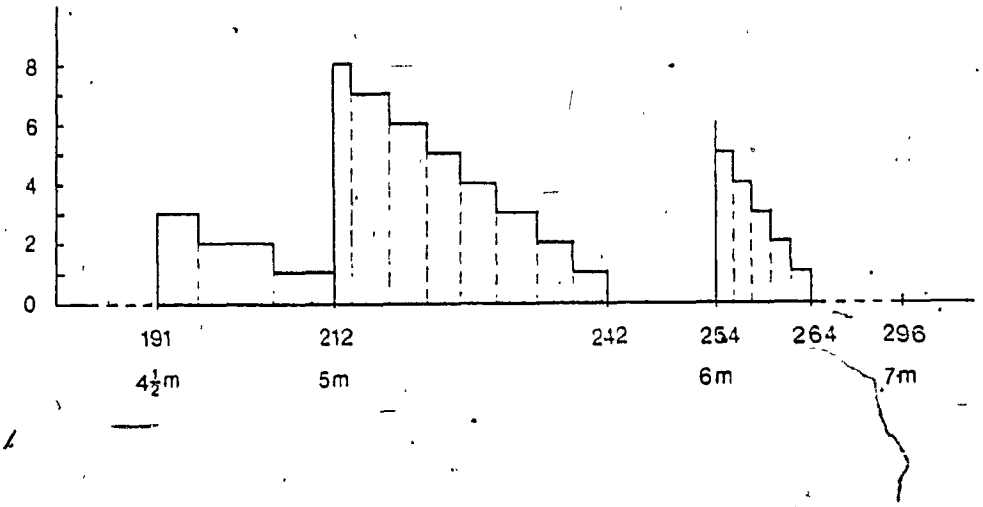


Figure 5.4

In this section, we will investigate the possibility of obtaining an eaven better improvement over the fja. The results of section 5.4 show that at t=5m, algorithm s1 gives the largest gain of m/4-logm/2+O(1) comparisons compared to the fja. THis gain reduces by one comparison for every four additional elements to be sorted until t=6m-2logm where algorithm s1 can not beat the fja. (See figure 5.4 for the case k=7). Suppose that we sort $t=u_k+u_{k-2}$ elements by algorithm s1 to achieve the largest gain. If we now increase t by one, what is the best strategy to deal with this additional element? Equation (5.20) shows that

algorithm s1 requires k+1+1/4 comparisons for this additional element. However, we note that for one additional element, the fja requires $S_{fja}(t)-S_{fja}(t-1) = \lceil \log(3/4)t \rceil$ additional comparisons. Insert one element into the list of length t requires $M(1,t) = \lceil \log t \rceil$ comparisons. For the range of t in $u_k+u_{k-2} < t < 2^{k+1}$ we have

$$\lceil \log(3/4)t \rceil = \lceil \log t \rceil = k+1.$$

This means that the additional element can be inserted by the binary insertion and we still preserve the same gain as at the point $t+u_k+u_{k-2}$. We thus arrive the following method. for sorting t elements where $u_k+u_{k-2} \leqslant t \leqslant 2^{k+1}$. We have the following lemma :

Lemma 5.6.

For any given integer t, $u_k \leqslant t < 2^{k+1}$, if a sorting method s can be found such that the fja is surpassed, i.e., $S_s(t)-S_{fja}(t)=d > 0$, then using this sorting method s, we can surpass the fja for all $t' \in [t, 2^{k+1}]$.

Proof :

This comes from $S_{fja}(t) - S_{fja}(t-1) = k+1 = M(1,t-1)$,

i.e., the difference of comparisons to spend on the extra element is the same to as to insert this element.

Q.E.D.

Algorithm s3 :

For $u_k + u_{k-2} < t < 2^{k+1}$:

Step 1. Sort u_k and u_{k-2} elements seperately by the

fja and merge them by the ftbi algorithm.

Step 2. Insert $t-(u_k+u_{k-2})$ elements into the sorted list, one by one, using the binary insertion.

