ON SPACE-TIME TRADEOFFS

by

Sowmitri Swamy

B. Tech., Indian Institute of Technology, 1970

M.Sc., Indian Institute of Science, 1973

Thesis

Submitted in partial fulfillment of the
requirements for the Degree of Doctor of
Philosophy in the Division of Engineering

Brown University

June, 1978

Abstract of

On Space-Time Tradeoffs

by

Sowmitri Swamy

Ph.D., Brown University, June 1978

An important question in the analysis of algorithms is the nature of Space-Time exchanges for various computations. We use a pebble game played on directed acyclic graphs as a model for studying Space-Time tradeoffs for straight-line programs. These tradeoffs are generally expressed as hyperbolic functions in Space and Time with the number of inputs to the algorithm as a parameter.

1) We show that the number of pebbling moves, T, to compute the Fast Fourier Transform (FFT) algorithm using S pebbles is bounded as

$$2n^2 \geq (T-n\log_2 n)(S-\log_2 n) \geq \frac{n^2}{32}$$

A more detailed expression than the above indicates that the upper and lower bounds differ by about a factor of 4, for large n.

2) We model a linear recursion implementation that uses a stack of height n by a pebble game played on a directed acyclic graph of n vertices. The number of pebbles used, p, represents the maximum height of the stack allowed and $T_p(n)$, the number of moves in the pebble game, represents the number of recursive calls made. We obtain an exact expression for $T_p(n)$.

The rates of growth of $T_p(n)$ are

$$T_p(n) \simeq \begin{cases} p\ n^{1+1/p}(p/1+p) & p << \log_2 n \\ k_1\ n\log_2 n & p = k_2 \log_2 n \\ n\log_2 n/\log_2 p & p \gtrsim \log_2 n \end{cases}$$

where k_1 and k_2 are constants.

We present in simple terms an analysis of the space-time tradeoff which allows us to determine the entire class of optimal algorithms for linear recursion for the entire range of values of space and time. We also provide in pseudo-ALGOL simple implementations called Partial Stack Algorithms of one of the optimal algorithms.

3) We present an analysis of the space-time tradeoff of all oblivious Sorting algorithms that use the operations Minimum (a,b) and Maximum (a,b) where a and b are elements from the list to be sorted. Unlike the FFT or Linear recursion, the results apply to any straight-line algorithm of the above class. If S is the number of items temporarily stored during the execution of the algorithm and T the total number of operations performed, we show that

$$T \gtrsim \frac{n^2}{S+1} - n + S$$

4) For the class of straight-line algorithms we show that a RAM computation that uses time T and space S can be simulated by a TM in time $O(T(\log T)^3)$ and space $O(T\log T)$. We discuss the significance of such transformations as it pertains to tne idea that Space and Time are canonical parameters of computation.

This thesis by Sowmitri Swamy is accepted in its present form by the Division of Engineering as satisfying the thesis requirement for the degree of Doctor of Philosophy.

Date . . . .

Recommended to the Graduate Council

Date . . . .

Date . . . .

Approved by the Graduate Council

Date . . . .

VITA

The author was born in Bangalore, South India, on June 17, 1949. He received his B. Tech. degree in Electrical Engineering from the Indian Institute of Science, Madras, in 1970, and his M.Sc. degree in Electrical Communication Engineering from the Indian Institute of Science, Bangalore in 1973.

TABLE OF CONTENTS

To my father and mother

Narayan and Sumitra

ACKNOWLEDGEMENTS

The author wishes to express thanks to Professor J. E. Savage for his advice and constant encouragement during the research and preparation of this dissertation. The time and effort he spent on the author's behalf is sincerely appreciated and will always be remembered. Special thanks are also due to Professors S. Reiss and R. Sedgewick for their diligent reading of the dissertation. I wish to especially thank Ms. Pat Orzechowski for typing the manuscript and for the extra effort she put in to bring it to its final form. Thanks are also due to Messrs. Pierre Burgers, Kai-Bor Yu, Robert Gauthier and Yehoshua Imber for their assistance in the proofreading of the manuscript.

Chapter 1

Introduction

An important question in the analysis of algorithms is the nature of Space-Time exchanges for various computations. For example, consider the computation of finite functions viz., functions with finite domains and ranges. One way to compute such functions would be to store the value of the function for each point in its domain. The computation process would then consist of a table look-up algorithm to retrieve the required output. Another way to compute such functions would be to compute the value of the function from its formula each time an input is presented. The first method may require more Space than the second while requiring less Time.

In this dissertation we consider the class of Straight Line Programs, These are algorithms which contain no loops or jumps, conditional or unconditional. They are also known as Oblivious Algorithms [1], to denote the fact that the computation is oblivious to the value of the input involved. Many common algorithms are either of this type or can be made so by unraveling loops. The oblivious property of the algorithms makes it possible to derive precise Space-Time tradeoffs. The maximum number of references to temporary storage locations and registers is considered as the Space parameter and the total number of statements executed is the Time parameter.

A more useful idea of the same process as above is that of the pebble game, due to Paterson and Hewitt [12] and Hopcroft, Paul and Valiant [1]. The pebble game is played on directed acyclic graphs which represent the straight line algorithm. The rules of the pebble game mimic the computation process of the straight-line code. The number of pebbles used in the game is considered as the Space constraint while the minimum number of moves made during the game is considered as the Time. We observe that the computational model does not differentiate between operations in computing the total Time necessary. This leaves the algorithm uninterpreted, with the topological relationships of the steps the only significant factor. However we point out that the significant Space-Time relationships found in the thesis are in the form of lower bounds which are therefore still valid in even more complicated models of computation processes.

1) We show that the number of pebbling moves, T, to compute the Fast Fourier Transform (FFT) algorithm using S pebbles is bounded as

$$2n^2 \geq (T-n\log_2 n)(S-\log_2 n) \geq \frac{n^2}{32}$$

A more detailed expression than the above indicates that the upper and lower bounds differ by about a factor of 4, for large n.

2) We model a linear recursion implementation that uses a stack of height n by a pebble game played on a directed acyclic graph of n vertices. The number of pebbles used, p, represents the maximum height of the stack allowed and $T_p(n)$, the number of moves in the pebble game, represents the number of recursive calls made. We obtain an exact expression for $T_p(n)$.

The rates of growth of $T_p(n)$ are

$$T_p(n) \simeq \begin{cases} p\ n^{1+1/p}(p/1+p) & p << \log_2 n \\ k_1\ n\log_2 n & p = k_2\log_2 n \\ n\log_2 n/\log_2 p & p \geq \log_2 n \end{cases}$$

where k_1 and k_2 are constants.

We present in simple terms an analysis of the space-time tradeoff which allows us to determine the entire class of optimal algorithms for linear recursion for the entire range of values of space and time. We also provide in pseudo-ALGOL a simple implementation called Partial Stack Algorithm of one of the optimal algorithms.

3) We present an analysis of the space-time tradeoff of all oblivious Sorting algorithms that use the operations Minimum (a,b) and Maximum (a,b) where a and b are elements from the list to be sorted. Unlike the FFT or Linear recursion, the results apply to any straight-line algorithm of the above class. If S is the number of items temporarily stored during the execution of the algorithm and T the total number of operations performed, we show that

$$T \geq \frac{n^2}{S+1} - n + S$$

4) For the class of straight-line algorithms we show that a RAM computation that uses time T and space S can be simulated by a TM in time $O(T(\log T)^3)$ and space $O(T\log T)$. We discuss the significance of such transformations as it pertains to the idea that Space and Time are canonical parameters of computation.

Chapter 2 begins with a comparison of Space-Time parameters for a computation on the Turing Machine and Random Access Memory machine. These two models represent different aspects of real computing machines and are two popular models of computation in the analysis of algorithms. We prove that for oblivious algorithms the Space-Time parameters for both models have values which are bounded above in terms of each other by almost linear relationships. We describe the pebble game in Sec. 2.2 and provide **methods** to compute the minimum number of pebbles necessary to play the game on Trees and Synchronous Graphs. These graphs are important because they are used in deriving the Space-Time tradeoffs for the Fast Fourier Transform algorithm in Chapter 3. Section 2.3 **deals** **with** the pebble requirements of arbitrary acyclic graphs. We derive an upper bound on the number of pebbles required by a graph G of fan-in at most r. The result obtained is an improvement over that of Hopcroft, Paul and Valiant [1]. We conclude the chapter with some general remarks, in Section 2.4, on the NP-completeness of finding the exact pebble requirements.

Chapters 3, 4 and 5 provide Space-Time tradeoffs for the Fast Fourier Transform, Linear Recursion and Sorting **problems** respectively. These tradeoffs are in the form of upper and lower bounds for Time as a function of Space. The bounds obtained are shown to be **close. An impor-**tant aspect of these chapters is that each one embodies not only a Space-Time result for an important problem but **also** the methods used to obtain these results are different **in each case and thus illustrate** the potential richness of the area of Space-Time **tradeoffs.**

Chapter 3 demonstrates a hyperbolic Space-Time tradeoff for the Fast Fourier Transform algorithm. We show that the tradeoff can be expressed as

$$\frac{n^2}{32} \leq (S-\log_2 n)(T-n\log_2 n) \leq 2n^2$$

where n is the number of inputs of the algorithm. A better, though more elaborate expression, is derived for both upper and lower bounds which differ by a factor of 4 for large n.

Chapter 4 concerns Linear Recursion. Here we illustrate the massive decrease in storage Space that can be achieved with only a relatively small increase in the number of recursive calls. The unit of Space is a section of the storage space that is used to hold a record of each recursive call. Time is reckoned as the number of recursive calls made during the computation. Our approach to determining the tradeoff is one introduced by Paterson and Hewitt [12] and Chandra [18]. We present in simple terms an analysis of the Space-Time tradeoff which allows us to determine the entire class of optimal algorithms for the entire range of values of Space and Time. We also provide simple implementations of some of the optimal algorithms. As an **example of the kind of results obtained** we show that for a depth of recursion n, Space can be reduced to $\sqrt{2n}$ from a nominal n with a corresponding increase of Time to 2n from a minimum of n.

Chapter 5 treats the problem of Sorting for a whole class of algorithms. Assuming that the algorithm uses comparison-exchanges or any other monotone operations we show that the product of Space and Time grows at least as n^2, where n is the number of items that are sorted.

Chapter 6 concludes the dissertation with suggestions for further research.

Chapter 2

We discussed in general terms in the last chapter the use of Space and Time as parameters to characterize computational processes. We believe that these parameters are canonical in that the values they assume do not depend crucially on the particular model of computation which is used. To support this contention two well known models of computation are examined, that is, the multitape Turing Machine (TM) and the Random Access memory Machine (RAM). These models represent different aspects of real machines and are two models which are commonly used in analysis.

For a large class of algorithms known as "oblivious" algorithms it is shown that parameters of Space and Time obtained using the TM model of computation are related to the same parameters obtained using the RAM as a model of computation. The results are used in a discussion comparing non-oblivious and oblivious algorithms with respect to Space and Time.

Section 2.2 begins with a description of a pebble game played on directed acyclic graphs. The rules of the game model computational processes with limited space or time and thus make it possible to discuss space-time tradeoffs for specific algorithms in later chapters. Upper bounds are provided on the number of pebbles required to play the game on binary trees and a class of graphs known as synchronous graphs which are defined in Section 2.

Sections 2.3 and 2.4 consider the general problem of computing pebble requirements for arbitrary acyclic graphs. We prove an upper bound on the pebble requirement as a function of the number of nodes or edges in the graph. This is an improvement on earlier published bounds [1]. In doing so we explicitly show the influence of fan-in on the minimal space requirement. Section 2.4 mentions a relevant result on computing exact pebble requirements and briefly discusses its impact on finding optimal ways to execute an algorithm using limited Space within this model.

2.1 Models of Computation

In this section we examine two well known models of computation, that is, the multitape Turing Machine (TM) and the Random Access memory Machine (RAM). The TM and RAM computations consume resources such as Storage Space and Computation Time. RAM oblivious computations are simulated on a TM to show that the values of the parameters Space and Time based on a TM model of computation are related to the values based on a RAM model of computation. We use these results to compare non-oblivious and oblivious algorithms with respect to their Space and Time parameters.

2.1.1 Preliminaries

A k-tape, ($k \geq 2$), Turing Machine [30] consists of k semi-infinite tapes divided into cells, a tape head for each tape, a tape alphabet Σ which consists of a finite number of distinct symbols, a set of S states with an initial state $s_0 \in S$ and a finite control defined by a "transition function" $\delta : S \times \Sigma^k \rightarrow S \times \Sigma^k \times \{L,N,R\}^k$, where L,R,N denotes a head

movement

left, right or not at all. The transition function specifies the next state of the machine, the symbols to be written into tape cells under the heads and the next cell to which the heads move. The machine has a special state called a halt state, which if entered terminates further state transitions and head movements. The input for the computation appears on one of the tapes called the input tape. Other tapes are work tapes and an output tape.

It is assumed in the following discussion that the TM is deterministic, the tapes extend to infinity to the right. The TM halts for all computations that are considered below.

A TM computation begins with all tape heads at the leftmost entry, if any, on their tapes and its state at s_0. The machine then undergoes a sequence of state transitions of its finite control accompanied by head movements and possible changes in the tape contents. These are defined by the transition function and computation continues until the halt state is reached. Time is the number of transitions and Space is the number of non-blank tape cells visited by the tape heads.

A Random Access Machine (RAM) consists of a read-only input tape, a write-only output tape both over a finite tape alphabet Σ, an arithmetic/logic unit (ALU), a set of M memory locations, and a finite control called the program. The arithmetic/logic unit computes arithmetic (e.g. addition and multiplication) and Boolean functions on arguments stored in the memory. Each memory cell is of a fixed size. The program is a sequence of instructions each of which specifies either a binary arithmetic or logical operation to be performed by the ALU, an input operation specifying the transfer of symbols in Σ from the input tape to the memory, an output operation specifying the transfer of symbols from Σ from the memory to the output tape or a Jump (conditional or unconditional) operation indicating a change in the sequence of execution of the instructions in the program. A halt instruction implies no more changes in the contents of the memory M or the output tape.

A RAM computation begins with the ALU executing the first instruction of the program. Other instructions of the program are executed in sequence unless a Jump instruction specifies a break in the sequence. The computation ends when the halt instruction is executed. The total number of instructions executed by the ALU during the computation is a measure of the running-time of the program while the maximum number of memory cells used during the computation is a measure of the space needed by the program.

Definition 2.1

A TM computation P is oblivious if the position of the j^{th} head at step i in the computation on an input x is solely dependent on i, j and the length of x, for all i and j.

Informally, an oblivious algorithm is one where the sequence of steps in the computation does not depend on the value of the input.

Definition 2.2

A RAM computation is said to be a straight-line computation if the set of instructions does not contain any instruction indicating that a JUMP operation is to be executed.

2.1.2 The Simulation

We now consider some details of the simulation of a RAM computation of a straight-line program by a TM computation of an oblivious algorithm. As stated before our objective is to compare the amount of space and time needed on the TM and RAM models of computation.

Let P be a straight-line program requiring space S and time T on a RAM. We simulate program P on a TM in about $T(\log_2 T)^3$ steps and $O(T\log_2 T)$ space, with the constant implied in the big O notation depending on the TM. Without loss of generality we assume that every instruction of program P has fan-out two. [6, Chapter 2]

Program P is represented on the input tape of the TM as a sequence of records, one record per instruction. A record for the ℓth instruction of the program is of the form

$$\{u(1),\ u(2),\ \xi(\ell),\ v(1),\ v(2)\}$$

where $\xi(\ell)$, ($\xi(\ell)$: is a binary operation on a finite domain), denotes the ℓth operation with u(1) and u(2) being symbols used as place markers for the values of the arguments, and v(1) and v(2) are addresses of instructions which use the output of the ℓth instruction in their computations. The output is attached to v(1) and v(2) in the form of labels, using a constant amount of space. During the simulation u(1) and u(2) will be replaced by the arguments of $\xi(\ell)$. It should be observed that the number of tape squares needed to store a record is about $O(\log_2 T)$ since the addresses v(1) and v(2) can be represented in a maximum of $\lceil \log_2 T \rceil$ bits. Thus $O(T\log_2 T)$ tape squares are needed to represent the RAM program. The constant of proportionality implied in the O-notation depends on the TM and RAM models used.

To simulate a sequence of 2^j records (or RAM instructions) we perform the following steps:

1. (Recursively) simulate the first 2^{j-1} records.
2. Scan the first 2^{j-1} records for fan-out addresses (v(1) and v(2) in the format specification) that refer to the later set of 2^{j-1} records. Copy these addresses with the value of the **output** attached as labels onto a work tape.
3. Sort the addresses along with their **output** value labels in increasing order.
4. Copy the **output** labels into the u(1) or u(2) fields in the records referred to by the addresses.
5. (Recursively) simulate the last 2^{j-1} records.

We use the notation Sim(j) to denote the simulation of 2^j consecutive records. To implement steps 2, 3 and 4 as detailed above we use the TM programs Copy(j-1), Sort(j-1), and RCopy(j-1).

The program Copy(j-1) causes at most 2^{j-1} addresses together with their **output** labels to be copied onto the work tape as in step 2.

Of course, all the addresses copied on to the work tape refer only to the later 2^{j-1} records to be simulated.

Sort(j-1) causes the addresses on the work tape to be sorted in increasing order. RCopy(j-1) (R standing for reverse) copies the sorted **output** labels from the work tape to their respective fields (u(1) and u(2)) in the later 2^{j-1} records that are yet to be simulated. All the entries on the work tape are at the same time erased. We are now in a position to define Sim(j) to be

```
If $j \neq 0$ then
Sim(j)            [Simulate $2^j$ consecutive records corresponding
                   to $2^j$ RAM instructions]
    Sim(j-1)      [Step 1. Recursively simulate the first $2^{j-1}$
                   records]
    Copy(j-1)     [Step 2. Copy output addresses onto work tape]
    Sort(j-1)     [Step 3. Sort addresses together with fan-out
                   labels]
    RCopy(j-1)    [Step 4. Copy output labels onto latter
                   $2^{j-1}$ records]
    Sim(j-1)      [Step 5. Recursively simulate latter $2^{j-1}$
                   records]
    End
```

To complete the definition, Sim(0) is the TM program which changes the record

$$\{u'(1),\ u'(2),\ \xi(\ell),\ v(1),\ v(2)\}$$

to

$$\{u'(1),\ u'(2),\ \xi(\ell),\ \tau v(1),\ \tau v(2)\}$$

where $u'(1)$ and $u'(2)$ are values from the domain D of $\xi(\ell)$, and τ is a value label representing the result of applying $\xi(\ell)$ to $u'(1)$ and $u'(2)$. Sort(0), Copy(0), and RCopy(0) are self evident.

Many logistical details together with formal definitions of Sort(j-1), Copy(j-1) and RCopy(j-1) have not been provided since they do not bear directly on the method of simulation illustrated above. The correctness of the program can be easily proved by induction.

2.1.3 Time Required by the Simulation

The total time required by Sim(k) is determined by the two calls of Sim(k-1), and the time required by the Copy(k-1), Sort(k-1), RCopy(k-1) programs.

Let $T_{TM}(x)$ denote the time required by the TM program x.

a. $T_{TM}(\text{Copy}(k-1)) \leq c_1\ 2^{k-1}\lceil \log_2 T\rceil$, c_1 a constant.

This is because copying a string of length a in an adjacent space on a work tape requires time proportional to a. Copy(k-1) copies from at most 2^{k-1} records each $O(\lceil \log_2 T\rceil)$ in length.

b. $T_{TM}(\text{Sort}(k-1)) \leq c_2(k-1)\ 2^{k-1}\lceil \log_2 T\rceil$, c_2 a constant.

We use a sorting algorithm that allows a TM to sort a list of n items in O(nlogn) moves. One such algorithm of this kind is Radix sorting [31]. This is suitable because the items to be sorted are addresses. Since 2^k addresses are sorted, the number of moves necessary is proportional to $k\cdot 2^k\lceil \log T\rceil$. The factor $\lceil \log T\rceil$ is included because each address is of length $\lceil \log T\rceil$ and a TM needs that much time to scan an address.

We note that these sorting algorithms are not oblivious with respect to the items they sort. However when used in Sim(k) they are oblivious to the **output** tags attached to the addresses that are sorted. Indeed for some RAM programs sorting may be entirely unnecessary.

c. $T_{TM}(\text{RCopy}(k-1)) \leq c_3\ 2^{k-1}\lceil \log_2 T\rceil$, c_3 a constant.

The proof is similar to that of Copy (k-1).

From the definition of Sim(k) we have

$$T_{TM}(\underline{\text{Sim}}(k)) = 2T_{TM}(\underline{\text{Sim}}(k-1)) + T_{TM}(\underline{\text{Copy}}(k-1))$$
$$+ T_{TM}(\underline{\text{Sort}}(k-1)) + T_{TM}(\underline{\text{RCopy}}(k-1))$$

The overhead involved is not included with the above terms as it essentially consists of the number of moves necessary to scan the 2^{k-1} records, and is therefore dominated by the above terms.

$$T_{TM}(\underline{\text{Sim}}(k)) \leq 2T_{TM}(\underline{\text{Sim}}(k-1)) + c_1\ 2^{k-1}\lceil\log_2 T\rceil$$
$$+ c_2\ (k-1)\ 2^{k-1}\lceil\log_2 T\rceil + c_3\ 2^{k-1}\lceil\log_2 T\rceil$$

The constants c_1, c_2, and c_3, are independent of k and T.

This can be simplified to

$$T_{TM}(\underline{\text{Sim}}(k) \leq 2T_{TM}(\underline{\text{Sim}}(k-1)) + c\cdot k\cdot 2^k\lceil\log_2 T\rceil$$

where c is a constant independent of k and T since k $2^k\lceil\log_2 T\rceil$ dominates the other terms. This can be solved to yield

$$T_{TM}(\underline{\text{Sim}}(k)) = O(2^k\cdot k^2\lceil\log_2 T\rceil)$$

The constant of the 'O' notation is independent of T. The cost of Sim(0) is included above in the 'O' notation.

If we choose k such that

$$2^{k-1} < T < 2^k$$

then Sim(k) can simulate the RAM computation that needs time T. We need at most $O(T(\log_2 T)^3)$ TM steps to simulate a RAM computation of time T. Pippenger [3] independently obtained the same result but has since then improved the bound to $O(T(\log_2 T)^2)$.

2.1.4 Space Considerations

Let the RAM program which needed time T require space S. We then know that at no point in the RAM computation are more than S temporary results stored. We then simulate the computation on the TM in the same way as above except that the Sim(j) program using space $O(T\log_2 T)$ for any j, needs to sort at most S addresses, each $\lceil \log_2 T \rceil$ bits in length. The time $T_{TM}(\underline{\text{Sim}}(k))$ undergoes the following modification

$$T_{TM}(\underline{\text{Sim}}(k)) \leq 2T_{TM}(\underline{\text{Sim}}(k-1)) + T_{TM}(\underline{\text{Copy}}(k-1)) + T_{TM}(\underline{\text{Sort}}(S)) + T_{TM}(\underline{\text{RCopy}}(k-1))$$

If m is chosen such that

$$2^m < S < 2^{m+1}$$

we have

$$T_{TM}(\underline{\text{Sim}}(k)) \leq 2T_{TM}(\underline{\text{Sim}}(k-1)) + c_1\ 2^{k-1}\lceil \log_2 T\rceil + c_2\ S\log_2 S\lceil \log_2 T\rceil + c_3\ 2^{k-1}\lceil \log_2 T\rceil$$

for $k \geq m+1$. For $k \leq m$ we use the equation of the last section. The general solution is of the form

$$T_{TM}(\underline{\text{Sim}}(k)) \leq c'\ 2^k(\log_2 T)(\log_2 S)^2 + c''\ 2^k(\log_2 T)^2 + c'''\ 2^k\ \log_2 S\log_2 T$$

where c', c'' and c''' are constants independent of k, T and S.

We choose k to be such that

$$2^{k-1} < T \leq 2^k$$

We have

THEOREM 2.1

Any RAM oblivious computation which needs time T and space S can be simulated by a multitape TM in time T_{TM} and space S_{TM} where

$$T_{TM} = O((T\log T)(\log T + (\log S)^2))$$

with

$$S_{TM} = O(T\log T)$$

The notions of oblivious programs on a TM and straight-line programs on a RAM are identical (though technically different) and hence the two terms will be used interchangeably.

2.1.5 General Space and Time Comparisons for the TM and RAM

Thus far we have discussed transformations from RAM computations to TM computations of oblivious algorithms. To obliviously simulate arbitrary TM computations on RAM's it is sufficient to build a Boolean circuit [5] (a circuit consisting of Boolean gates) that simulates the TM computation. The Boolean circuit can be simulated obliviously by a RAM in time proportional to the number of gates.

To build the Boolean circuit that simulates the TM we build a circuit that computes the contents of T squares of each tape of the TM for each computation step of the TM. This requires about O(T) Boolean gates or $O(T^2)$ gates for T computation steps. Fischer and Pippenger [4] were able to construct a Boolean circuit using $O(T\log_2 T)$ gates that simulated T steps of a Turing computation. Schnorr [5] and Savage [6] have improved upon this to produce a Boolean circuit using $O(T\log_2 S)$ gates that simulates the T computation steps of the TM that uses space S. The RAM can thus simulate a TM computation with time T and space S in time $O(T\log_2 S)$ and space O(S). A RAM simulation of a TM [31] consists of storing the contents of each tape sequentially in memory. The position of the heads are stored in separate registers, as is the current state of the machine. Then every step in the TM computation is simulated by an updating of the contents of at most k memory locations and updating the contents of the registers containing information regarding the state and the locations of the tape heads. The RAM thus uses O(T) time and O(S) space, where T and S are the time and space of the computation.

The commutative diagram of Fig. 2.1 provides a concise way of expressing TM-RAM Space-Time transformations. The labels on the arrows denote the Time and Space transformations respectively, in the direction of the arrowheads. The O-notation is implied on the arrow labels.

From the diagram we deduce that for TM computations the parameters of Time and Space transformations from non-oblivious computations to oblivious computations is of the order of $(T\log_2 S, S)$. Unfortunately this is not true in the case of the RAM as seen below.

Consider the problem of matching an input item with a linearly ordered list of n items which are indexed. The well known Binary Search algorithm, which is non-oblivious takes time proportional to $\lceil \log_2 n \rceil$. An oblivious algorithm for the problem takes time proportional to n since the input item can be any one of n items. The space needed by both algorithms is the same. Thus in Fig. 2.1, the above example indicates that transformation β, which compares Time and Space parameters for non-oblivious RAM algorithms to that of oblivious RAM algorithms could be exponential. This forces transformation α to be exponential as well.

An interesting comparison of non-oblivious and oblivious RAM algorithms is when we consider non-oblivious RAM algorithms on n inputs with the number of instructions bounded above by $P_1(n)$, and time and space bounded above by $P_2(n)$ and $P_3(n)$ respectively. $P_1(n)$, $P_2(n)$ and $P_3(n)$ are polynomials of fixed degrees. On a TM the algorithm can be described in about $O(P_1(n)\log_2 P_1(n))$ tape squares. After a particular instruction has been simulated the next instruction to be simulated can be found by the input tape head in at most $O(P_1(n)\log_2 P_1(n))$ time, since that is the physical size of the description on the input tape. For each such instruction the work-tape head may have to travel $O(P_3(n))$ tape squares to fetch arguments if any to execute the instruction. Since there are at most $P_2(n)$ such instructions, the total time taken by the TM simulation α (see Fig. 2.1) is at most $O(P_1(n)\log_2 P_1(n) \cdot P_2(n) \cdot P_3(n))$. The space required by the simulation is $O(P_3(n))$ tape squares. Since α has been shown to be a polynomial transformation, the transformation β, from a non-oblivious RAM to an oblivious RAM is also polynomial.

2.2 Computation on Graphs

In this section we consider space limited RAM computations on graphs. Graphs are studied because they represent the topological relationship between instructions that constitute an oblivious algorithm.

We begin by describing a pebble game played on graphs which provides a good model of computation with limited space. We then provide upper bounds on the minimum space required for RAM computations for two classes of graphs, namely trees and synchronous graphs.

Graphs are studied because they represent the topological relationship between instructions that constitute an oblivious algorithm.

2.2.1 A Game on Graphs

A pebble game on directed acyclic graphs introduced in [8] provides a good model of computation with limited space. Given a directed acyclic graph G the goal of the game is to place pebbles on the output vertices of G using a certain number of pebbles under the following rules:

1. A pebble may be placed on an input vertex (a vertex with no fan-in) at any time.
2. A pebble may be placed on a non-input vertex only if all vertices that are its immediate ancestors have pebbles placed on them.
3. A pebble may be removed from a vertex at any time.

As an example consider the graph of Fig. 2.2 and the following pebbling strategy

1. Pebble vertex 3 (Rule 1)
2. Pebble vertex 1 (Rule 1)
3. Pebble vertex 5 (Rule 2)
4. Remove pebble from vertex 3 (Rule 3)
5. Pebble vertex 2 (Rule 1)
6. Pebble vertex 6 (Rule 2)
7. Remove pebble from vertex 5 (Rule 3)
8. Pebble vertex 4 (Rule 3)
9. Remove pebble from vertex 2 (Rule 3)
10. Pebble vertex 7 (Rule 2)

The above strategy indicates that vertex 7 can be pebbled with a maximum of 4 pebbles.

To study space-time tradeoffs for oblivious algorithms we represent the topological relationships between the various steps of the algorithm as a directed acyclic graph. A pebble game as described on such a graph would reflect a RAM computation if the rules of the game were interpreted as:

a. Rule 1 represents the fact that inputs can be obtained at any time.

b. Rule 2 represents the execution of an operation and the storing of the computed result when all its arguments have been computed and stored.

c. Rule 3 represents the erasure of a stored result.

We consider placing a pebble on a vertex costs one unit of time and one unit of space as it represents executing an instruction and storing the result. Removing a pebble from a vertex has no cost in time and a negative cost of 1 unit in space as no instruction has to be executed.

With this interpretation the total number of placements of pebbles (Rules 1 and 2) in a pebble game and the maximum number of pebbles used during the entire game will be a measure of the Time T and Space S of the algorithms discussed in later chapters.

2.2.2 Pebbling Trees and Synchronous Graphs

We discuss below minimum pebble requirements of trees and synchronous graphs. Tree structures are commonly used in the study of algorithms and synchronous graphs, in some cases, provide good bounds on pebble requirements of arbitrary directed acyclic graphs. Let space (j) denote the minimum number of pebbles required to reach vertex j in a graph G.

Two simple rules govern all computation of space requirements on graphs.

Rule 1 If i_1 and i_2 are immediate ancestors of vertex i and space (i_1) < space (i_2) then

$$\text{space } (i) = \text{space } (i_2)$$

<u>Rule 2</u> If space (i_1) = space (i_2)

space (i) = space $(i_1) + 1$, space $(i_1) \neq 1$

space (i) = space $(i_1) + 2$, space $(i_1) = 1$

if i_1 and i_2 cannot be simultaneously pebbled using space (i_1) pebbles. Otherwise

space (i) = space (i_1)

These rules apply only to graphs with fan-in 2, but can be easily generalized to cover arbitrary fan-in. Though the rules are simple they are difficult to apply.

2.2.2.1 <u>Trees</u>

Let G be a binary tree with root a. The following algorithm determines the exact pebble requirement of G.

```
Algorithm Tree (a)
1.  If a is a leaf then
    Space (a) ← 1
2.  Else, let $a_1$ and $a_2$ be the immediate descendants of a.
    Space (a) = max(Space $(a_1)$, Space $(a_2)$) if Space $(a_1) \neq$ Space $(a_2)$
              = Space $(a_1) + 1$ if Space $(a_1) \neq 1$
              = 3 otherwise
```

<u>Lemma 2.1</u>

Algorithm TREE computes the pebble requirement of G.

<u>Proof</u>

The strategies for computing a_1 and a_2 are disjoint since they do not share any ancestor. Rule 2 simplifies to the second step of algorithm TREE

Q.E.D.

It has been shown[12] that any tree of n vertices needs at most $\lceil \log_2 n \rceil + 1$ pebbles. However there are trees with n vertices which require a constant number of pebbles, for all n. (See Fig. 2.6).

2.2.2.2 Pebble Requirements of Synchronous Graphs

We define a rank function on the vertices of a directed acyclic graph as follows. Input vertices are of rank 0 and a non-input vertex has rank one more than the maximum rank of its immediate ancestors.

Definition 2.3

A Synchronous graph is a directed acyclic graph in which every non-input vertex has all its immediate ancestors of the same rank.

The pebble requirements of such graphs may be stated in terms of general graph parameters such as maximum depth, width or number of vertices.

Definition 2.4

Depth (G) of a directed acyclic graph G is defined to be the length of the longest path in G.

A path refers to a sequence of vertices which are in linear order in the graph.

We extend the above notion to define the depth of a vertex in G to be the length of the longest path from the vertex to the root of the graph.

Definition 2.5

Width (G) of a synchronous graph G is defined to be the maximum number of vertices of the same rank in G.

are used to pebble a graph G of depth d+1. The above method is called depth-pebbling and is used extensively.

The second assertion is proved by a method called level-pebbling. In this method pebbles are stored on all the inputs (level 0) of the graph G. Another set of pebbles are then placed simultaneously on all vertices at level 1 of the graph. The pebbles on vertices at level 0 are then removed and placed on all vertices at level 2. In this way successive levels of vertices are pebbled till the entire graph is pebbled. Since at any given time at most two successive levels have pebbles on them a maximum of 2 x Width (G) pebbles is sufficient.

Q.E.D.

When the fan-in is restricted to be at most 2 for all vertices in the graph, it can be proved that (see appendix).

$$S \leq \text{Width }(G) + 2$$

The upper bounds obtained above are fairly tight. Cook's graph (Fig. 2.3) is an example of a synchronous graph with fan-in 2 whose pebble requirement is equal to Width (G)+1 and Depth (G)+2. The concept of level and depth pebbling is important and is the basis for a Space-Time tradeoff on the Fast Fourier Transform algorithm of the next chapter.

Another method of proving an upper bound on the space required by synchronous graphs is given below.

If G is a synchronous graph of fan-in 2 with n vertices and Depth (G) > $\lceil\sqrt{2n}\rceil$, the number of levels of width > $\lceil\sqrt{n/2}\rceil$ is < $\lceil\sqrt{2n}\rceil$.

Otherwise the number of levels of width > $\lceil\sqrt{n/2}\rceil$ is at least $\lceil\sqrt{2n}\rceil$. Then

$$n \geq (\lceil\sqrt{n/2}\rceil + 1)\ (\lceil\sqrt{2n}\rceil)$$

$$\geq n + 1$$

which is a contradiction

We use the above fact to illustrate an algorithm to pebble G. The algorithm provides an upper bound on the number of pebbles required by G.

Algorithm Synchronous (G)

1. If

$$\text{Width (G) or Depth (G)} \leq \lceil 2\sqrt{2n}\rceil + 2$$

then level-pebble or depth-pebble graph G.

2a. Otherwise, as shown, there exists a level ℓ, $\ell \leq \lceil\sqrt{2n}\rceil$ with width $\leq \lceil\sqrt{n/2}\rceil$. We depth-pebble all the vertices of level ℓ storing a pebble on each vertex. The number of pebbles used is at most the sum of the number of vertices at level ℓ and the number of pebbles required to depth-pebble each vertex. This is,

$$\leq \lceil\sqrt{n/2}\rceil + \lceil\sqrt{2n}\rceil + 2$$

$$< \lceil 2\sqrt{2n}\rceil + 2, \text{ for } n \geq 2.$$

2b. If the remaining number of levels of G $\leq \lceil\sqrt{2n}\rceil$ depth-pebble the root of G, using the vertices of level ℓ as new level-zero vertices. The maximum number of pebbles used

$$\leq \lceil\sqrt{2n}\rceil + 2 + \lceil\sqrt{n/2}\rceil$$

$$< \lceil 2\sqrt{2n}\rceil + 2$$

3. Otherwise, as shown, there exists a level $\ell' > \ell$, such that $\ell' - \ell \leq \lceil\sqrt{2n}\rceil$ and width of level $\ell' \leq \lceil\sqrt{n/2}\rceil$. Depth-pebble all vertices at level ℓ' storing a pebble on each vertex. The vertices at level ℓ are used as new level zero vertices. The maximum number of pebbles needed is at most the sum of the widths of levels ℓ and ℓ' and the number of pebbles needed to depth-pebble vertices in level ℓ' using vertices in level ℓ as level zero vertices.

This is

$$\leq \lceil\sqrt{n/2}\rceil + \lceil\sqrt{n/2}\rceil + \lceil\sqrt{2n}\rceil + 2$$

$$\leq \lceil 2\sqrt{2n}\rceil + 4$$

Step 3 requires the maximum number of pebbles of all the steps. We now remove all pebbles from level ℓ. Using level ℓ' vertices as a new set of zero level vertices we repeat step 3 until the conditions for step 2b are satisfied; the root of G is pebbled using step 2b.