We now obtain the exact number of comparisons gained by this algorithm as compared to the fja. Using equation (5.5) we have :

$$S_{fja}(t) = u_k(k-1) + \lfloor k/2 \rfloor + 1$$
$$= u_k(k-1) + \frac{1}{2}[k+1+(\frac{1}{2}+\frac{(-1)^k}{2})] \quad (5.29)$$

and

$$S_{fja}(u_k+u_{k-2}) = (u_k+u_{k-2})(k+1) - u_{k+1} + (k+1)/2 + 1$$
$$= (u_k+u_{k-2})(k+1) - u_{k+1} + [k+2+(\frac{1}{2}-\frac{(-1)^k}{2})] \quad (5.30)$$

To merge u_k and u_{k-2} sorted elements by the ftbi, we need :

$$M_{ftbi}(u_k,u_{k-2}) = 3u_{k-2} + \left\lceil \frac{u_k+u_{k-2}-2}{2} \right\rceil - 1$$
$$= 3u_{k-2} + 2^{k-2} - 1 \quad (5.31)$$

where we use the fact that

$$u_k = \left\lfloor \frac{4}{3}2^k \right\rfloor = \frac{1}{3}[2^{k+2}-1-(\frac{1}{2}-(\frac{(-1)^k}{2})] \quad (5.32)$$

The number of comparisons required by algorithm s3 to sort $t = u_k + u_{k-2}$ elements is :

$$S_{fja}(u_k) + S_{fja}(u_{k-2}) + M_{ftbi}(u_k,u_{k-2}) =$$
$$k(u_k+u_{k-2}) - u_k + 2^{k-2} - 1 + (\frac{1}{2}+\frac{(-1)}{2})^k \quad (5.33)$$

Compare equations (5.30) and (5.33), we obtain the

exact gain for all values of t in the range $u_k+u_{k-2}<t<2^{k+1}$:

$$G = S_{fja}(u_k+u_{k-2}) - S_{fja}(u_k)+S_{fja}(u_{k-2})+M_{ftbi}(u_k,u_{k-2}))$$

$$= u_{k-2} - 2^{k-2} - \frac{1}{2}[k-2- (\frac{1}{2}-\frac{(-1)^k}{2})]$$

$$(5.34) = u_{k-2} - 2^{k-2} - \left\lfloor\frac{k-2}{2}\right\rfloor.$$

For values of t larger than 2^{k+1} but less than u_{k+1}, let $t' = t-u_{k-2}$ then $u_k+u_{k-2} < t' < 2^{k+1}$.

We sort t elements by algorithm s4.

Algorithm s4.

For $2^{k+1}< t < u_{k+1}$:

Step 1. Sort $t' = t - u_{k-2}$ elements by algorithm s3, and u_{k-2} elements by the fja.

Step 2. Merge the two sorted lists in step 1 by the ftbi.

Let t_1 be the largest value in this range for which algorithm s4 still surpasses the fja. Let $Z(t_1)$ be the number of comparisons required by algorithm s3 to sort t'_1 elements where $t'_1 = t_1 - u_{k-2}$.

Then, we find t_1 by maximizing t'_1 such that :

$$(5.35) \quad Z(t'_1)+S_{fja}(u_{k-2})+M_{ftbi}(t'_1,u_{k-2}) = S_{fja}(t'_1+u_{k-2})-1.$$

Since $u_k+u_{k-2} \leq t_1<2^{k+1}$, equation (5.34) gives :

$$F(t'_1) - Z(t'_1) = G$$

Therefore, equation (5.35) becomes :

$$S_{fja}(t_1)-G+S_{fja}(u_{k-2})+M_{ftbi}(t'_1,u_{k-2}) = S_{fja}(t'_1+u_{k-2})-1$$

or

$$M_{ftbi}(t'_1,u_{k-2})-G+S_{fja}(t'_1+u_{k-2})-S_{fja}(t'_1)+G-S_{fja}(u_{k-2})-1$$

or

$$(5.36)\quad 3u_{k-2}+\left\lceil\frac{t_1-u_{k-2}-2}{4}\right\rceil \quad 1 = u_{k-2}(k+1) + G\text{-}S_{fja}(u_{k-2})-1$$

Using equation (5.34), we can solve for t_1 :

$$(5.37)\qquad t_1 = 6u_{k-2} - 4(k-2) - 3[\tfrac{1}{2}+\tfrac{(-1)^k}{2})]$$

Therefore,

$$(5.38)\qquad t_1 = 7u_{k-2} - 4(k-2) - 3[\tfrac{1}{2}+\tfrac{(-1)^k}{2})]$$

This is the maximum value of t in the range between u_k and u_{k+1}, for $k\geq 7$, where algorithm s4 surpasses the fja. for k=5,6 we have $t_1<2^{k+1}$ and therefore algorithm s3 can be used to beat the fja up to $t=2^{k+1}$(see table 5.2).