Lemma 2.3

Let G be a synchronous graph of fan-in 2 with n vertices. Then G can be pebbled using S pebbles where

$$S \leq \lceil 2\sqrt{2n}\rceil + 4$$

Proof

Using algorithm Synchronous (G) we need at most $\lceil 2\sqrt{2n} \rceil + 4$ pebbles in step 3.

Q.E.D.

Cook's graph (Fig. 2.3) is an example of a synchronous graph of fan-in 2 which requires about $\sqrt{2n}$ pebbles. So the upper bound of Lemma 2.3 is off by a factor of 2. The above proof can be easily extended to cover the general case of synchronous graphs of fan-in r to obtain $2\sqrt{2}\ \sqrt{(r-1)n}$ as the maximum number of pebbles required.

2.3 Space requirements for arbitrary directed acyclic graphs.

In the last section we derived bounds on the minimum pebble requirements for two special kinds of graphs. In this section we derive similar results for arbitrary directed acyclic graphs of fan-in r. We show that the number of edges in a directed acyclic graph G that requires s pebbles is at least $c_1 s \log_2 s - c_2 s$, where c_1 and c_2 are constants, c_2 being a function of the fan-in r. This is an improvement on an earlier result due to Hopcroft, Paul and Valiant [1] in that the constant c_1 is independent of the fan-in.

Let G be a directed acyclic graph of fan-in r which needs s pebbles to pebble every vertex. The pebble requirements of vertices of the graph induce a partial ordering on the vertices in the sense that a vertex requiring p pebbles cannot be a successor of any vertex requiring less than p pebbles. This makes it possible to partition the graph G into subgraphs G_1 and G_2 such that

1. G_1 consists of vertices of G such that each can be pebbled in at most s/2-r pebbles, together with all the edges between these vertices which are also in G.

2. G_2 consists of the rest of the vertices of G together with all edges between these vertices which are also in G.

All edges from vertices in G_1 to vertices in G_2 are erased. However vertices in G_2 with fan-in from vertices in G_1 are specially marked to indicate that fact. We then have

Lemma 2.4

G_2 needs at least s/2+1 pebbles to pebble all its vertices.

Proof

Otherwise G_2 needs at most s/2 pebbles. To pebble a vertex in G which is also in G_2 we use the pebbling strategy of G_2. Whenever we need to place a pebble on a vertex which is marked we pebble all its ancestors in G_1. Since the number of ancestors in G_1 is at most r, and each such ancestor requires at most s/2-r pebbles, all the ancestors in G_1 can be pebbled using no more than $(s/2-r)+(r-1) = s/2-1$ pebbles, by pebbling and storing pebbles on r-1 of the ancestors and using at most s/2-r pebbles to pebble the r^{th} ancestor. Since the strategy uses no more than s/2 pebbles on G_2 and s/2-1 pebbles on G_1, a total of at most s-1 pebbles are needed to pebble every vertex of G contrary to our assumption that G needs s pebbles.

Q.E.D.

We now obtain a series of graphs G_3, G_4, ..., G_k as follows:

To obtain G_3, we remove all vertices from G_2 that require up to s/2-1 pebbles to pebble as vertices in G. All edges incident upon these vertices are removed. Treating G_3 as a subgraph of G an argument almost identical to that of Lemma 2.4 shows that G_3 as an independent graph needs at least s/2-r+2 pebbles to pebble all its vertices. G_4 is obtained from G_3 by removing all vertices that require s/2, s/2+1, ..., s/2+r-2 pebbles as vertices in G. In general G_i is obtained from G_{i-1} by removing all vertices that as vertices of G require between $\frac{s}{2}$ + (i-4)(r-1) and $\frac{s}{2}$ + (i-3)(r-1) - 1 pebbles.

Lemma 2.5

For some k, k $\geq$ 2, G_k has the property that at least s/2-r+2 pebbles and at most s/2 pebbles are needed to pebble all its vertices.

Proof

Graph G_i needs at most as many pebbles to pebble as an independent graph as does G_{i-1} since G_i is obtained by removing vertices from G_{i-1}. Hence the pebble requirements of successive graphs must be non-increasing. By a proof similar to that of Lemma 2.4, we see that G_i requires at least as many pebbles as G_{i-1} does, less (r-1). Since the limits s/2 and s/2-r+2 cover a range of r-1, G_k must be obtained at some stage in the process.

Q.E.D.

Let the maximum pebble requirement of vertices of G removed in the above process be $s/2-r+m$, $m \geq r-1$. We define

$A = \{v/v$ a vertex of G with pebble requirement in the range $s/2-r+1, \ldots, s/2-r+m\}$

We note that A is just the set of vertices removed in the above process. Let $|A|$ denote the number of vertices in A. We use the set A to prove that there are at least $s/2$ edges in G which are neither in G_1 nor in G_k.

We attempt to pebble simultaneously and store pebbles on all the vertices in the set A using no more than $s-1$ pebbles. To obtain a lower bound on the number of edges not in G_1 or G_k we have the following.

Sublemma A

If all vertices in set A in G can be pebbled simultaneously using no more than $s-1$ pebbles, the number of edges in G from vertices in G_1 to vertices in G_k must be at least $s/2 - |A|$.

Proof

Otherwise there can be at most $s/2 - |A| - 1$ such edges, and hence at most $s/2 - |A| - 1$ vertices in G_1 which have edges to vertices in G_k. But graph G_1 needs at most $s/2-r$ pebbles to pebble all its vertices. Hence the above set of $s/2 - |A| - 1$ vertices in G_1 can have pebbles stored on them using no more than $s/2 - |A| -1-1+ s/2 - r < s - |A| - 1$ pebbles. Thus using only $s-1$ pebbles (including $|A|$ for A), pebbles can be placed on all the vertices in the set A and the $s/2 - |A| - 1$ vertices in graph G_1 which have edges to graph G_k. Notice that $|A| + s/2-|A|- 1 = s/2 - 1$ vertices will have pebbles on them at the end of this process. Since G_k can be pebbled in at most s/2 pebbles as an independent graph, G can be pebbled using an additional s/2 pebbles for a total of at most s-1 pebbles. This is a contradiction. Hence the number of edges from G_1 to G_k is at least $s/2 - |A|$.

Q.E.D.

<u>Sublemma B</u>

If all vertices of set A in G cannot be pebbled simultaneously using no more than s-1 pebbles, then $|A| > s/2$.

<u>Proof</u>

Let $\ell_m, \ell_{m-1}, \ldots, \ell_1$ be the number of vertices in A with pebble requirements $s/2-r+m$, $s/2-r+m-1$, ..., $s/2-r+1$. Some of $\ell_m, \ell_{m-1}, \ldots, \ell_1$ may be equal to zero but $\ell_m \neq 0$ by assumption. Also at least one of $\ell_1, \ell_2, \ldots, \ell_{r-1}$ is non-zero since some vertices of A have all immediate ancestors in G_1. We first pebble and store pebbles on all ℓ_m vertices that require $s/2-r+m$ pebbles each, the ℓ_{m-1} vertices that require $s/2-r+m-1$ pebbles each and so on. Since the process of pebbling and storing is assumed to be unsuccessful it fails, for some j, to pebble all ℓ_j vertices that require $s/2-r+j$ pebbles each. Thus

$$s - 1 - \sum_{p=j+1}^{m} \ell_p < s/2 - r + j + \ell_j - 1 , \quad j \geq 1$$

where the term on the left is the maximum number of pebbles remaining after pebbling and storing on $\ell_m + \ell_{m-1} \ldots \ell_{j+1}$ vertices, and the term on the right is the maximum number of pebbles needed to pebble and store all ℓ_j vertices that need $s/2-r+j$ pebbles each. That is

$$\sum_{p=j}^{m} \ell_p > s/2 - j + r$$

Consider the subset of vertices B in A that need at most $s/2-r+j-1$ pebbles each and that are ancestors of the ℓ_j vertices that each need $s/2-r+j$ pebbles. If the set B has at most $j-r$ vertices, all the vertices in set B can be pebbled simultaneously using at most $j-r$ pebbles on the set B and at most $s/2-1$ pebbles on vertices of graph G_1 which may be ancestors to vertices in the set B. That is using at most $s/2-1+j-r$ pebbles, all the vertices of set B can have pebbles stored on them. One of the ℓ_j vertices that need $s/2-j+r$ pebbles each has all its immediate ancestors in the set B and (possibly) some in the graph G_1. Since vertices in set B already have pebbles stored on them any immediate ancestors in the graph G_1 can be pebbled using no more than $s/2-1$ pebbles. Thus using at most $s/2-1+j-r$ pebbles we are able to pebble a vertex that needs $s/2+j-r$ pebbles which is a contradiction. The subset B must therefore have at least $j-r+1$ vertices. Thus

$$\begin{aligned}|A| &\geq \sum_{p=j}^{m} \ell_p + j - r + 1 \\ &\geq s/2 - j+r + j-r + 1 \\ &\geq s/2\end{aligned}$$

Q.E.D.

<u>Lemma 2.6</u>

Let E be the number of edges in G not in G_1 or G_k.

$$E \geq s/2$$

Proof

In the above process of pebbling set A, if the process **succeeds**, by Case A, there are $s/2 - |A|$ edges in G from vertices in G_1 to vertices in G_k. We can associate with each vertex in set A, an edge of G not in G_1 or G_k(say a fan-in edge).

Thus

$$E \geq s/2 - |A| + |A| \geq s/2$$

If Case B applies we have seen that

$$|A| \geq s/2$$

Since we can associate an edge in G, not in G_1 or G_k, with every vertex of A, we have

$$E \geq s/2$$

Q.E.D.

Let E(s) be the minimum number of edges of any graph of fan-in r that requires s pebbles to pebble every node. We have seen that G_1 and G_k have at least $E(s/2-r)$ and $E(s/2-r+2)$ edges respectively and the set E of edges, $|E| > s/2$, is in G but not in G_1 or G_k. Thus

$$\begin{aligned} E(s) &\geq E(s/2-r) + E(s/2 - r+2) + s/2 \\ &\geq 2E(s/2-r) + s/2 \\ &\geq \frac{1}{2}s \log_2 s - c_2 s \end{aligned}$$

and $c_2 \simeq \frac{1}{2}\log_2 r + 1$

Hopcroft, Paul and Valiant [1] first obtained a similar result. Their result is $E(S) \geq \frac{1}{2r} S \log_2 S - c_2 S$ where $c_2 \simeq \frac{1}{r} \log_2 r$.

2.4 General Remarks on Computing Pebble Requirements

A problem of some interest is to find the minimum number of pebbles to pebble a directed acyclic graph G with n vertices with each vertex pebbled just once. Fig. 2.4 shows a graph G of n vertices that requires n/2 pebbles if each vertex is pebbled once. The graph can also be pebbled with 3 pebbles if a vertex can be repebbled an arbitrary number of times. In contrast Cook's graph (Fig. 2.3) requires the same number of pebbles for both types of pebbling. These two examples indicate that the pebble requirements of the two methods of pebbling are not related in general.

As commented upon in Section 2.2 the two general rules for computing pebble requirements for graphs are hard to apply in practice. Indeed, computing pebble requirements for both methods of pebbling arbitrary directed acyclic graphs has been identified by Sethi [9] to belong to the class of NP-complete problems [10]. Solutions to these problems have been obtained only through enumeration of all possible solutions, which is a process which needs time proportional to a factorial or exponential in n, where n is the total number of inputs to the problem. An immediate consequence is that finding a strategy to pebble a graph with a given number of pebbles, not to mention an optimal strategy, is also NP-complete. This is unfortunate because while computing the exact pebble requirement may not be really necessary in practice, finding a strategy of pebbling with a given number of pebbles may be necessary. As will be seen in the next chapter, regularity of graph structures enables one to find some strategies.

CHAPTER 3

Space-Time Tradeoffs on the Fast-Fourier Transform Algorithm

3.1 Introduction

We present in the following sections an analysis of the Space-Time tradeoff for the Fast Fourier Transform (FFT) algorithm. The FFT is an algorithm for computing the discrete Fourier transform on n inputs in $O(n\log_2 n)$ steps as opposed to $O(n^2)$ steps for the naive algorithm. Cooley and Tukey's paper [18] on the FFT led to its widespread use.

The FFT is a member of the class of oblivious or straight line algorithms described in the last chapter. When the algorithm is used on a machine with a limited number of temporary storage locations it is necessary to recompute many sub-computations. This leads to a Space-time tradeoff, the topic of this Chapter.

Two reasons can be cited for studying the Space-Time Tradeoff of the FFT. One of the reasons is that the size of the input of the FFT is a drawback in a limited storage environment. Measuring instruments like a spectrometer have their maximum resolutions limited by the storage space available for the FFT algorithm.

A more important reason is that the FFT is an example of an algorithm that is a natural candidate for analysis. The algorithm and its accompanying data flow have a degree of symmetry found in few other interesting algorithms. This aids its analysis to a great extent and gives an insight into the Space-Time tradeoff for other problems.

The pebble game discussed in the last section is used as our model of a RAM computation of the FFT. We assume that we can pebble a vertex according to Rule 2 of Chapter 2, but by using a pebble already placed on one of its ancestors. The space S used in the FFT using the modified rule is exactly one less than if Rule 2 were used as in Chapter 2.

The space and time bounds of the FFT are expressed in terms of the number of pebbles used, S, and the number of pebbling moves made on a graph which represents the topological properties of the steps of the algorithm.

We obtain fairly tight upper and lower bounds on the time, T, required to pebble the FFT graph with a limited number of pebbles. The bounds for $S = 2^j + d - j,\ 1 \leq j \leq d - 1$ are

$$\frac{n^2}{2.2^j} \leq T \leq \frac{2n^2}{2^j} + (j-1)n$$

A more succinct though weaker expression in describing these bounds is

$$n^2/32 \leq (T - n\log_2 n)(S - \log_2 n) \leq 2n^2$$

The upper bound grows more rapidly than $n\log_2 n + n$ when S the number of pebbles used is $o(n/\log_2 n)$ and grows as $\frac{2}{3}n^2 + O(n)$ when $S = S_{min} + O(1)$ where S_{min}, the minimum space necessary is $\log_2 n + 1$. The upper and lower bounds differ by a factor of 4/3 when $S = S_{min}$. For other values of S, $S = 2^j + d - j$, the two bounds differ by a factor of 4.

These results are minor modifications of results first presented in Savage and Swamy [14]. Based on the methods presented in this chapter Tompa [15] has obtained bounds on the Space-Time tradeoff for two classes of graphs known as Superconcentrators and Grates.

Section 3.2.1 presents the FFT algorithm and describes the construction of the FFT graph. The properties of this graph are studied in Section 3.2.2 in the form of three lemmas describing its structure. Section 3.3 presents an upperbound on the amount of time necessary to pebble the graph using S pebbles. The bound is based on a simple method which is later modified to improve the bounds. Section 3.4 contains a derivation of a lower bound on the number of pebble placements necessary to pebble the whole graph. The method is first described for the case $S = S_{min}$ and later extended to other values of S. Section 3.5 presents the upper and lower bounds in a neater though weaker form as bounds on the expression $(S-\log_2 n)(T-n\log_2 n)$. Section 3.6 concludes the discussion.

3.2 Properties of the Fast Fourier Transform

In this section we define the Fast Fourier Transform Algorithm and study its properties. Section 3.2.1 is a brief description and derivation of the FFT algorithm intended to explain the features of the FFT graph which is used to obtain the Space-Time tradeoffs. The graph has many properties due to its symmetry. After a formal definition of the graph, necessary to provide rigor to the proofs that follow, Section 3.2.2 presents three lemmas that describe the features of the graph that are necessary in the sections that follow.

3.2.1 The FFT Algorithm

Let R be a commutative ring and let ω be a principal n^{th} root of unity in R. Let R^n be the n-direct sum of R.

We define an n x n Vandermonde matrix V in the elements 1, ω, ω^2, ..., ω^{n-1}. That is

$$V = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{n-2} & \cdots & \omega \end{bmatrix}$$

Definition 3.1

The Discrete Fourier Transform (DFT) of order n is defined to be a linear transformation of R^n onto itself as determined by the Vandermonde matrix V. That is, given an element $\underline{a} \epsilon R^n$, the DFT $\underline{b}$ of $\underline{a}$ is

$$\underline{b} = V\underline{a}$$

A convenient representation of the transformation is obtained by treating the elements of R^n as polynomials in the variable x. Thus

$$P(x) = a_o + a_1 x + \ldots + a_{n-1} x^{n-1}; \qquad a_o, a_1 \ldots, a_{n-1} \varepsilon R$$

is transformed to

$$Q(x) = b_o + b_1 x + \ldots + b_{n-1} x^{n-1}; \qquad b_o, b_1, \ldots, b_{n-1} \varepsilon R$$

where

$$b_i = P(\omega^i) \qquad 0 \leq i \leq n - 1$$

To compute the coefficients b_i, $0 \leq i \leq n - 1$ using the above definition needs $2n - 1$ ring operations for each b_i giving a total of $2n^2 - n$ ring operations required to compute all n coefficients.

The Fast Fourier Transform (FFT) is an efficient algorithm to compute the DFT. By judiciously computing common subexpressions of the coefficients only once each, the total number of ring operations is reduced and does not exceed $2n\log_2 n$ when n is a power of 2. A recursive derivation of the Fast Fourier Transform when n is a power of 2, $n = 2^d$, say, is given by Borodin and Munro [16] as follows. The odd and even terms of p(x) are collected in two polynomials $p_o(x)$ and $p_e(x)$ where

$$p_e(x) = a_o + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2 - 1}$$

$$p_o(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2 - 1}$$

so that

$$p(x) = p_e(x^2) + x\, p_o(x^2)$$

Thus, the Fourier transform of a̲, which consists of elements from $P = \{p(x) | x=2^i,\ 0 \leq i \leq n-1\}$, can be formed by computing $P_e = \{p_e(x) | x=(\omega^2)^i,\ 0 \leq i \leq \frac{n}{2} - 1\}$ and $P_o = \{p_o(x) | x = (\omega^2)^i,\ 0 \leq i \leq \frac{n}{2} - 1\}$ and combining the results as indicated above. This follows because $\omega^n = 1$. However, ω^2 is a principal n/2th root of unity so P_e and P_o represent discrete Fourier transforms of sequences of length n/2.

A simple analysis of the derivation indicates that about $2n\log_2 n$ ring operations are needed to compute the FFT of order n.

Figure 3.1.a is a representation of the straight-line algorithm that was derived above. The graph is constructed from a butterfly-like 2-input, 2-output graph shown in Figure 3.1.b. This graph represents a 2-input 2-output FFT if we assume the inputs vertices to represent for $0 \leq i \leq n-1$ the partial results $p_e(\omega^{2i})$ and $p_o(\omega^{2i})$ and the output vertices to represent the terms $p_e(\omega^{2i}) + \omega^i p_o(\omega^{2i})$ and $p_e(\omega^{2i}) + \omega^{i+n/2}\, p_o(\omega^{2i})$ respectively. We have deliberately omitted all indications of the ring operations being performed at any vertex as they do not figure in any of our later discussion.

3.2.2 Properties of the FFT graph

In this sub-section we prove some important properties of the FFT graph. These are necessary to prove bounds on the Space-Time tradeoffs for the graph and hence for the FFT algorithm. A formal definition of the FFT graph (Fig. 3.1.a) is

Definition 3.2

Let $n = 2^d$, $d \geq 1$. The FFT graph on n inputs is a directed graph $F^{(d)} = (V^{(d)}, D^{(d)}, E^{(d)})$ with vertex set $V^{(d)}$, output vertex set $D^{(d)} \subset V^{(d)}$ and edge set $E^{(d)} \subset V^{(d)} \times V^{(d)}$. It is defined recursively in terms of two disjoint FFT graphs on n/2 inputs each, ${}_oF^{(d-1)} = ({}_oV^{(d-1)}, {}_oD^{(d-1)}, {}_oE^{(d-1)})$ and ${}_1F^{(d-1)} = ({}_1V^{(d-1)}, {}_1D^{(d-1)}, {}_1E^{(d-1)})$ as follows: Let $D^{(d)} = \{v_o^{(d)}, v_1^{(d)}, \ldots, v_{2^d-1}^{(d)}\}$ be a set of vertices that is disjoint from ${}_oV^{(d-1)}$ and ${}_1V^{(d-1)}$. Then,

$$V^{(d)} = {}_oV^{(d-1)} \cup {}_1V^{(d-1)} \cup D^{(d)}$$

$$E^{(d)} = {}_oE^{(d-1)} \cup {}_1E^{(d-1)} \cup \Delta^{(d)}$$

where

$$\Delta^{(d)} = \bigcup_{i=0}^{2^{d-1}} \delta_i^{(d)}$$

$\delta_i^{(d)} = \{(v_i^{(d)}, {}_ov_i^{(d-1)}), (v_i^{(d)}, {}_1v_i^{(d-1)}), (v_{i+2^{d-1}}^{(d)}, {}_ov_i^{(d-1)}), (v_{i+2^{d-1}}^{(d)}, {}_1v_i^{(d-1)})\}$ and (v, v') denotes an edge directed from v' to v. Also, we define $F^{(o)} = (V^{(o)}, V^{(o)}, \phi)$ where $V^{(o)} = \{v_o^{(o)}\}$ and ϕ is the empty set.

The next lemma describes an important property of the FFT graph. This is used not only in proving subsequent lemmas but also in proving Theorem 3.2 which is a lower bound on the time T when the minimum amount of space necessary to pebble the graph is used. The lemma does not distinguish between directed and undirected paths.

Lemma 3.1

In an FFT graph $F^{(d)}$ the paths from any input vertex to all the output vertices of $F^{(d)}$ form a complete binary tree with 2^d leaves as do all the paths from any output vertex of $F^{(d)}$ to all the input vertices.

Proof

By induction on d.

Basis

$d = 1$. $F^{(1)}$ is the two-input two-output butterfly graph shown in Fig. 3.1b. By inspection the lemma is seen to be true.

Induction Step

The hypothesis is assumed to be true for $F^{(d-1)}$. We prove that it is true for $F^{(d)}$.

From the definition of $F^{(d)}$ there are exactly two edges from each output vertex of ${}_oF^{(d-1)}$ to a distinct pair of output vertices of $F^{(d)}$. Thus a complete binary tree with an input vertex of ${}_oF^{(d-1)}$ as its root and all the 2^{d-1} output vertices of ${}_oF^{(d-1)}$ as its leaves (which exists by the hypothesis) is also a complete binary tree with the same root and the 2^d output vertices of $F^{(d)}$ as its leaves. A similar argument holds when we consider the input vertices of ${}_1F^{(d-1)}$.

For any i, $0 \leq i \leq 2^{d-1} - 1$, the output vertex $v_i^{(d)}$ of $F^{(d)}$ is the root of a binary tree whose leaves are ${}_ov_i^{(d-1)}$ and ${}_1v_i^{(d-1)}$ which are output vertices of ${}_oF^{(d-1)}$ and ${}_1F^{(d-1)}$ respectively.

The two complete binary trees with roots ${}_ov_i^{(d-1)}$ and ${}_1v_i^{(d-1)}$ and all 2^{d-1} input vertices of ${}_oF^{(d-1)}$ and ${}_1F^{(d-1)}$ respectively as leaves (which exist by the hypothesis) form a complete binary tree with the output vertex $v_i^{(d)}$ of $F^{(d)}$ as its root and all the 2^d input vertices of ${}_0F^{(d-1)}$ and ${}_1F^{(d-1)}$ together, as its leaves. A similar argument holds for the remaining output vertices $v_{i+2^{d-1}}$, $0 \leq i \leq 2^{d-1} - 1$ of $F^{(d)}$.

Q.E.D.

Since the FFT graph $F^{(d)}$ is synchronous (by inspection) the concept of levels of the graph is well defined. The input vertices of $F^{(d)}$ are at level 0 and the output vertices are at level d. There are a total of d+1 levels in the entire graph.

We separate $F^{(d)}$ at level d-j into two sets of graphs, one set with its input vertices at level d-j and output vertices at level d, called the $A^{(j)}$ graphs, the other set with its input vertices at level 0 and output vertices at level d-j called the $B^{(d-j)}$ graphs. These graphs are necessary in proving the lower bound of Theorem 3.3. We observe that the set $A^{(j)}$ consists of 2^{d-j} disjoint FFT graphs $F^{(j)}$ on 2^j inputs each. The set $B^{(d-j)}$ consists of 2^j disjoint FFT graphs $F^{(d-j)}$ on 2^{d-j} inputs each. In addition, every input vertex of an FFT graph $F^{(j)}$ in the set $A^{(j)}$ can be identified with an output vertex of a unique FFT graph $F^{(d-j)}$ in the set $B^{(d-j)}$. These observations are proved below.

Lemma 3.2

For $d \geq 2$ and each $1 \leq j \leq d-1$ the FFT graph $F^{(d)}$ can be represented as the composition of 2^{d-j} disjoint FFT graphs on 2^j inputs $\{A_\ell^{(j)} | 0 \leq \ell \leq 2^{d-j} - 1\}$ with 2^j disjoint FFT graphs on 2^{d-j} inputs $\{B_m^{(d-j)} | 0 \leq m \leq 2^j - 1\}$ where the m th input to $A_\ell^{(j)}$ is the ℓ th output of $B_m^{(d-j)}$ for $0 \leq m \leq 2^j-1$ and $0 \leq \ell \leq 2^{d-j}-1$. The i th output of $F^{(d)}$

is the r th output of $A_s^{(j)}$ for the unique integers r, s ≥ 0 such that $i = r2^{d-j} + s$ and $s < 2^{d-j}$. (Refer to Fig. 3.1.a).

Proof

By induction.

Basis

$d = 2$, the hypothesis is true by inspection.

Inductive Step

Assume that the hypothesis is true for $d \leq d_o - 1$ and show that it holds for $d = d_o$. Let $j = 1$. Then $F^{(d_o)}$ is formed from ${}_0F^{(d_o-1)}$ and ${}_1F^{(d_o-1)}$ by adding the output nodes $D^{(d_o)}$ and edges $\Delta^{(d_o)}$. But the graphs $(\{v_i^{(d_o)}, {}_0v_i^{(d_o-1)}, v_{i+2^{d_o-1}}^{(d_o)}, {}_1v_i^{(d_o-1)}\}, \{v_i^{(d_o)}, v_{i+2^{d_o-1}}^{(d_o)}\}, \delta_i^{(d_o)})$, $0 \leq i \leq 2^{d_o-1}$, are 2^{d_o-j} FFT graphs $\{A_\ell^{(1)}\}$ on two inputs so the hypothesis follows from the definition of $F^{(d_o)}$ when $j = 1$.

Now let $2 \leq j \leq d_o - 1$ and consider the graphs ${}_0A_\ell^{(j-1)}$, ${}_0B_m^{(d_o-j)}$ and ${}_1A_\ell^{(j-1)}$, ${}_1B_m^{(d_o-j)}$ into which ${}_0F^{(d_o-1)}$ and ${}_1F^{(d_o-1)}$ may be decomposed for $0 \leq \ell \leq 2^{d_o-j}-1$ and $0 \leq m \leq 2^j - 1$. These four sets of graphs are disjoint except for nodes that the first two and last two sets have in common. The graphs ${}_0A_\ell^{(j-1)}$ and ${}_1A_\ell^{(j-1)}$ together with the output nodes $\{v_{i(s)}^{(d_o)}, v_{i(s)+2^{d_o-1}}^{(d_o)} \mid i(s) = \ell 2^{d_o-j} + s, 0 \leq s \leq 2^{d_o-1} - 1\}$ and the edge sets $\{\delta_{i(s)}^{(d_o)} \mid i(s) = \ell 2^{d_o-j} + s, 0 \leq s \leq 2^{d_o-1} - 1\}$ form an FFT graph on 2^j inputs $A_\ell^{(j)}$, as can be seen from the definition of an FFT graph. Furthermore, there are 2^{d_o-j} such graphs $\{A_\ell^{(j)}\}$ where the m th input to

$A_\ell^{(j)}$ is the ℓ th output of ${}_0B_m^{(d_o-j)}$ if $0 \leq m \leq 2^{d_o} - 1$ or the ℓ th output of ${}_1B_m^{(d_o-j)}$ if $2^{d_o-1} \leq m \leq 2^{d_o} - 1$. If we let $B_m^{(d_o-j)}$ be ${}_0B_m^{(d_o-j)}$ in the first case and ${}_1B_m^{(d_o-j)}$ in the second, the conclusion follows.

Q.E.D.

Another property of the $F^{(d)}$ graph which is used in deriving upper bounds for the Space-Time tradeoff concerns the number of $A^{(j)}$ graphs at level d-j that have common ancestor vertices in the $B^{(d-j)}$ FFT graphs. This is discussed in Lemma 3.3. We define

Property P(j) A set of 2^k FFT graphs in $A^{(j)}$ have property P(j) if their input vertices can be partitioned into subsets $\{I_h\}$, $0 \leq h \leq 2^j - 1$ with each set containing 2^k vertices with one input vertex from each of the 2^k FFT graphs and having $(2^{d-j+1} - 2^k)$ ancestor vertices from the graphs $B^{(d-j)}$ in common.

Lemma 3.3

Consider the level d-j, $1 \leq j \leq d-1$ of the graph $F^{(d)}$. For $1 \leq k \leq d - j - 1$, the set of 2^{d-j} FFT graphs $A^{(j)}$ can be partitioned into subsets $\{D_g^{(k)}\}$, $0 \leq g \leq 2^{d-j-k} - 1$, of 2^k FFT graphs $F^{(j)}$ each, with the property P(j). (See Fig. 3.1.a).

Proof

Consider an arbitrary FFT graph $F^{(j)}$ from the set $A^{(j)}$. From Lemma 3.2 we know that every input vertex of this graph is a root of a complete binary tree of depth d - j.

The complete binary tree with its leaves at level 0 and an input vertex of $A_\ell^{(j)}$, $0 \leq \ell \leq 2^{d-j} - 1$, as its root has two complete binary subtrees of depth d - j - 1 all of whose vertices are in the FFT graphs $B^{(d-j)}$. Since every vertex in the entire FFT graph $F^{(d)}$ has a fan-out of 2 the two complete binary subtrees of depth d - j - 1 are also the subtrees of another complete binary tree with its leaves at level 0 and its root as an input vertex of a graph $A_m^{(j)}$,

$0 \leq m \leq 2^{d-j} - 1$. However $\ell \neq m$ since the input vertices of $A^{(j)}$ are the roots of distinct binary trees. Thus a pair of input vertices of $A_m^{(j)}$ and $A_\ell^{(m)}$ share two complete binary trees of depth $d - j - 1$ all of whose vertices are in $B^{(d-j)}$. The number of vertices is $(2^{d-j} - 1) \times 2 = 2^{d-j+1} - 2$.

The same argument is true for general k. We group together the 2^k complete binary subtrees each of depth $d - j - k$, of the complete binary tree of depth $d - j$ mentioned above. These trees are common ancestor vertices of 2^k input vertices of the set $A^{(j)}$ each of which can be identified to belong to a unique FFT graph $A_\ell^{(j)}$. The total number of vertices in the 2^k subtrees of depth $d - j - k$ each is

$$(2^{d-j-k+1} - 1)\, 2^k = 2^{d-j+1} - 2^k$$

Q.E.D.

The graphs of Figure 3.1 which represents $F^{(3)}$ illustrates Lemmas 3.1, 3.2 and 3.3.

The construction of $F^{(3)}$ from ${}_oF^{(2)}$ and ${}_1F^{(2)}$ by the addition of output vertices and edges to them from output vertices of ${}_oF^{(2)}$ and ${}_1F^{(2)}$ should be clear. The graph has four levels with inputs at level 0 and outputs at level $d = 3$. There are four edge sets $\delta_i^{(3)}$, $0 \leq i \leq 3$, and each set with corresponding input and output vertices forms an FFT graph on 2-sequences. In particular, that associated with $\delta_o^{(3)}$ is $(\{v_o^{(3)}, v_4^{(3)}, {}_ov_o^{(2)}, {}_1v_o^{(2)}\}, \{v_o^{(3)}, v_4^{(3)}\}, \delta_o^{(3)})$. These four FFT graphs are disjoint and take their inputs from the outputs of ${}_oF^{(2)}$ and ${}_1F^{(2)}$. The ith of these four graphs has as inputs the ith outputs of ${}_oF^{(2)}$ and ${}_1F^{(2)}$ for $0 \leq i \leq 3$. Also, as can be seen by inspection of Fig. 3.1a the vertices $\{v_o^{(3)}, v_2^{(3)}, v_4^{(3)}, v_6^{(3)}, {}_ov_o^{(2)}, {}_ov_2^{(2)}, {}_1v_o^{(2)}, {}_1v_2^{(2)}, {}_{oo}v_o^{(1)}, {}_{1o}v_o^{(1)}, {}_{o1}v_o^{(1)}, {}_{11}v_o^{(1)}\}$ and their connecting edges form an FFT graph on 4-sequences, as do the remaining vertices at levels 1, 2, and 3 and connecting edges. Their inputs are taken from the outputs of four FFT graphs on 2-sequences, namely, ${}_{oo}F^{(1)}$, ${}_{1o}F^{(1)}$, ${}_{o1}F^{(1)}$, and ${}_{11}F^{(1)}$. Also, the i^{th} of these two FFT graphs on 4-sequences takes its inputs from the i^{th} outputs of these four FFT graphs for $0 \leq i \leq 1$.

3.3 An Upper Bound

In this section we present an upper bound on the Space-Time tradeoff of the FFT algorithm.

We derive an upper bound by estimating the maximum number of moves necessary in a pebble game that pebbles all the outputs of the FFT graph using a given number of pebbles. The rules of the game were described in Section 2.2. The maximum number of pebbles used during the game and the number of pebble placements on vertices made during the game are a measure of the Space S and Time T parameters of the implementation of the FFT algorithm.

We now describe a pebbling strategy for $F^{(d)}$ and derive an upper bound on time T using space S. To simplify the explanation we first derive an upper bound using a simple pebbling strategy. We indicate how this strategy can be modified using Lemma 3.3 to obtain a better upper bound. We subtract a "correction" term arising from the modification of the strategy for the upper bound derived earlier.

The FFT graph $F^{(d)}$ can be pebbled in $(d+1)2^d$ moves with $S = 2^d + 1$ pebbles by the level-pebbling technique discussed in Chapter 2. The number of moves made during the level-pebbling strategy is exactly one move for every vertex of $F^{(d)}$. Thus we use exactly $(d+1)2^d$ moves to pebble $F^{(d)}$.

To pebble $F^{(d)}$ with $S = 2^j + d - j$ pebbles, $1 \leq j \leq d - 1$, we consider level $d - j$ of the graph and pebble $A_\ell^{(j)}$, $0 \leq \ell \leq 2^{d-j} - 1$, by first pebbling and storing pebbles on its 2^j input vertices and then level-pebbling the remaining vertices using $2^j + 1$ pebbles. The input vertices of $A_\ell^{(j)}$ at level $d - j$ are the roots of 2^j complete binary trees of $B^{(d-j)}$. The number of moves made on the set $B^{(d-j)}$ to pebble the input vertices of $A_\ell^{(j)}$ at level $d - j$ is $2^j(2^{d-j+1} - 1)$ since the subtrees rooted at the input vertices of $A_\ell^{(j)}$ are 2^j in number and have depth $d - j$. (The number of vertices in a complete binary tree of depth $d - j$ is $2^{d-j+1} - 1$). An additional $j \cdot 2^j$ moves are made while pebbling the non-input levels of the $A_\ell^{(j)}$ graph. Since the above computation is repeated for each ℓ, $0 \leq \ell \leq 2^{d-j} - 1$ we have the following upper bound. The FFT graph $F^{(d)}$ can be pebbled in T moves with S pebbles where

$$T \leq \begin{cases} \dfrac{2n^2}{2^j} + (j-1)n & \text{for } S \geq 2^j + d - j,\ 1 \leq j \leq d - 1 \\[2ex] n(1 + \log_2 n) & \text{for } S \geq 2^d + 1 \end{cases}$$

where $n = 2^d$.

The upper bound obtained above can be modified to take advantage of the common ancestor vertices in $B^{(d-j)}$ that the input vertices of $A^{(j)}$ share according to Lemma 3.3.