For values of t less than u_k+u_{k-2}, but grreater than u_k, we let $t' = t-u_k$ and use the following algorithm to sort t elements.

Algorithm s5.

For $u_k < t < u_{k-2}$:

Step 1. Sort $t' = t-u_k$ and u_k elements seperately by the fja.

Step 2. Merge the two sorted lists of step 1 by the ftbi.

Theorem 5.7.

Algorithm s5 use fewer comparisons than the fja when $u_{k+8} \leq t \leq u_k + u_{k-2}$.

Proof :

It is easy to check that the inequality is true at t=8,9.

Let t' be such that

$$u_r < t' < u_{r-1}$$

where $0 < r < k-3$.

Then we use algorithm s5 to seperately sort(t'), sort(u_k) by the fja and merge (t', u_k) by the ftbi.

From equations (5.5) and (3.18) we have :

$$S_{fja}(t) + M_{ftbi}(t, u_k) < t(k+1).$$

On the other hand, from equation (5.5), we have :

$$S_{fja}(t) - S_{fja}(u_k) = t(k+1).$$

Hence, algorithm s5 uses fewer comparisons than the fja in the range $[u_k+8, u_k+u_{k-2}]$.

This completes the proof.

Q.E.D.

Table 5.2

k	u_k	u_{k-1}	The range $[t_o,t_1]$ in which fja is surpassed of		
			our method	manacher's	monting's
5	42	85	[50, 64]	---	[47, 64]
6	85	170	[93,130]	---	[90,128]
7	170	341	[178,277]	[189,191]	[175,256]
8	341	682	[349,568]	[364,383]	[346,522]
9	682	1365	[690,1162]	[712,767]	[687,1064]
10	1365	2730	[1373,2352]	[1381,1535]	[1370,2138]

In table 5.2, we give the range $[t_0,t_1]$ between u_k and u_{k+1} for certain values of k in which the fja is surpassed by our algorithms given in this section. It is noted that the lowest value of t for which the fja is beaten is 50. Also, the ranges $[t_0,t_1]$ in which the fja is surpasses are broaded significantly compared to the Manacher's results and Monting's results. For example, in the case k=7, our algorithms surpass the fja for all values of t in the continous range between 178 and 274. Our algorithms gain 8 comparisons for values of t from 212 to 256. In this case (k=7), Manacher's results surpass the fja for values of t from 189 to 191 and only one comparison is gained in this region. Monting's results surpass the fja for the values of

t from 175 to 256 and gain only 3 comparisons at the point 212.

For cases k=6 and k=7, we can further extend the upper limit t_1. For k=7, we improve the fja at t=275, 276 and 277 by one comparison. This is done as follows: we sort $2^{k+1}(=256)$ elements by algorithm s3 and t'=t-256 elements by the fja. The number of comparisons required to sort 256 elements by algorithm s3 is Z(256) = 1704. Therefore, for t=275, we have $Z(256)+S_{fja}(19)+M_{ftbi}(256,19) = 1863$ whereas $S_{fja}(275) = 1864$. Similarly, for t=276, we have $Z(256)+S_{fja}(20)+M_{ftbi}(256,20) = 1871$ whereas $S_{fja}(276) = 1880$. For t=277, we have $Z(256)+S_{fja}(21)+M_{ftbi}(21) = 1879$ and $S_{fja}(277) = 1880$.

For k=6, we can extend t_1 to 130 instead of 128 by the similar approach. However, it is not possible with this approach to extend t_1 any further for other values of k.