We divide the FFT graphs $A^{(j)}$ into subsets $D_g^{(1)}$, $0 \leq g \leq 2^{d-j-1} - 1$, each of which consists of two FFT graphs $A_p^{(j)}$ and $A_q^{(j)}$, $0 \leq p,q \leq 2^{d-j} - 1$, according to Lemma 3.3. When the last input vertex of $A_p^{(j)}$ is pebbled the corresponding input vertex of $A_q^{(j)}$ is also pebbled in one more pebble placement. Since the two subtrees of depth d - j - 1 were not repebbled in order to pebble the input vertex of $A_q^{(j)}$ the number of pebble placements avoided

$$= (2^{d-j} - 1)2 = 2^{d-j+1} - 2$$

Since the same argument holds for each of the sets $D_g^{(1)}$, $0 \leq g \leq 2^{d-j-1} - 1$, the total number of pebble placements saved

$$= (2^{d-j+1} - 2) \times 2^{d-j-1}$$

Observe that there are always $2^j + 1$ pebbles available to pebble $A^{(j)}$.

We now consider the sets $\{D_g^{(2)}\}$ $0 \leq g \leq 2^{d-j-2} - 1$ each of which consists of a pair of subsets $D_{g1}^{(1)}$, $D_{g2}^{(1)}$, $0 \leq g_1, g_2, \leq 2^{d-j-1} - 1$. During the pebbling of the last input vertex of the pair of FFT graphs $D_{g_1}^{(1)}$ we place two pebbles on the two roots of a subtree of depth d - j - 1. One root is an ancestor of the FFT graphs $D_{g_1}^{(1)}$, and shares two subtrees of depth d - j - 2 with the other root which is an ancestor of the FFT graphs $D_{g_1}^{(1)}$. The savings in pebble placements is the number of vertices in two complete binary trees of depth d - j - 2. This is

$$(2^{d-j-1} - 1)\ 2 = 2^{d-j} - 2$$

Since there are 2^{d-j-2} sets $\{D_g^{(2)}\}$ the total savings in pebble placements amounts to

$$(2^{d-j} - 2)\ 2^{d-j-2}$$

In general it can be seen that the total savings in pebble placements for the sets $\{D_g^{(k)}\}$ is

$$(2^{d-j-k+2} - 2)\ 2^{d-j-k}$$

for $1 \leq k \leq d - j$. We observe that we are using exactly one pebble to place and store on a vertex which is not an ancestor of the $A^{(j)}$ graph whose last input is being presently pebbled. This is possible for all values of j except $j = d-1$. For $j = d-1$, $2^j+d-j=2^{d-1}+1$ pebbles are needed to pebble the $A^{(d-1)}$ graph and hence no spare pebbles are available to be used to save computations.

We note that the strategies used to pebble the subsets $\{D_g^{(1)}\}$, $\{D_g^{(2)}\}$, ..., $\{D_g^{(d-j)}\}$ are used independently of one another and therefore the total savings in pebble placements for the entire strategy is the sum of the savings in pebble placements realized by the individual strategies for $\{D_g^{(1)}\}$, $\{D_g^{(2)}\}$, ..., $\{D_g^{(d-j)}\}$.

Therefore

$$\begin{array}{l}\text{Total Savings}\\ \text{in Time}\end{array} = \sum_{k=1}^{d-j} (2^{d-j-k+2} - 2)\ 2^{d-j-k}$$

$$= \frac{4}{3}\ 2^{2d-2j} - 2^{d-j+1} + 2/3$$

For $j = d-1$, the above expression is equal to 2. In the final statement of the theorem a constant 2 is added to the upper bound to compensate for the savings of 2 that cannot be achieved for $j = d-1$.

The simple strategy to pebble $F^{(d)}$ described in the beginning of this section pebbled all the ancestor vertices in $B^{(d-j)}$ of the inputs of $A_\ell^{(j)}$ individually for every ℓ, $0 \leq \ell \leq 2^{d-j} - 1$. Thus an upper bound on the time required by the modified strategy is obtained by subtracting the total saving in time as detailed above from the upper bound obtained earlier. Thus

Theorem 3.1

The FFT graph $F^{(d)}$ can be pebbled in T moves with S pebbles where

$$T \leq \frac{2n^2}{2^j} + (j-1)n - \frac{4}{3}\,\frac{n^2}{2^{2j}} + \frac{2n}{2^j} - 2/3 + 2$$

for

$$S \geq d + 2^j - j, \quad 1 \leq j \leq d - 1$$

and

$$T \leq n(1+d) \qquad \text{for } S \geq 2^d + 1$$

The upper bound grows more rapidly than nlogn+n when S, the number of pebbles used is $o(n/(\log n))$ and grows as $2/3n^2 + O(n)$ when $S = S_{min} + O(1)$ where $S_{min} = d + 1 = \log_2 n + 1$. As will be seen in the next section the upperbound of $\frac{2}{3}n^2$ for $S = S_{min} + O(1)$ cannot be improved by more than a factor 4/3. For the case when $S = n + 1$ the bound is exact. The bounds as stated are for certain values of S. For $S \neq 2^j + d - j$ the upper bound can be improved by better estimations of the savings in moves on the $B_m^{(d-j)}$ possible. Of course the bound obtained for $S = d + 2^j - j$ applies to $S > d + 2^j - j$ as well.

3.4 Lower Bounds on Space-Time Tradeoffs

The key to the derivation of lower bounds on T given $S = d + 2^j - j$ is the observation that in the pebbling strategy given above, exactly one move is made on each vertex of $A_\ell^{(j)}$, $0 \leq \ell \leq 2^{d-j} - 1$, that is, vertices at level $d - j$ or above, but that most vertices of $B_m^{(d-j)}$, $0 \leq m \leq 2^j - 1$, or vertices at levels less than $d - j$ have many moves made on them, that is, they are recomputed many times. An argument based on this observation is presented but is preceded by an argument for the special case of $S = d + 1$. This will allow us to introduce a number of ideas that are useful in the later presentation.

3.4.1 The Case $S = d + 1$

By Lemma 3.1 the paths from a given output of $F^{(d)}$ to all the inputs form a complete binary tree of depth d. From Chapter 2 we know that a complete binary tree of depth d requires $d + 1$ pebbles to pebble its root.

In order to obtain lower bounds on the amount of time needed to pebble $F^{(d)}$ we identify specific pebble patterns which occur during the pebbling of every output and calculate the minimum number of moves necessary between two successive patterns.

We identify one such pattern that occurs during the pebbling of every output. The point in time during the pebble game at which the pattern occurs is called a Critical Time.

Definition 3.2

A Critical Time associated with the pebbling of an output of $F^{(d)}$ is the first time that every path from the output to all the inputs is blocked, that is, has a pebble on it.

The pebble pattern at this time is called a Critical Event.

Lemma 3.4 (Paterson and Hewitt [12])

For a complete binary tree of depth d a critical event using d + 1 pebbles can occur only when there is a single pebble at levels d - 1, d - 2, ..., 2, 1 that blocks 2^{d-1}, 2^{d-2}, ..., 2 paths respectively and two pebbles at level 0 which block the last 2 remaining paths.

Proof

We can associate every leaf of the tree with a distinct path from the leaf to the root. Consider the first point in time when the last path from a leaf to the root that is pebble free is closed. Branching off from this path are d distinct subtrees to the remaining leaves. The subtrees contain 2^{d-1}, 2^{d-2}, ..., 2, 1 distinct leaves respectively. Since paths from the root to these leaves are assumed blocked and since there is only 1 pebble per subtree, the pebbles will have to be placed at the roots of these subtrees corresponding to levels d-1, d-2, ..., 2, 1, 0. Since the last open path has just been blocked the d+1st pebble must have been placed on a leaf at level 0.

Q.E.D.

Figure 3.2 illustrates the position of pebbles at a critical event.

Consider the first output for which a critical event occurs when pebbling $F^{(d)}$ with d + 1 pebbles. By Lemma 3.4 there can be only a single pebble on ${}_0F^{(d-1)}$ or ${}_1F^{(d-1)}$ at level d-1. From Lemma 3.3 we see that exactly two outputs can have the same critical event. Before the next distinct critical time occurs the portion of $F^{(d)}$ which contained a single pebble at height (d-1) will have to have all paths from its inputs to the new output closed by either 1 pebble at level d-1 or d pebbles at devels d-2, d-3, ..., 1, 1 according to Lemma 3.4. In the first case 2^d-1 moves are necessary (since we have to pebble a complete binary tree of depth d-1) while in the latter at least 2^d-d moves are necessary (since we pebble a complete binary tree of depth d-1 except for d-1 vertices along a path from an input to the root, the input being pebbled). To each of these we may add 2 moves made to pebble the 2 outputs which have the same critical event.

Since there are 2^d outputs there are 2^{d-1} pairs of outputs with distinct critical events. There are at least $(2^d-d)+2$ moves between two distinct critical events. There have to be at least $2^d-1 + 2^d-d$ moves made before the first critical event (to achieve the pebbling pattern of Lemma 3.4). We thus obtain a total of

$$2^d-1 + (2^d-d+2)2^{d-1}$$

Theorem 3.2

At least $T \geq 2^{d-1}(2^d-d+4) - 1$ moves are necessary to pebble $F^{(d)}$ with S = d+1 pebbles, the minimum number of pebbles necessary.

Comparing Theorems 3.1 and 3.2 we see that the lower bound differs from the upper bound by a factor of 4/3 for large d.

3.4.2 The General Case

The generalization of the above argument becomes clear if we assume that S is in the range

$$2^{j-1} + d - (j-1) < S \leq 2^j + d - j$$

and concentrate our attention on the set of $A^{(j)}$ graphs at level d - j. Each of these graphs has 2^j inputs at level d - j and we define a critical time for an output of $F^{(d)}$ below.

Definition 3.3

A j-Critical Time associated with the pebbling of an output of $F^{(d)}$ is the first time that every path from the output to all its inputs at level j has a pebble on it.

The pebble pattern at the j-critical time is called a j-Critical Event. Unlike the case in Section 3.4.1 it is not possible to identify any unique pebble pattern that occurs during a j-critical event. Instead we derive estimates on the number of outputs of $F^{(d)}$ that could possible have j-critical events at the same time. We show that if there are a pebbles on the $A^{(j)}$ graphs at a given j-critical time then at most 2^{j+1} outputs have paths blocked by pebbles to a/2 or more inputs of $A^{(j)}$. This implies that about 2^d moves have to be made on vertices in the $B^{(d-j)}$ graphs between the given j-critical time and a j-critical time for one of the $(2^d - 2^{j+1})$ outputs of $F^{(d)}$ which have less than a/2 paths to inputs of the $A^{(j)}$ graphs blocked by pebbles. This means that there are $2^d/2^{j+1}$ distinct j-critical events during the entire pebbling with the property that there are about 2^d moves between any two successive j-critical events. This is the lower bounding argument.

Lemma 3.5

Let a pebbles reside on the subgraphs $A_\ell^{(j)}$, $0 \leq \ell \leq 2^{d-j} - 1$ of $F^{(d)}$. Let $N(h)$ be the number of outputs of $F^{(d)}$ each of which have paths blocked by pebbles to h or more input vertices of the graphs $A^{(j)}$. Then

$$N(h) \cdot h \leq a \cdot 2^j$$

Proof

The graph $A_\ell^{(j)}$, $0 \leq \ell \leq 2^{d-j} - 1$, is an FFT graph on 2^j inputs. From Lemma 3.1 and 3.2, a pebble placed at level k of $A_\ell^{(j)}$ blocks 2^k inputs for each of 2^{j-k} outputs. If C_i is the number of inputs of the set $A^{(j)}$ blocked by pebbles from the ith output, we have

$$\sum_{i=0}^{2^d-1} C_i \leq a \cdot 2^j$$

since each pebble contributes 2^j to the sum. But

$$N(h) \cdot h \leq \sum_{i=0}^{2^d-1} C_i$$

Thus

$$N(h) \cdot h \leq a \cdot 2^j$$

Q.E.D.

Corollary 1

$N(a/2) \leq 2^{j+1}$

Corollary 2

$N(2^{j-1}) \leq 2a$

When the first j-critical event occurs for some output of $F^{(d)}$ by Corollary 1 there are at most $2^{j+1} - 1$ additional outputs with the property that for each output paths from $\lfloor a/2 \rfloor$ or more inputs of the $A^{(j)}$ graphs are blocked by pebbles. Thus there exists an output v of $F^{(d)}$, not in the above set, such that v has at most $\lfloor a/2 \rfloor$ of the inputs of $A^{(j)}$ blocked by pebbles. The output v has exactly 2^j inputs at level $d - j$ and all these inputs will have to be pebbled at least once before a j-critical event of v can occur. Thus at least $2^j - \lfloor a/2 \rfloor$ inputs of $A^{(j)}$ will have to be pebbled before output v's j-critical event can occur.

The first j-critical event occurs when all the paths from some output to the inputs of $A^{(j)}$ are closed for the first time. This occurs when some input of the graphs $A^{(j)}$ has a pebble placed on it at the j-critical time. But the paths from this input of $A^{(j)}$ to the inputs of $F^{(d)}$ form a complete binary tree of depth $d - j$, which needs at least $d - j + 1$ pebbles to pebble the root. Thus $\underline{a}$, the number of pebbles on the $A^{(j)}$ graphs at the critical time must satisfy

$$a + d - j + 1 + b = S - 1$$

for some b, $b \geq 0$. (One pebble is used to make further moves in the game.) Since each input of $A^{(j)}$ is the root of a complete binary tree of depth $d - j$, it has 2 distinct complete binary trees of depth $d - j - 1$.

Before output v's j-critical event can occur we have shown that $2^j - \lfloor a/2 \rfloor$ inputs at level d - j or $2(2^j - \lfloor a/2 \rfloor)$ distinct complete binary trees in $B^{(d-j-1)}$ of depth d-j-1 have to be pebbled. Of the $2^{j+1} - a$ such trees at most b + 1 have pebbles on them leaving $2^{j+1} - a - b - 1$ trees of height d - j - 1 to be pebbled. Since $a + b \leq 2^j - 1$ at least $\underline{2^j}$ complete binary trees in $B^{(d-j-1)}$ of depth d - j - 1 each will have to be pebbled before v has its j-critical event.

In the pebbling of $F^{(d)}$ at least 2^d j-critical events must occur since each output is pebbled. Some of these events may occur at the same time. We identify a subset C of the j-critical events which occur at distinct times and count the number of moves necessary between these events. The first j-critical event is a member of this subset, as is the first subsequent j-critical event that is associated with an output with at most $\lfloor a/2 \rfloor$ paths to inputs of $A^{(j)}$ closed at that time. Other members of the subset C are defined in the same manner, that is, in terms of preceding j-critical events. It follows that the subset C contains at least $2^{d-(j+1)}$ j-critical events if $j \leq d - 1$ because at most $2^{j+1} - 1$ j-critical events occur between events in this subset. At least one j-critical event occurs if $j \geq d$. We now state

<u>Theorem 3.3</u>

The number of moves necessary to pebble the FFT graph $F^{(d)}$ on $n = 2^d$ inputs using S pebbles satisfies

$$\frac{n^2}{2 \cdot 2^j} + n(j + 3/2) - 2^j$$

where

$$2^{j-1} + d - (j-1) < S \leq 2^j + d - j, \quad j \leq d.$$

Proof

For the first j-critical event of set C to occur 2^{j+1} trees in $B^{(d-j-1)}$ of depth d - j - 1 that are ancestors to the first output must be pebbled at least once. This requires $2^{j+1}(2^{d-j} - 1)$ moves. For the $2^{d-(j+1)} - 1$ or more outputs in the set C, we have seen, that at least 2^j complete binary trees in $\{B^{(d-j-1)}\}$ have to be pebbled between any two successive j-critical events. This requires at least $2^j(2^{d-j} - 1)$ moves each. There are a total of $2^d(j+1)$ vertices in $A^{(j)}$ and at least so many moves on them. The total number of moves T necessary to pebble $F^{(d)}$ is therefore at least equal to the sum of the number of moves necessary on $B^{(d-j)}$ before the first critical event, the number of necessary moves on $B^{(d-j)}$ between the subsequent $2^{d-j-1} - 1$ j-critical events of set C and the number of moves necessary on the $A^{(j)}$ graphs.

$$T \geq 2^{j+1}(2^{d-j} - 1) + 2^j(2^{d-j} - 1)(2^{d-j-1} - 1) + 2^d(j+1)$$

$$\geq \frac{n^2}{2 \cdot 2^j} + n(j + 3/2) - 2^j$$

where $n = 2^d$.

Q.E.D.

Comparison of this result with the upper bound of Theorem 3.1 indicates that it is weaker by a factor of 4, for large n.

3.5 Final Remarks

The upper and lower bounds derived so far have utilized only the topological properties of the FFT graph. If more details of the FFT computation are included in the graph it should be clear that both the upper and lower bounds on the number of moves required, will increase by the same constant factor. This is because pebbling any vertex of the FFT graph of Fig. 3.1.a corresponds, uniformly, to the computation indicated by the butterfly graph of Fig. 3.1.b, which can be performed with a fixed amount of time and space. Thus there will not be a qualitative change in the Space-Time relationships.

The rather clumsy expressions for the upper and lower bounds can be more succintly expressed as bounds on the quantity (S-d)(T-nd) albeit at some degradation of the gap between the upper and lower bounds.

Since, from Theorem 3.1

$$T \leq \frac{2n^2}{2^j} + n(j-1) - 4/3\,\frac{n^2}{2^{2j}} + \frac{2n}{2^j} - 2/3$$

$$\leq \frac{2n^2}{2^j} + nd + \frac{2n}{2^j} - n$$

Using the inequality

$$S - d + j \leq 2^j$$

we obtain

$$T - nd \leq 2n^2/(S-d+j)$$

or

$$(T-nd)(S-d) \leq 2n^2$$

From Theorem 3.3

$$T \geq n(j+3/2) + \frac{n^2}{2\cdot 2^j} - 2^j$$

or

$$T - nd \geq n(j+3/2-d) + \frac{n^2}{2\cdot 2^j} - 2^j$$

$$(T-nd)\cdot 2^{j-2} \geq \frac{n^2}{8} + n(j + \frac{1}{2} - d)\cdot 2^{j-2}$$

Since

$$2^{j-1} + d - (j-1) < S$$

$$2^{j-2} < S - d$$

We obtain

$$(T-nd)(S-d) \geq \frac{n^2}{8} + n(j + \frac{1}{2} - d)\ 2^{j-2}$$

The minimum for the right hand side of the expression occurs at about $j = d - 2$, when it is equal to $-\frac{3n^2}{32}$.

$$(T-nd)(S-d) \geq \frac{n^2}{8} - \frac{3n^2}{32}$$

Thus

$$2n^2 \geq (T-nd)(S-d) \geq \frac{n^2}{32}$$

The above bounds are much weaker than the upper and lower bounds of Theorems 3.1 and 3.3 respectively.

3.6 Conclusion

The FFT algorithm on n inputs is an example of an important straight-line algorithm for which the Space-Time tradeoffs can be derived on an entire range of values of Space and Time. The gap between these bounds is fairly small and the upper bound derived is based on a method that is practically feasible.

We have shown that if the amount of Space S used is less than $o(n/\log n)$ the amount of time necessary grows faster than $O(n\log n)$, the minimum time necessary with no limit on space. If $S = S_{min}$, the minimum space necessary, is used the time grows as $O(n^2)$. Thus the savings offered by the use of the FFT is possible only when the amount of space available is comparable to the number of inputs of the FFT.

Chapter 4

Space-Time Tradeoffs for Linear Recursion

4.1 Introduction

Recursion is an important feature of many high-level programming languages. Its importance lies in the fact that many algorithms and functions from a variety of applications are most succintly expressed by a recursive definition [23].

Linear Recursion is an important class of recursive procedures in which the value of a recursively defined function at any point in its domain can be computed from the value of the function at most one another point in its domain.

The chief feature of the implementation of a recursive procedure by a compiler is a stack in which records are stored. The maximum size of the stack as measured by the number of records and the number of recursive calls of the procedure which occurs during a computation serves as space and time parameters respectively of the computation. In the usual implementation [32] of linear recursion by a system the size of the stack grows linearly with the number of recursive calls made and may in fact occupy much more storage space than the size of the input itself.

In this chapter we investigate a general method to reduce the storage space on the stack required by a linear recursive computation. This is achieved at the expense of a number of recursive calls that is greater than the minimum necessary when the stack size in unlimited.

We use the pebble game used in Chapters 2 and 3 to model a linear recursive computation. This enables us to obtain a Space-Time tradeoff. We model a linear recursive computation by a pebble game played on directed acyclic graphs. We present, in simple terms, an analysis of the space-time tradeoff which allows us to determine the class of optimal algorithms for the entire range of values of space and time. We also provide simple implementations of some of the optimal algorithms.

Section 4.2 describes, with examples, general linear recursive procedures and their implementation. Section 4.3 derives a graph model of a linear recursive computation and shows that a pebble game on this graph adequately models the space-time tradeoff for the computation. Section 4.4 describes an optimal pebbling strategy which is used in Section 4.5 to derive an expression for the space-time tradeoff. In Section 4.6 we introduce partial stack algorithms which have the optimal space and time parameters derived in Section 4.5. Section 4.7 is an analysis of the functional behavior of the space-time tradeoff in different regions of the space-time domain. We show that if n is the depth of recursion and $T_p(n)$ the number of recursive calls made when the stack size is at most p, then

$$T_p(n) \simeq \begin{cases} p\, n^{1+1/p}(p/1+p) & p \ll \log_2 n \\ k_1\, n \log_2 n & p = k_2 \log_2 n \\ n \log_2 n/\log_2 p & p \gg \log_2 n \end{cases}$$

where k_1 and k_2 are constants.

The analysis indicates that favorable tradeoffs can be made over a wide range of space and time. For example it is shown that space can be reduced to about $\sqrt{2n}$ from a nominal value of n in exchange for an increase of time from n to 2n. Paterson and Hewitt [12] showed that for a fixed value of space p, $T_p(n) \simeq n^{1+1/p}$. Chandra [18] proved it to be a lower bound and provided non-optimal algorithms for various values of space.

4.2 Linear Recursion

The above definition implies that when a linear recursive procedure is invoked for any input in its domain a sequence of recursive calls of the peocedure are generated until a single terminal procedure call is reached. The single terminal procedure call ends all further procedure calls and initiates a sequence of iterative computations involving non-recursive functions found in the body of the procedure.

To illustrate linear recursive procedures we use the format of Program Schemata [20]. Informally, program schemas are procedures which use symbols to denote functions and predicates.

Let p and q denote predicates, f, g and h denote functions and let R be a non-recursive procedure.

<u>Example 4.1</u>

```
F(x):= if p(x) then R(F(f(x))) else if q(x) then g(f(x))
       else F(g(f(x))) fi fi
```

F is a linear recursive procedure with input variable x. We observe that F(x) calls at most one of F(f(x)) or F(g(f(x))). The terminal procedure call of F(f(x)) or F(g(f(x))) is also the terminal call of F(x).

Example 4.2

```
F(x):=  if p(x) then F(f(x)) else if q(x) then G(f(x)) else
        R(x) fi fi
G(x):=  if p(f(x)) then G(f(x)) else G(g(x)) fi
```

Here the two procedures F(x) and G(x) are linear recursive procedures where F(x) may call G(x). However both F(x) and G(x) have single terminal calls.

A general schema for linear recursion [12], which we use through the rest of the chapter, is

```
F(x):=  if p(x) then h(x) else g(x, F(f(x))) fi
```

Here F(x) calls itself until the predicate allows a terminal call of F to compute the function h. Then the computation sequence is reversed and F(x) is computed with no further procedure calls of F.

To illustrate this let us consider the sequence of procedure calls for an input a in the domain of F. Let $f^{(0)}(a) = a$ and for $r \geq 1$ $f^{(r)}(a) = f(f^{(r-1)}(a))$. The smallest integer $n - 1$ such that $p(f^{(n-1)}(a))$ is TRUE, is called the depth of recursion. It follows that

$$F(f^{(n-1)}(a)) = h(f^{(n-1)}(a))$$

and the sequence of procedure calls of F is

$$F(a),\ F(f(a)),\ F(f^{(2)}(a)),\ \ldots,\ F(f^{(n-1)}(a)). \qquad (1)$$

To obtain $F(a)$ we compute in order the sequence

$$F(f^{(n-1)}(a)),\ F(f^{(n-2)}(a)),\ \ldots,\ F(a) \qquad (2)$$

without any further recursive calls of F by using the relation

$$F(f^{(r)}(a)) = g(f^{(r)}(a),\ F(f^{(r+1)}(a))),\quad 0 \leq r \leq n-2$$

with

$$F(f^{(n-1)}(a)) = h(f^{(n-1)}(a))$$

We observe that when the arguments of F are compared the order of sequence (2) is the exact reverse of the order of the sequence (1) of F.

The standard implementation of the procedure F(x) uses a stack to store <u>records</u> of procedure calls in a last-in-first-out manner, that is the record of the last procedure call is at the top of the stack. The <u>record</u> of a procedure call is a portion of the memory stack that stores the fact that the procedural call has been made together with the relevant parameters of the procedure call. These may include values of local variables used in the procedure body. The size of a record (the amount of memory it occupies) is determined by the details of the procedure body and not by the values of parameters of a particular procedure call. Hence it is possible to treat these records as nominal units of space. The space parameter of the computation which is the maximum space occupied by the stack during the computation is therefore expressed as the maximum number of such records in the stack during the computation. The actual memory space being used may be many times larger.

Some important programming languages like FORTRAN do not allow recursive procedures and may instead convert them into a functionally equivalent iterative procedure. The iterative procedure is still implemented using a stack [21] whose size depends on the number of iterations carried out. Hence the space-time tradeoffs are still applicable. The only exception is when the function f(x) is invertible. The following implementation shows that a stack is unnecessary.

```
Proc F(x)
x := a
While $p(x) \neq$ TRUE do x := f(x) od
z := h(x)
while $x \neq a$ do x := $f^{-1}(x)$; z := g(x,z) od
F := z
end
```

4.3.1 A graph model of Linear Recursion

In this section we show that the pebble game of Chapter 2 can adequately model any linear recursive computation on a suitably chosen graph. The graph model and the pebble game are useful in computing the space-time tradeoff.

Figure 4.1 shows a directed acyclic graph L_n called a chain which will be used to model a linear recursion computation. The graph has n vertices corresponding to the depth of recursion n-1 of the procedure F(a). For $1 \leq r \leq n$, vertex r represents a call of F on the argument $f^{(r-1)}(a)$ and the directed edge from node r to node r+1 indicates that the argument $f^{(r)}(a)$ can be computed only after the argument $f^{(r-1)}(a)$ has been computed. The sequence of vertices of the graph L_n represents the computation of the arguments for the successive recursive calls of $F(a)$.

In the pebble game played on the graph L_n, the object is to pebble the vertices of the graph in the decreasing order of their indices subject to the constraints of the direction of the edges of the graph. This corresponds to the rules of the pebble game introduced in Section 2.2. We call this pebbling in <u>reverse order</u>. Pebbling L_n in reverse order corresponds to the computation of the sequence $F(f^{(n-1)}(a))$, $F(f^{(n-2)}(a))$, ..., $F(a)$ using the relation $F(f^{(r-1)}(a)) = g(f^{(r-1)}(a), F(f^{(r)}(a)))$, $1 \leq r \leq n-1$. This computation occurs <u>after</u> the sequence of calls of $F(x)$ have been terminated.

We interpret placing a pebble on a vertex r, $1 \leq r \leq n$, to indicate a recursive call of $F(x)$ at the argument $f^{(r-1)}(a)$. The rules of the pebble game (Chapter 2) make it impossible to pebble the rth vertex, $2 \leq r \leq n$, in reverse order unless a pebble is already stored on the r-1st vertex. This is in accord with the recursive computation of $F(a)$ where a call of $F(f^{(r-1)}(a))$ can be invoked only during a computation of $F(f^{(r-2)}(a))$. A computation of $F(f^{(r-2)}(a))$ can be initiated either by a new recursive call of $F(f^{(r-2)}(a))$ which corresponds to pebbling the r-1st vertex of L_n, or by a record of a call of $F(f^{(r-2)}(a))$ on the top of the stack made earlier in the computation which corresponds to a pebble being stored on the (r-1)st vertex at some point earlier in the game.

In the first case a new call of $F(f^{(r-2)}(a))$ can only be invoked during a computation of $F(f^{(r-3)}(a))$ which may be initiated by a call of $F(f^{(r-4)}(a))$ and so on until for some $k \geqslant 1$, a record of a call of $F(f^{(r-k-1)}(a))$ is on the top of the stack. Thus there is a choice between storing a record of $F(f^{(r-2)}(a))$ and initiating a sequence of at least k successive calls of F corresponding to placing pebbles on the vertices $r-k+1$, $r-k+2$, ..., $r-1$ in that order.

Thus a tradeoff between storing a pebble on a vertex and a sequence of pebble placements mirrors the tradeoff between storing a record of a procedure call of F in the stack and initiating a sequence of procedure calls of F.

4.3.2 An Optimal Pebbling Strategy

In the last section we modeled a linear recrusive computation of depth n by a reverse pebbling of the vertices of the graph L_n of Fig. 4.1. In this game the number of pebbles used, p, and the number of times pebbles are placed on vertices, $T_p(n)$, represent the space and time parameters of the linear recursive computation.

In this section we describe a property of optimal pebbling strategies that pebble in reverse order the graph L_n. This property is used to express $T_p(n)$ as a recurrence relation which is solved in Section 4.5. The modified rule of the pebble game (Chapter 3) is used below.

We consider the case when a single pebble is used in the game. This implies that a single record of a recursive call may be stored on the stack at any one time. To pebble vertex n of L_n it is necessary to move the pebble successively on vertices 1, 2, ... n. Pebbling vertex n-1, n-2, ... in that order is the same as pebbling L_{n-1} in reverse order. Thus

$$T_1(n) \quad = \quad n + T_1(n-1)$$

and since $T_1(1) = 1$, we have

$$T_1(n) \quad = \quad \frac{n(n+1)}{2}$$

If n pebbles are used we need to place and store a pebble on the vertices in the order 1, 2, ..., n. Pebbling in reverse order is trivially accomplished since the vertices already have pebbles on them. Thus if vertex r has a pebble on it when it is visited the visitation is not counted in $T_p(n)$. This reflects the fact that in a linear recursive computation a procedure call for an argument is not made if a record of an earlier call for the same argument is available at the top of the stack. Thus

$$T_n(n) \quad = \quad n$$

This case corresponds to a linear recursive computation with unlimited space.

Consider the general case when L_n is pebbled using p pebbles, $p \geq 2$. We shall say that a gap has developed at vertex r, $1 \leq r \leq n-2$, if during the game pebbles are placed on vertices r, r+1, r+2, ... in that order with a pebble stored on vertex r but not on vertex r+1.

We claim that in an optimal strategy, if a gap develops at vertex r, the pebble stored on vertex r is not removed until vertex r+1 is pebbled in reverse order. Otherwise, let r+k, $2 \leq k \leq n-r$ be the last vertex pebbled in reverse order after which the pebble on vertex r is removed. Let r-d, $d \geq 1$ be the closest predecessor of vertex r which has a pebble stored on it at the time the pebble on vertex r is removed. Since by assumption, vertex r+k has been pebbled in reverse order the vertices r-d+1, ..., r+k-1 which do not have a pebble stored on any one of them must be pebbled in reverse order.

We consider a new strategy almost identical to the optimal strategy described above except that the pebble stored on vertex r is stored on vertex r+k-1 instead. No additional placements of pebbles are required by the new strategy as compared to the optimal strategy. Since vertex r+k-1 has a pebble stored on it when vertex r+k is pebbled in reverse order the new strategy has to pebble the vertices r-d+1, r-d+2, ..., r+k-2 in reverse order, which needs at least one placement of pebble less than the above optimal strategy, which is a contradiction.

<u>Lemma 4.1</u>

In an optimal strategy using p pebbles there is a point in time when the pebble on the vertex of lowest index, say r, is held in place while the remaining p-1 pebbles are used to pebble in reverse order the subgraph L_{n-r} consisting of the vertices r+1, r+2, ..., n.

Since vertex r has a pebble stored on it when vertex r+1 is pebbled in reverse order, no additional placement of a pebble on r is necessary.

The p pebbles are used to pebble the r-1 vertices 1, 2, ..., r-1 in reverse order corresponding to a graph L_{r-1}. We need r placements of pebbles to store a pebble on vertex r. Vertex r is called a <u>splitting vertex</u>. Since r has been chosen optimally we have

$$T_p(n) = \min_{1 \leq r \leq n-p+1} (r + T_p(r-1) + T_{p-1}(n-r)) \qquad (3)$$

4.4 A Time-Space Tradeoff

We have derived in Section 4.3 a recurrence relation for $T_p(n)$, the minimal number of recursive calls necessary to compute F(a) when the stack size is no larger than p. In this section we solve this relation using a binomial number system. The analysis also indicates the entire class of optimal pebble placements possible. Lemma 4.2 illustrates a suitable choice for the location of the splitting vertex.

To solve the recurrence we introduce a binomial number system. Given a positive integer $p \geq 2$ (to be interpreted later as the number of pebbles), for each positive integer N there are unique non-negative integers m and ℓ such that

$$N = S_{p-1,m} + \ell , \qquad 0 \leq \ell \leq S_{p-2,m+1} - 1$$

where

$$S_{q,m} = \binom{m+q}{q+1}$$

The uniqueness of these integers follows from the monotonicity of $S_{p-1,m}$ with m and the following identity

$$S_{q,m+1} = S_{q,m} + S_{q-1,m+1} \tag{4}$$

The number system can be extended to the case p=1, which is important below, if we set $S_{-1,m} = 1$. Then, when p=1 we have $\ell = 0$ and $m = N$. Also, $S_{p-1,2} = p+1$ so if $p \geq N$ we have m=1 and $\ell = N-1$.

Theorem 4.1

For all $p \geq 1$, the minimum number of placements of pebbles required to pebble the chain L_n of $n \geq 1$ nodes with at most p pebbles, $T_p(n)$, satisfies

$$T_p(n) = \frac{p}{p+1} (m-1)\, S_{p-1,m} + m(\ell+1) \qquad (5)$$

where $1 \leq m$ and $0 \leq \ell \leq S_{p-2,m+1} - 1$ are the unique integers such that

$$n = S_{p-1,m} + \ell \qquad (6)$$

Proof

The proof is by induction on n and p.

Basis

a) The case of $p=1$, namely,

$$T_1(n) = n(n+1)/2 = m(m+1)/2$$

has been established above which agrees with (5).

B) For $p \geq n$, L_n can be completely pebbled in n moves so $T_p(n) = n$. Also, $m=1$ and $\ell = n-1$ in this case, which agrees with (5).

The basis states expressions for $T_p(n)$ on the boundaries which are shown in Figure 4.2.

Inductive Hypothesis

If, $T_p(n)$ is given by (5) for all $1 \leq n \leq p$ when $p \leq P-1$ and for $1 \leq n \leq N-1$ when $p = P$, then $T_P(N)$ is also given by (5).

Figure 4.2 also shows the order in which the induction sequence is carried out. We now state the conditions under which the minimum of equation (3) is achieved. Let

$$G(r) = r + T_p(r-1) + T_{p-1}(n-r) \tag{7}$$

Then,

$$T_p(n) = \min_{1 \leq r \leq n-p+1} G(r) \tag{8}$$

Consider the forward difference

$$\begin{aligned} \nabla G(r) &= G(r+1) - G(r) \\ &= 1 + \nabla T_p(r-1) - \nabla T_{p-1}(n-r-1) \end{aligned} \tag{9}$$

where

$$\nabla T_p(j) = T_p(j+1) - T_p(j)$$

The minimum in (8) is achieved at a value of r such that $\nabla G(r) \geq 0$. Therefore, we further evaluate $\nabla G(r)$.

Since $1 \leq r \leq n - p + 1$ and $p \geq 2$ we invoke the inductive hypothesis and use (5) to evaluate the forward differences in (9). To do this we let (u,h) where $u \geq 0$, $0 \leq h \leq S_{p-2,u+1} - 1$ and (v,i) where $v \geq 0$, $0 \leq i \leq S_{p-3,v+1} - 1$, be the unique pairs of integers such that

$$\begin{aligned} r &= S_{p-1,u} + h \\ n - r &= S_{p-2,v} + i \end{aligned} \tag{10}$$

Clearly, from (6) we have

$$S_{p-1,m} + \ell = S_{p-1,u} + S_{p-2,v} + h + i \tag{11}$$

which will be used later.

The forward difference $\nabla T_p(r-1)$ is easily seen to be u when $h \geq 1$ (and $r-1 \geq S_{p-1,u}$) and can also be shown equal to u when $h = 0$ by straightforward manipulation of binomial coefficients. Similarly $\nabla T_{p-1}(n-r-1)$ is equal to v. Summarizing we have

$$\nabla T_p(r-1) = u \,, \quad 0 \leq h \leq S_{p-2,u+1} - 1$$

$$\nabla T_{p-1}(n-r-1) = v \,, \quad 0 \leq i \leq S_{p-3,v+1} - 1$$

We will also encounter the forward differences $\nabla T_p(r-2)$ and $\nabla T_{p-1}(n-r)$ and we have

$$\nabla T_p(r-2) = \begin{cases} u & h > 0 \\ u-1 & h = 0 \end{cases}$$

$$\nabla T_{p-1}(n-r) = \begin{cases} v+1 & i = S_{p-3,v+1} - 1 \\ v & i < S_{p-3,v+1} - 1 \end{cases}$$

as a direct consequence of the above analysis.