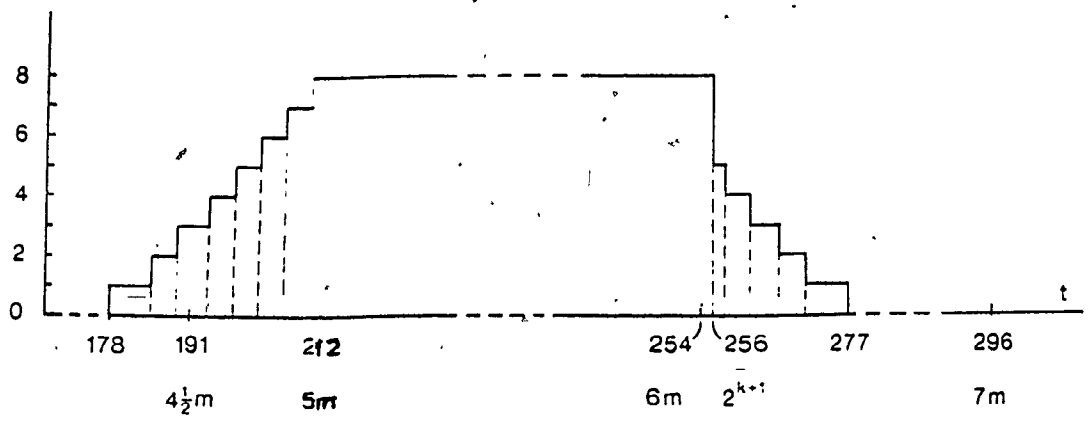


Figure 5.5

In figure 5.5, we plot the number of comparisons gained by our algorithms as compared to the fja for k=7. Finally, we observe that the lowest value of t between u_k and u_{k+1} for which our algorithm is better than the fja is always $t_\emptyset = u_k+8$, and the ratio $(t_\emptyset-u_k)/u_k$ approaches to $\emptyset$ as k tends to infinity.

The largest value of t for which our algorithms surpass the fja is given by equation (5.33). from which we find that the ratio $(u_{k+1}-t_1)/u_{k+1}$ approaches to 1/8 as k tends to infinity.

Thus, in an interval $[u_k,u_{k+1}]$, our algorithm has surpassed the fja in a range $A_k = [u_k,(1+7/8)u_k]$. The ratio of these range $|A_k|/|I_k|$ tends to 7/8 as k tends to infinity. The similar ratio in Manacher's method is 1/8 and in Monting's is 4.272/8.

Monting has proved that the 4-chain and the 3-chain are not good for beating the Ford-Johnson algorithm. So the smallest value of t in the range between u_k and u_{k+1} for which the fja can be surpassed is $t_\emptyset = u_{k+5}$. For other values of t, it is not obvious how Monting's algorithm could improve our results presented here.

CHAPTER 6

CONCLUSION

Recent works in computational complexity have shown that founding an optimal algorithm for solving a problem is essential for a throrough understanding of the potential power for some computing devices. The mathematical analysis required to prove that an algorithm is optimal can also lead to a better algorithm.

Throughout the thesis, we have presented various algorithms for Sort/Merge problems. These problems are of interest in the area of analysis of algorithms. Both sort and merge are of theoretical interest and practical importance. They have many applications in different disciplines. Sorting algorithms are widely used in the commercial world, industries, etc... Even in the algorithm design, to sort a sequence of elements is an essential part of many algorithms. Merge algorithms have been applied in Group-Testing procedures which are widely used in Medical sciences, Electronics industries, etc...

This thesis has provided a new approach for solving the merging problem, namely the Top-Down approach. This

approach is shown to be succesful for the merge(2,n) problem[Ø2]. The main feature of the approach is its systematic and algorithmic nature. This method explain the why and the how at every step of the merging procedures, therefore it lends itseft to a solution of a general merge(m,n) problem. We believe that the crucial step in an algorithmic development of merge(m,n) problem for general m and n is the development of strong assertions that will forbid a large number of configurations from further discussion. Unless some such assertions are established and the problem at each stage is transformed into an equivalent and tractable algebraic form, the general merge pattern and the finding of f(m,k) or even a close approximation of it seems extremely difficult.

We note that such assertions may be derived from the conjecture in section 4.2.1. This conjecture is very powerful. If proved, it may be used as a tool for solving the merge problem since it restricts the length of the larger list to be merged once the number of comparisons is given.

As for the average analysis, to the best of our knowledge, this is the first paper which is concerned with the average-case optimal analysis for merging problems. The average-case analysis of the particular cases for merging (2,n) and (3,n) given in chapter four may be helpful in finding the general bound M(m,n) which is not known at the present time.

As for sorting, the hybrid method we design in chapter five has improved the worst-case optimal lower bound for the sort problems. Our algorithms are neither restricted by a particular fixed sizee of the shorter list as in Monting's algorithms nor by a fixed ratio of the two lists to be merged as in Manacher's method. So, it is not surprised that the range in which our algorithms "beat" the Ford-Johnson algorithm is much larger than that of Monting and Manacher.

The main contribution of this thesis is that we have presented basic steps for solving the worst-case optimal problems for both sort and merge. However, the general sort/meerge problems remain open for future research.