From these observations and (9) we have that

$$\nabla G(r) = 1 + u - v \tag{12}$$

and since

$$\nabla G(r-1) = 1 + \nabla T_p(r-2) - \nabla T_{p-1}(n-r)$$

we have

$$\nabla G(r-1) = \begin{cases} u - v & h > 0, \quad i = S_{p-3,v+1} - 1 & \text{(13a)} \\ 1 + u - v & h > 0, \quad i < S_{p-3,v+1} - 1 & \text{(13b)} \\ u - v - 1 & h = 0, \quad i = S_{p-3,v+1} - 1 & \text{(13c)} \\ u - v & h = 0, \quad i < S_{p-3,v+1} - 1 & \text{(13d)} \end{cases}$$

As indicated above, the integers r which minimize (8), the optimal splitting nodes, satisfy $\nabla G(r) \geq 0$ although not all such integers minimize this expression. We consider two classes of integers r which minimize (8) and satisfy $\nabla G(r) \geq 0$, namely,

$$A = \{r, r+1 \mid \nabla G(r) = 0\}$$

$$B = \{r \mid \nabla G(r) \geq 1, \nabla G(r-1) \leq -1\}$$

The integers for which $\nabla G(r) \geq 1$ and $\nabla G(r-1) \geq 1$ do not minimize (8) while those for which $\nabla G(r) \geq 1$ and $\nabla G(r-1) = 0$ fall into A.

Consider $r \in A$ such that $\nabla G(r) = 0$; then from (12)

$$\nabla G(r) = 1 + u - v = 0$$

or $v = u + 1$. From (11) and the identity (4) we have

$$S_{p-1,m} + \ell = S_{p-1,u+1} + (h+i)$$

and since

$$0 \leq h + i \leq S_{p-2,u+1} - 1 + S_{p-3,v+1} - 1 = S_{p-2,u+2} - 2$$

we conclude that

$$u = m - 1, \; v = m, \; 0 \leq h + i = \ell \leq S_{p-2,m+1} - 2 \qquad (14)$$

Consider next the case of $r \in B$. Here we have $u - v \geq 0$ and $\nabla G(r-1) \leq -1$ and the only case for which both conditions hold is (13c). This requires

$$u = v, \qquad h = 0, \qquad i = S_{p-3,v+1} - 1$$

and from (4) and (11) we have

$$S_{p-1,m} + \ell = S_{p-1,u} + S_{p-2,u} + S_{p-3,u+1} - 1$$

$$= S_{p-1,u} + S_{p-2,u+1} - 1$$

from which it follows that

$$u = m, \quad v = m, \quad \ell = S_{p-2,m+1} - 1, \quad h = 0, \quad i = S_{p-3,m+1} - 1 \tag{15}$$

Thus, if $\ell = S_{p-2,m+1} - 1$ there is exactly one value for r that minimizes G(r), namely, $r = S_{p-1,m}$, while if $0 \leq \ell \leq S_{p-2,m+1} - 2$ then the minimizing value of r satisfies $S_{p-1,m-1} \leq r \leq S_{p-1,m} - 1$.

By simple manipulation of binomial coefficinets we can verify that the minimum is indeed $T_p(n)$.

Q.E.D.

We extract some additional information from this theorem that will facilitate the construction of a partial stack algorithm for linear recursion.

<u>Corollary</u>

If r_o is a splitting node of L_n then it satisfies the following conditions:

Case a)

If $S_{p-1,m} \leq n \leq S_{p-1,m+1} - 2$, then

$$S_{p-1,m-1} \leq r_o \leq S_{p-1,m} - 1, \quad S_{p-2,m} \leq n - r_o \leq S_{p-2,m+1} - 1$$

Case b)

If $n = S_{p-1,m+1} - 1$, then

$$r_o = S_{p-1,m} \quad \text{and} \quad n - r_o = S_{p-2,m+1} - 1$$

We now identify a single, simply computed integer r_1 which is the label of a splitting node of L_n.

Lemma 4.2

The integer r_1 defined by

$$r_1 = \max(S_{p-1,m-1} , n - S_{p-2,m+1} + 1) \qquad (16)$$

satisfies the conditions of the above corollary.

Proof

The conditions of the corollary can be stated as bounds on r_o, when $S_{p-1,m} \leq n \leq S_{p-1,m+1} - 2$, as shown below.

$$S_{p-1,m-1} \leq r_o \leq S_{p-1,m} - 1, \quad n - S_{p-2,m+1} + 1 \leq r_o \leq n - S_{p-2,m}$$

It is easy to demonstrate that r_1, the larger of the two lower bounds, satisfies both upper bounds. When $n = S_{p-1,m+1} - 1$, $n - S_{p-2,m+1} + 1 = S_{p-1,m}$ so that $r_1 = S_{p-1,m}$, which is the optimizing value of r_o in this case.

Q.E.D.

4.5 Partial Stack Algorithms

In this section we discuss realization of the space-time exchanges of Section 4.4 by a class of recursive algorithms called Partial Stack Algorithms. Essentially, a partial stack algorithm stores a record of a recursive call of $F(f^{(r-1)}(a))$, where r is a splitting vertex. In Section 4.3 we saw that the splitting vertex divided the reverse pebbling of L_n into the reverse pebbling of graphs L_{r-1} and L_{n-r} which have their own respective splitting vertices. Thus a partial stack algorithm that uses a stack size of at most p records stores records of recursive calls for arguments of F which correspond to splitting vertices of L_n. The analysis indicates that there are no more than p splitting vertices that have to be stored at any given time. The overhead involved in implementing a partial stack algorithm consists of computing the locations of the splitting vertices. This is discussed in Section 4.6.

The partial stack algorithm PSTK that uses splitting vertices defined by Lemma 4.2 as

$$r = \max(S_{p-1,m-1} , n - S_{p-2,m+1} + 1)$$

is described below in pseudo-ALGOL. The algorithm needs the depth of recursion, n, as an input parameter. The depth of recursion can be obtained by the iterative algorithm DEP (a). We describe below in detail all the algorithms that are used.

DEP(a) DEP is a non-recursive procedure which computes the depth of recursion n for a given input a. This is obtained by iteratively testing the predicate p(x) for the arguments a, f(a), $f^2(a)$, ... until $p(f^n(a))$ is found to be TRUE. The value n is returned.

SPL(a) This computes the recursively defined function F at the value a using a stack of height one. This is done by iteratively computing $f^n(a)$, for which $F(f^n(a)) = h(f^n(a))$. The algorithm then computes $f^{(n-1)}(a)$ from which it obtains $F(f^{(n-1)}(a)) = g(f^{(n-1)}(a), F(f^n(a)))$. Thus the value of F for $F(f^r(a))$, $r \geqslant 0$ are computed according to a decreasing value of r.

STK(a) This is a procedure which uses a stack of height n + 1, where n is the depth of recursion. The procedure tests the predicate for each of the arguments a, f(a), $f^2(a)$, ... in turn. A record of a procedure call is created each time the test fails. When the stack grows to a height of n + 1, where n is the depth of recursion, the predicate is satisfied. The value of the linear recursive function F is computed immediately for the argument found in the record on the top of the stack since $F(f^n(a)) = h(f^n(a))$, and h is non-recursive. The record is popped and $F(f^{(n-1)}(a))$ is computed using the argument stored in the record on top of the stack using the function g as in SPL(a). After n + 1 such computations, the value of F at the argument a is obtained while at the same time the stack height decreases to 0.

PSTK(A,n,p,m,s,F) This is the procedure that implements the partial stack algorithm, corresponding to pebbling L_N using p pebbles. The procedure first determines the value of N = n + 1. N represents the number of vertices in the Linear Recursion graph of Section 4.3.

On the basis of the value of the input p, the maximum stack height allowed, the special cases of $p = N$ and $p = 1$ are checked to see if they apply, and invoked if they do. Otherwise, the procedure chooses a splitting node, r, according to Lemma 4.2. The splitting node is taken to be the maximum of $S_{p-1,m-1}$ (denoted by $s\ell$) and $n - S_{p-2,m+1} + 1$ (denoted by u). $s\ell$ and u are computed from the simple formula $s\ell = s(m-1)/(m+p-1)$ and $u = N - s(p/m) + 1$. Here s is such that $s \leq N \leq s(m+p)/m - 1$ (See proof of Theorem 4.1).

Once r, the splitting node, has been chosen, the remaining $N - r$ vertices have to be computed recursively using a stack of height $p - 1$. Since $S_{p-2,m} \leq N - r \leq S_{p-2,m+1} - 1$, $S_{p-2,m}$ (denoted by su) is computed using the formula $su = sp/(m+p-1)$. Thus su replaces s in the recursive computation of $L_{(n-r)}$ using a stack of height of $p - 1$.

Before computing $L_{(n-r)}$ recursively, a sequence of r procedure calls of F starting at input a is initiated. However no record of any of these calls is stored on the stack. This corresponds to forward pebbling up to the splitting vertex r. A call of PSTK, corresponding to pebbling the graph L_{n-r}, is now made, since all necessary inputs have been calculated.

A second call of PSTK, corresponding to pebbling L_{r-1} using p pebbles, is now initiated. The inputs that are necessary are $r - 1$ (denoted by $N\ell$), p, and $s\ell$, where $s\ell$ replaces s, according to Theorem 4.1.

1. Procedure DEP (a)

[DEP(a) determines the depth of recursion n for the input a.]

```
x := a
n := 0
while NOT(p(x)) do x := f(x); n := n+1 od
Return (n)
end
```

2. Procedure SPL(a)

[Procedure SPL computes F with a stack size of one.]

```
begin
for i := n-1 until 1 step - 1 do
x := a
for j := i until 1 step - 1 do
x := f(x) od
F := g(x,F) od
F := g(a,F)
end
```

3. Procedure STK(a)

[Procedure STK uses a stack height of n+1, the depth of recursion being n, to compute S at the value a.]

```
    begin stack A; "Clear A"; x := a
```

[A is declared to be a variable of type STACK. A := x and x := A, denote PUSH and POP operations respectively.]

```
    while NOT p(x) do A := x; x := f(x) od
    F := h(x)
    while |A| ≠ 0 do x := A; F := g(x,F) od
    end
```

4. Procedure PSTK(a,n,p,m,s)

[PSTK is a partial stack algorithm that computes F at the value a given n the depth of recursion, p the stack height allowed.]

```
    N := n + 1.
    If p=1 then SPL(a) else if p=N then STK(a) fi fi
```

[Takes care of special cases p=1 and p=N]

```
    sℓ := s(m-1)/(m+p-1)
    u := N - s(p/m)+1
```

[sℓ and u are the lower and upper bounds respectively of the splitting vertex r, according to Lemma 4.2.]

```
su := sp/(m+p-1)
pu := p-1
mℓ := m - 1
```

[su = $S_{p-2,m} \leq (N-r) \leq S_{p-2,m+1} - 1$. Thus N-r vertices, su $\leq$ N-r, will be pebbled using pu = p-1 pebbles.]

```
    If sℓ ≥ u then r := sℓ ; mℓ := mℓ-1; sℓ := mℓ(sℓ)/(mℓ+p)
    else r := u ; fi
```

[The splitting vertex r = max(sℓ,u) is chosen.]

```
    i := r - 1
    Nℓ := r - 1
    Nu := N - r
     x := a
    while i ≠ 0 do x := f(x) ; i := i - 1 od
     z := f(x)
```

[A forward pebbling up to the splitting vertex is carried out.]

```
    PSTK (z, Nu, pu, m, su)
```

[Vertices r+1, r+2, ..., N are pebbled in reverse order.]

```
    F := g(x,F)
```

[F is computed at the splitting vertex.]

```
    If Nℓ := 0 then F := g(x,F) else PSTK (a, Nℓ, p, mℓ, sℓ) fi
```

[If the recursion is ended F is computed at the value a. Otherwise the recursion is continued on the remaining vertices.]

3.6 Implementation Issues

Theorem 4.1 shows that $T_p(n)$, the number of pebbling moves needed to pebble L_n, is a linear function of n in the range between $S_{p-1,m'}$, $m' = 1, 2, 3, \ldots$. For $n = S_{p-1,m}$ we have

$$T_p(n) = \frac{p}{p+1}(m-1)n$$

For $p = 2$, $n \approx \sqrt{2n}$ and

$$T_p(n) = (2\sqrt{2}/3)n^{3/2}, \text{ for large n.}$$

Similarly when $m = 3$, $n = (p+2)(p+1)/2$, $p \approx \sqrt{2n}$ and

$$T_p(n) \approx 2n \qquad \text{for large n}$$

This indicates that very favorable space-time exchanges are possible without too large an increase in $T_p(n)$.

Since the time, $T_p(n)$, needed to pebble L_n is dependent on m it is of interest to examine the relationship of the numbers n, p and m. When $n = S_{p-1,m}$, we can show from the following inequality [22, p.530], that the smaller of p and (m-1) is no larger than $2 \log_2 n$ when $n \geq 4$.

$$\frac{1}{\sqrt{2N}} \leq \sqrt{\frac{N}{8k(N-k)}} \leq \binom{N}{k} 2^{-NH(\frac{k}{N})} \leq \sqrt{\frac{N}{2\pi k(N-k)}} \leq 1 \qquad (17)$$

Here $1 \leq k \leq N-1$, $N \geq 2$ and $H(x)$ is the entropy function and $N = m + p - 1$.

$$H(x) = -x \log_2 x - (1-x) \log_2 (1-x)$$

and

$$N = m + p - 1$$

Since n is one term in the binomial expansion of $(1+1)^{m+p-1}$, we have

$$n \leq 2^{m+p-1}$$

or that the sum m+p-1 is at least $\log_2 n$. Furthermore, from (17) if m and p are comparable in size and n is large, it follows that they are both comparable to $\log_2 n$. Thus, we consider three cases when n is large

$$m << p \rightarrow m << \log_2 n$$

and

$$p << m \rightarrow p << \log_2 n$$

and

$$p \text{ comparable to } m \rightarrow p/(m+p-1) = \lambda, \ 0 < \lambda < 1$$

We examine $T_p(n)$ below.

We have

$$n = \frac{(m+p-1)(m+p-2)\ldots(m)}{p!} \geqslant \left(\frac{m}{p}\right)^p$$

which is also a good approximation when p << m. This implies

$$m \lesssim pn^{1/p}$$

and by the symmetry of n in (m-1) and p we have

$$p+1 \lesssim (m-1)n^{1/(m-1)}$$

which is a good approximation when m << p. Reworking this equation we have

$$(m-1) \lesssim \frac{\log_2 n}{\log_2 p - \log_2 (m-1)} \simeq \frac{\log_2 n}{\log_2 p}$$

and the approximation holds when $m \ll p$. Since $m \ll p$ if $p \gg \log_2 n$ (note that $m+p-1 \geq \log_2 n$), we have

$$T_p(n) \approx \begin{cases} \frac{p^2}{p+1} n^{1+1/p} & p \ll \log_2 n \\ \frac{n\log_2 n}{\log_2 p} & p \gg \log_2 n \end{cases}$$

In the remaining case, when p is proportional to $\log_2 n$, we use (17) to approximate n. If

$$\lambda = \frac{p}{m+p-1} \quad \text{or} \quad m-1 = \frac{(1-\lambda)}{\lambda} p$$

for $0 < \lambda < 1$ and if n is large, then taking logarithms we have

$$\log_2 n \approx (m+p-1)\, H(\lambda) = p\frac{H(\lambda)}{\lambda}$$

which implies that p is proportional to $\log_2 n$. Then,

$$T_p(n) \approx \frac{1-\lambda}{H(\lambda)}\, n\log_2 n$$

when

$$p \approx \frac{\lambda}{H(\lambda)} \log_2 n$$

The expressions $\lambda/H(\lambda)$ and $(1-\lambda)/H(\lambda)$ are shown in Figure 4.3.

Summarizing, we find that $T_p(n)$ grows as $pn^{1+1/p}$ for p small, $n\log n/\log p$ for $p \gg \log_2 n$ and as $n\log n$ if λ is neither near zero $(p \ll m)$ nor near $1 (m \ll p)$, that is, for p proportional to $\log_2 n$. The three different rates of growth of $T_p(n)$ with n can be selected by choosing p to be a function of n which grows more slowly than $\log_2 n$, such as $\sqrt{\log_2 n}$ or a constant, more rapidly than $\log_2 n$, such as $(\log_2 n)^2$ or $n^{1/5}$, or which grows in proportion to $\log_2 n$, respectively. The actual value of p may also be determined by an upper limit on temporary storage space.

Once p is chosen, the next step is to determine m and $S_{p-1,m}$ such that

$$S_{p-1,m} \leq n < S_{p-1,m+1} \tag{18}$$

By a previous argument, the smaller of p and m-1 is no larger than $2 \log_2 n$ for $n \geq 4$, hence $S_{p-1,m}$ can be computed in at most $8 \log_2 n$ multiplications or divisions from one of the following two expressions:

$$S_{p-1,m} = \frac{(m+p-1)(m+p-2)\ldots(m)}{p!} = \frac{(m+p-1)(m+p-2)\ldots(p+1)}{m!}$$

In fact, many fewer multiplications may suffice if either p or m are very small. To compute m, start m at n (note that $S_{o,m} = m$) and use binary search by halving m until (18) is satisfied. This will take $O(\log_2 n)$ steps so the entire process can be done in $O(\log^2 n)$ steps.

3.7 Conclusion

Linear Recursion is an important class of recursive procedures which has significant space-time exchanges. These exchanges have been derived by modeling the recursive computation by a pebble game played on a chain of n nodes, n being the depth of recursion. The model provides a clear and concise manner of expressing space-time exchanges. The derivation of an expression for these exchanges also allows us to describe the entire class of strategies which are optimal with respect to spaee and time. The chief feature of these algorithms, known as partial stack algorithms, is the use of the stack in storing records of procedure calls at selected vertices of L_n. These vertices known as splitting vertices can each be easily computed at a small extra cost which does not depend on the depth of recursion.