R E F E R E N C E S

[Ø1] AHO,A.V., J.E.HOPCROPT and J.D.ULLMAN.
"The Design and Analysis of Computer Algorithms".
Addisson-Wesley, Reading, Mass(1974).

[Ø2] ALAGAR,V.S., T.D.BUI and MAI THANH.
"Efficient Algorithms for Merging".
Accepted for publication in BIT,(1983).

[Ø3] ALAGAR,V.S., T.D.BUI and MAI THANH.
"Efficient algorithms for merging".
Technical Report,Departmentof Computer Science,
Concordia University, 1983.

[Ø4] BUI,T.D. and MAI THANH.
"New Efficient Algorithms for Sorting".
Proceedings of the Comp.Sci.Conf., Orlando,
Florida, U.S.A., Feb 14-17, 1983.

[Ø5] BUI,T.D. and MAI THANH.
"Efficient Algorithms for Sorting ".
Submitted to the Journal of the ACM.

[Ø6] CHRISTEN,C.
"Improving the bounds on optimal merging".
Proc. 19th Annual IEEE on the Foundations of
Computer Science,1968, pp. 259-276.

[Ø7] CHRISTEN,C.
"On the optimality of the straight merging procedure".
Publication 296 ,Departement d'Informatique et de
Recherche operationnelle,University of Montreal.

[Ø8] EDWARD,M.R., N.JURG and D.NARSINGH.
"Combinatorial Algorithms: Theory and Practice".
Prentice-Hall,1977,N.J.

[Ø9] FORD,L.R.Jr. and S.B.JOHNSON.
"A Tournament Problem".
A. Math. Monthly 66,vol.5(1959),pp 387-389.

[10] GRAHAM,R.L.
"On Sorting by Comparisons".
Proc. Second Atlas Conf., 1971, pp.263-269.

[11] HWANG,F.K.
"Optimal Merging of 3 Elements with n Elements".
SIAM Journal Computing,vol.9(1980), pp.298-320.

[12] HWANG,F.K. and S.LIN.
"Some Optimality Results in Merging Two Disjoint Linearly-ordered Sets".
Memo., Bell Laboratories, Murray Hill, N.J.(1972).

[13] HWANG,F.K. and S.LIN.
"A Simple Algoritthm for Merging Two Disjoint Linearly-ordered Sets".
SIAM Journal Computing,vol.1 (1972), pp.31-39.

[14] HWANG,F.K. and S.LIN.
"Optimal Merging Two Elements with n Elements".
Acta Informatica, vol.1 (1971), pp.145-158.

[15] HWANG,F.K. and S.LIN.
"An analysis of Ford and Johnson's Sorting Algorithm".
Proc. Third Annual Princeton Conf. on Info.Sci. and and Syst., 1969, pp.292-296.

[16] KNUTH,D.E.
"The Art of Computer Programming,vol.3 : Sorting and Searching".
Addison-Wesley, Reading, Mass.,1973.

[17] KNUTH,D.E. and E.B.KAEHLER.
"An Experiment in Optimal Sorting".
IPL 1,1972, pp.173-176.

[18] MAI THANH and T.D.BUI.
"On the Complexity of Optimal Merging by Comparisons".
Proceedings of the Can.Ap.Math.Soc.Meeting, Toronto, June 20-22, 1983.

[19] MAI THANH and T.D.BUI.
"An Improvement of the Binary Merging Algorithm".
BIT 22, no.4, Dec 1982, pp.454-462.

[20] MANACHER,G.K.
"Significant Improvements to the Hwang-Lin Merging Algorithm".
J. ACM 26,vol. 3, July 1979, pp.434-440.

[21] MANACHER,G.K.
"The Ford-Johnson Sorting Algorithm is not optimal".
J. ACM 26,vol. 3, July 1979, pp.441-456.

[22] MONTING,J.S.
"Optimal Merging of 4 or 5 Elements with n-Elements".
Theoretical Computer Science 14 (1981), pp.19-37.

[23] MONTING,J.S.
"Optimal Merging of 3,4 and 5 Elements with n Elements".
Tecchnical Report, Univetsitat Tubingen (1978),W.Germany.

[24] BAASE,S.
"Computer Algorithms :
An Introduction to Design and Analysis".
Addison-Wesley, Reading, Mass (1978).

[25] STOCKMEYER,P.K. and F.F.YAO.
"On the Optimality of Linear Merge".
SIAM Jounal Computing,vol. 9 (1980), pp.85-90.

[26] TANNER,R.M.
"Minimean Merging and Sorting: An Algorithm".
SIAM Journal Computing, vol.7,Feb. 1978, pp.18-37.

APPENDIX A

Solution of the recurrence equations :

$$(A.1) \qquad \overline{T}(n) = \sum_{x=m_0}^{m'_0} \overline{T}(x-1)\overline{T}(n-x)$$

$$(A.2) \qquad T(n) = \sum_{x=m_1}^{m'_1} T(x-1)T(n-x)$$

where

$$m_\emptyset = \max\{2^{\lfloor \log n \rfloor - 1}, n-2^{\lfloor \log n \rfloor}+1\}$$

$$m'_\emptyset = \min\{2^{\lfloor \log n \rfloor}, n-2^{\lfloor \log n \rfloor - 1}+1\}$$

$$m_1 = n-2^{\lfloor \log n \rfloor}+1$$

$$m'_1 = 2^{\lfloor \log n \rfloor}$$

with the initial value :

$$T(1) = \overline{T}(1) = 1.$$

Solution :

When $n=2^k-1$, we have :

$$m_\emptyset = m'_\emptyset = \quad = 2^{k-1}-1,$$

then (A.1) becomes :

$$\overline{T}(2^k-1) = \overline{T}(2^{k-1}-1)\overline{T}(2^{k-1}-1).$$

Hence, we have :

$$\begin{aligned}\bar{T}(2^k-1) &= \bar{T}(2^{k-1}-1)\,2 \\ &= T(2^{k-2}-1)\,4 \\ &= \cdots\cdots \\ &= T(2^1-1)\,2^{k-1} \\ &= T(1)\,2^{k-1} \\ &= 1. \qquad \text{(A.3)}\end{aligned}$$

Similarly for $T(2^k-1)$, we also have :

(A.3) $T(2^k-1) = 1.$

When $n = 2^k$, we have

$$\begin{aligned}m_0 &= \max\{2^{k-1}, 2^k-2^k+1\} \\ &= 2^{k-1}, \\ m_0 &= \min\{2^k, 2^k-2^{k-1}+1\} \\ &= 2^{k-1}+1.\end{aligned}$$

(A.1) becomes :

$$\begin{aligned}\bar{T}(2^k) &= \bar{T}(2^{k-1})\bar{T}(2^{k-1}-1) + \bar{T}(2^{k-1}-1)\bar{T}(2^{k-1}) \\ &= 2\bar{T}(2^{k-1}) \\ &= 2^2\bar{T}(2^{k-2}) \\ &= \cdots\cdots \\ &= 2^k\bar{T}(1).\end{aligned}$$

Hence,

(A.4) $\bar{T}(2^k) = 2^k.$

When $n=2^k-2$, we have :

$$\begin{aligned}m_1 &= 2^k-2-2^{k-1}+1 = 2^{k-1}-1, \\ m_1 &= 2^{k-1}.\end{aligned}$$

(A.2) becomes :

$$T(2^{k-2}) = 2T(2^{k-1}-2)$$
$$= 2^2T(2^{k-2}-2)$$
$$= \ldots$$
$$= 2^{k-2}T(2-1)$$
$$= 2^{k-2}T(1).$$

Hence,

(A.4) $\quad T(2^k-2) = 2^{k-1}.$

Using the same approach, we have the solutions :

For $\bar{T}(n)$ with $n=2^k+1,\ 2^k+2,\ldots$

(A.5) $\bar{T}(2^k+1) = 2.2^{2k-2} - 2^{k-1}.$

(A.6) $\bar{T}(2^k+2) = (4/3)2^{3k-3} - 2.2^{2k-2} + (5/3)2^{k-1}$

(A.7) $\bar{T}(2^k+3) = (2/3)2^{4k-4} - 2.2^{3k-3} + (23/6)2^{2k-2} - (9/7)2^{k-1}.$

and for $T(n)$ with $n=2^k-3, 2^k-4, \ldots$:

(A.5') $T(2^k-3) = 2.2^{2k-4} - 2^{k-2}.$

(A.6') $T(2^k-4) = (4/3)2^{3k-6} - 2.2^{2k-4} + (5/3)2^{k-2}.$

(A.7') $T(2^k-5) = (2/3)2^{4k-8} - 2.2^{3k-6} + (23/6)2^{2k-4} - (9/7)2^{k-2}.$