Chapter 5

Space-Time Tradeoff for Sorting Algorithms

5.1 Introduction

We have seen in Chapters 3 and 4 two different methods of obtaining Space-Time tradeoffs for two algorithms of interest. In this chapter we present yet another method, which is used to obtain Space-Time tradeoffs for the problem of Sorting. The chief characteristic of this method is that it does not depend on the connection properties of the graphs of particular oblivious algorithm. Instead it relies on the properties of the function computed and the properties of the operations used in the algorithms that compute them. Thus the merit of this method is that the tradeoffs obtained are applicable to all algorithms of a class defined below. The drawback of the method is that the lower bounds on Space-Time tradeoffs are in general weaker than those that may be obtained by an analysis of a specific algorithm.

We consider in this chapter the problem of sorting a list of n items. The class of oblivious algorithms we consider use only the operations Min(a,b) and Max(a,b), where a and b are elements from the list of n items that is to be sorted and Min(a,b) and Max(a,b) are the minimum and maximum of a and b respectively. We note that the tradeoff obtained is also applicable to other classes of oblivious algorithms which use operations that are monotone mappable [24], that is those operations which can be homomorphically mapped to Min(a,b) and Max(a,b).

If S is the number of items temporarily stored during the execution of an algorithm of the above class, and T the number of operations performed, we show that

$$T \geq \frac{n^2}{S+1} - n + S$$

The method we use is a non-trivial extension of a method due to Grigoriev [19]. Tompa [15], using a different method, has obtained the result $T \geq \frac{n^2}{16S}$ for the problem of oblivious Merging of two sorted lists of n items each.

We consider the class of Sorting algorithms to be all oblivious Sorting algorithms which use the operations Min(a,b) and Max(a,b), where a and b are items from the list that is to be sorted. Some examples of this class of algorithms are Bubble Sort, Batcher's Sort, and Bitonic Sort [2]. In the following it will be easier, for expository purposes, to let the list of items to be sorted be the Boolean constants 0 and 1. The Max(a,b) and Min(a,b) operations can be uniformly replaced by the Boolean OR and AND operations respectively. Since the algorithms are oblivious and since the transformation from Max and Min to OR and AND is 1 - 1, the Space-Time tradeoffs that are derived for algorithms that use OR and AND also apply to those that use Max and Min operations and indeed to algorithms that use any other monotone mappable operations [24].

The pebble game used in the previous chapters is also used here as the model of computation.

Section 5.2 describes the class of algorithms used and gives a proof of the Space-Time tradeoff. Section 5.3 concludes the chapter.

5.2 A Lower Bound

Let $f_1, f_2, \ldots, f_n$ denote the n outputs of a Sorting algorithm, of the class described in Section 5.2.1, which sorts the n inputs $x_1, x_2, \ldots, x_n$. That is, for $1 \leq i \leq n$

$$f_i = \begin{cases} 1 & \text{if number of 1's in } (x_1, x_2, \ldots, x_n) \\ & \text{is } n - i + 1 \text{ or more} \\ 0 & \text{otherwise.} \end{cases}$$

Let S be the maximum number of temporary storage locations that are used and let T be the number of operations performed during the execution of the algorithm.

We divide the set of n outputs $f_1, f_2, \ldots, f_n$ into blocks of $S + 1$ outputs each, such that during the execution of the algorithm the set of outputs in any particular block are computed in sequence in some order. If $S + 1$ does not divide n there will be a block with less than $S + 1$ outputs.

In order to obtain a lower bound on T we again use the notion of a critical event.

Definition 5.1

A Critical Event for sorting a block of outputs computed in sequence is the pebbling of the last output of the sequence.

The lower bound on T will be established by obtaining a lower bound on the number of operations required between two successive critical events.

Let $X = \{x_{j_1}, x_{j_2}, \ldots, x_{j_p}\}$ be a subset of the set of inputs $x_1, x_2, \ldots, x_n$.

Definition 5.2

A p-tuple α in X is defined to be

$$\alpha = (\alpha_{j_1}, \alpha_{j_2}, \ldots, \alpha_{j_p}) \quad \alpha_{j_i} \varepsilon \{0,1\}, \ 1 \leqslant i \leqslant p$$

which is obtained by substituting **the constant** α_{j_i} **for** x_{j_i}, $1 \leqslant i \leqslant p$.

Definition 5.3

Let α and β be two p-tuples in X. Then $\alpha < \beta$ if $\alpha_{j_i} \leqslant \beta_{j_i}$, $1 \leqslant i \leqslant p$, and there exists a k, $1 \leqslant k \leqslant p$, such that $\alpha_{j_k} < \beta_{j_k}$.

We use the notion of a chain in deriving a lower bound on the Time required to compute the Sorting function.

Definition 5.4

A chain of length q in X is defined to be a sequence of q p-tuples in X

$$\alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_q}$$

such that the number of 1's in α_{i_ℓ} is equal to i_ℓ, $1 \leqslant \ell \leqslant q$ and

$$\alpha_{i_1} < \alpha_{i_2} < \ldots < \alpha_{i_q}$$

Lemma 5.1

Consider two successive critical events between which a block of S + 1 outputs are pebbled. Then at least n - S inputs are pebbled between the two critical events.

Proof

Let the set of outputs that are pebbled be

$$f_{i_1}, f_{i_2}, \ldots, f_{i_{S+1}} , \quad i_1 > i_2 \ldots > i_{S+1}$$

(We do not assume that they are necessarily pebbled in the above order).

Let m be the number of inputs that are not pebbled between the two successive critical events. If $m < S + 1$, then $n - m \geq n - S$, inputs are pebbled and the Lemma is proved. Otherwise, $m \geq S + 1$.

Let $X = \{x_{j_1}, x_{j_2}, \ldots, x_{j_{S+1}}\}$ be a set of $S + 1$ inputs chosen from the set of m inputs that are not pebbled, and let $Y = \{x_{k_1}, x_{k_2}, \ldots, x_{k_{n-S-1}}\}$ be the set of inputs not in X. We form chains

$$\alpha_0, \alpha_1, \ldots, \alpha_{S+1} \text{ in } X$$

and

$$\beta_0, \beta_1, \ldots, \beta_{n-S-1} \text{ in } Y.$$

We choose $\beta_{r_1}, \beta_{r_2}, \ldots, \beta_{r_{S+1}}$ from the chain $\beta_0, \beta_1, \ldots, \beta_{n-S-1}$ where

$$r_k = n - i_k - (k-1) , \quad 1 \leq k \leq S + 1.$$

We claim that

$$\beta_{r_1} \leq \beta_{r_2} \leq \ldots \leq \beta_{r_{S+1}}$$

Since $\beta_{r_1}, \beta_{r_2}, \ldots, \beta_{r_{S+1}}$ are elements taken from a chain it is sufficient to prove that

$$r_1 \leq r_2 \ldots \leq r_{S+1}$$

This follows directly from the fact that $i_k > i_{k+1}$. Also we note that

$$0 \leq r_1 \leq n - (S+1)$$

and

$$r_1 \leq r_{S+1} \leq n - (S+1)$$

Therefore $r_k \leq r_{k+1}$ for $1 \leq k \leq S$. Not all of $\beta_{r_1}, \beta_{r_2}, \ldots, \beta_{r_{S+1}}$ need necessarily be distinct (n-S-1)-tuples. However, for convenience, we use distinct symbols. The proof, as will be seen below, makes no assumption regarding the number of distinct (n-S-1)-tuples in the above sequence.

Consider a time instant before the first output, say f_{i_j}, $1 \leq j \leq S+1$, of the seqeence of functions in the given block is pebbled. The S pebbles at this time are stored on vertices of the graph of the algorithm which represent some S functions $P = \{P_1, P_2, \ldots, P_S\}$. $P_1, P_2, \ldots, P_S$ are monotone functions in the inputs $x_1, x_2, \ldots, x_n$, that is in the set of inputs $X \cup Y$.

We use the notation $P\Bigg|_{X = \alpha_{k_1}}^{Y = \beta_{j_1}}$ to denote the S output values of the functions $P_1, P_2, \ldots, P_S$ obtained by setting the inputs in the set Y to β_{j_1} and the inputs in the set X to α_{k_1}, where β_{j_1} and α_{k_1} are elements of the chains described earlier. The same notation is used to denote the output of a single function under the same input conditions as above.

If

$$P\Bigg|_{X = \alpha_0}^{Y = \beta_{r_1}} = P\Bigg|_{X = \alpha_1}^{Y = \beta_{r_1}}$$

then

$$f_{i_1}\Bigg|_{X = \alpha_0}^{Y = \beta_{r_1}} = f_{i_1}\Bigg|_{X = \alpha_1}^{Y = \beta_{r_1}}$$

Since only a change in the value of P can indicate a change in the value of the inputs of the set X, that is from $X = \alpha_0$ to $X = \alpha_1$. But the number of 1's in $X \cup Y = n - i_1$ if $X = \alpha_0$ and $n - i_1 + 1$ if $X = \alpha_1$. Therefore equality cannot hold and we must have

$$P\Bigg|_{X = \alpha_0}^{Y = \beta_{r_1}} \neq P\Bigg|_{X = \alpha_1}^{Y = \beta_{r_1}}$$

This implies that one of $P_1, P_2, \ldots, P_S$ changes from 0 to 1, since they are monotone. Similarly

$$P\Bigg|_{X = \alpha_1}^{Y = \beta_{r_2}} \neq P\Bigg|_{X = \alpha_2}^{Y = \beta_{r_2}}$$

By a similar argument at least two of $P_1, P_2, \ldots, P_S$ have their outputs equal to 1 since $(\alpha_1, \beta_{r_1}) < (\alpha_2, \beta_{r_2})$. Thus for the chain of input values

$$(\alpha_1, \beta_{r_1}) < (\alpha_2, \beta_{r_2}), \ldots, < (\alpha_S, \beta_{r_S}) \text{ in } X \cup Y.$$

all output values of the S functions $P_1, P_2, \ldots, P_S$ must change to 1 in some order. But

$$(\alpha_S, \beta_{r_S}) < (\alpha_{S+1}, \beta_{r_{S+1}})$$

and

$$P\Big|_{X = \alpha_S}^{Y = \beta_{r_{S+1}}} = P\Big|_{X = \alpha_{S+1}}^{Y = \beta_{r_{S+1}}}$$

since $P_1, P_2, \ldots, P_S$ are already equal to 1. This implies that

$$f_{i_{S+1}}\Big|_{X = \alpha_S}^{Y = \beta_{r_{S+1}}} = f_{i_{S+1}}\Big|_{X = \alpha_{S+1}}^{Y = \beta_{r_{S+1}}}$$

which is a contradiction. Thus $m < S + 1$ and at least $n - S$ inputs are pebbled between the two critical events.

Q.E.D.

Theorem 5.1

$$T \geq \frac{n^2}{S+1} - n + S$$

Proof

Let the first block of outputs pebbled consist of the first $n - \left\lfloor\frac{n}{S+1}\right\rfloor(S+1)$ outputs that are pebbled, if (S+1) does not divide n. The remaining $\left\lfloor\frac{n}{S+1}\right\rfloor$ blocks consist of groups of S + 1 outputs each, that are pebbled in sequence. If (S+1) divides n, then all $\frac{n}{S+1}$ blocks consist of S + 1 outputs each that are pebbled in sequence.

Since every output is a function of all the inputs, all n inputs have to be pebbled during the pebbling of the first block of outputs. This corresponds to n - 1 OR and AND operations.[6] We have seen from Lemma 5.1 that at least n - S inputs are pebbled during the pebbling of every succeeding block. The n - S inputs pebbled together with the S vertices corresponding to the functions $P_1, P_2, \ldots, P_S$ imply that at least n-(S+1)[6,p.24] OR and AND operations are performed during the pebbling of every block of outputs. Thus

$$T \geq (n-1) + (n-(S+1))\cdot\left\lfloor\frac{n}{S+1}\right\rfloor \quad \text{if } S+1 \text{ does not divide } n$$

$$\geq (n-1) + (n-(S+1))\left(\frac{n}{S+1}\right) \quad \text{if } S+1 \text{ divides } n$$

or

$$T \geq \frac{n^2}{S+1} - n + S$$

Q.E.D.

5.3 Concluding Remarks

We have demonstrated above a method of obtaining a Space-Time tradeoff for Sorting which is applicable to any oblivious Sorting algorithm that uses the functions Max(a,b) and Min(a,b). As indicated earlier they apply

when any other monotone mappable set of operations is used. The method we have used is an extension of a method due to Grigoriev [19]. The extension, as seen above, essentially consists of the idea that the set of inputs can be partitioned into two subsets and chains constructed on each subset of inputs such that the chain property is preserved when the two subsets of inputs are coalesced. Using the property that the outputs of monotone functions do not decrease when the sequence of input values forms a chain, we obtain a lower bound on the number of operations between two critical events. Grigoriev [19] has obtained bounds for the Space-Time tradeoff for matrix-multiplication ($O(n^3)$) and convolution ($O(n^2)$); Tompa [15] has obtained a bound of $T \geq \frac{n^2}{16S}$ for oblivious Merging of two sorted lists of n inputs each.
Of course the same bound also holds for Sorting. Our method is significant because oblivious Sorting algorithms possess neither Grate [25] properties as does oblivious Merging, nor ℓ-independence [19] for any value of $\ell \geq O(\log_2 n)$ as does Convolution and Matrix Multiplication (which by Grigoriev's method would yield $ST \geq \Omega$ (nlogn)). As an application of the above result, we can conlude that Batcher's algorithm needs space $\geq \Omega\ (n/(\log n)^2)$ if Time is equal to $O(n(\log n)^2)$.

Chapter 6

Conclusion

This dissertation has attempted to clarify the concept of Space-Time tradeoffs in the context of the class of oblivious algorithms. A pebble game has been used as a model of computation with restricted space, to obtain Space-Time tradeoffs for the FFT, Linear Recursion and Sorting. In doing so, we have used a different method of analysis for each of the above three cases. The upper and lower bounds on the tradeoffs obtained are equal to each other to within a small constant. The analysis of Sorting is significant because it is relevant to a whole class of algorithms. There are several research problems in the same area which are of interest.

1. Obtaining a non-linear Space-Time tradeoff for a specific algorithm that computes a single function.
   This problem is interesting because it would be the first measure of complexity that is provably non-linear for a specific function.
2. Obtaining a non-linear Space-Time tradeoff for a specific single output function using any of a general class of algorithms. This has greater practical significance than the first if the particular function is a popular or commonly used function, because it would give a programmer a choice of algorithms, to be chosen according to available resources.

3. Improving the $O(n^2)$ bound for Sorting by a factor of about $\log_2 n$. This is an interesting problem, because other results (e.g., combinational complexity [24]) indicate that the tradeoff must be about $n^2 \log n$.

4. Obtaining Space-Time tradeoffs when recomputation of intermediate results is not allowed. The tradeoff would probably result by considering different algorithms to compute the same function. This is important because many practical implementations of algorithms do not use the concept of recomputation of intermediate results.

Appendix

To prove that a synchronous graph G of fan-in 2 can be pebbled in Width (a)+2 pebbles we assume that all vertices at some level ℓ have pebbles currently on them and that there are Width (a) vertices at level ℓ. We then show how all vertices at level $\ell+1$ can be simultaneously pebbled using at most 2 more pebbles. This could be continued until all levels of G, and hence G itself, is pebbled.

We use induction on Width (a) to show that we can simultaneously pebble all vertices in level $\ell+1$.

Choosing Width (a) = 2 as a basis for the induction, level ℓ and $\ell+1$ together comprise at most 4 vertices and hence level $\ell+1$ can be pebbled in Width (a) + 2.

When Width (G) = k we have the following cases.

1. **<u>All vertices in level ℓ have fan-out 2.</u>**

The nodes in level ℓ and $\ell+1$ form a graph of the kind shown in Fig. 2.5.2. There can be one or more disjoint pieces of the same type. Referring to Fig. 2.5.2 we pebble vertices α and β in level $\ell+1$ which have a common ancestor γ in level ℓ. We then remove the pebble on γ in level ℓ and pebble vertex δ. This can be continued until all vertices are pebbled. We use at most Width (G)+2 pebbles.

2. **<u>There exists a vertex in level ℓ with fan-out 1, with no vertices of fan-out $\ell>2$.</u>**

Fig. 2.5.1 illustrates the two possible subcases. In Case a, vertex 2 is assumed to have fan-out two. We place the two extra pebbles on nodes α

and β. We remove the pebbles on vertices 1 and 2 in level ℓ. We are then left with two levels of width at most $k-2$. By the hypothesis assumption the two levels of width $k-2$ need at most $k-2+2 = k$ pebbles to simultaneously pebble vertices in level $\ell+1$. Since we have already used 2 pebbles we use a total of at most $k+2$ pebbles as was to be proved. **In Case b, vertex 2 is assumed to have fan-out 1. The same procedure as above can be followed.**

3. <u>There exists a vertex in level ℓ with fan-out m, $m > 2$</u>

Then there exists at least $m-2$ vertices in level ℓ of fan-out 1. Otherwise the total fan-out of vertices at level ℓ is

$$\geq m + (m-3) \times 1 + (k - (m-3+1))2$$

$$\geq m + m - 3 + 2k - 2m + 4$$

$$\geq 2k + 1$$

Since the fan-in of vertices of level $\ell+1$ is at most 2, the total fan-out of vertices of level ℓ cannot exceed the total fan-in at level $\ell+1$ which is at most $2k$. This leads to a contradiction. Therefore for $m \geq 3$ there exists a vertex at level ℓ with fan-out 1. **Using one extra pebble, the output corresponding to a fan-out 1 vertex is pebbled. The pebble on the fan-out 1 vertex is removed. The remaining $k - 1$ vertices can be pebbled using at most $k + 1$ pebbles, by the induction hypothesis.**

Q.E.D.

BIBLIOGRAPHY

1. Hopcroft, J. E., W. J. Paul, and L. G. Valiant, "On Time Versus Space," JACM, Vol. 24, No. 2, pp. 332-337, 1977.

2. Knuth, D. E., The Art of Computer Programming - Sorting and Searching, Vol. 3, Addison-Wesley, Reading, Mass., 1973.

3. Pippenger, N., "Fast Simulation of Combinational Logic Networks by Machines Without Random-Access Storage," IBM Report, June 1977.

4. Fischer, M. and N. Pippenger, unpublished, 1973.

5. Schnorr, C. P., "The Network Complexity and the Turing Machine Complexity of Finite Functions," Acta Informatica, V. 7, pp. 95-107, 1976.

6. Savage, J. E., The Complexity of Computing, Wiley-Interscience, New York, 1976.

7. Shannon, C. E., "A Symbolic Analysis of Relay and Switching Circuits," Trans. AIEE, Vol. 57, pp. 713-723, 1938.

8. Paul, W. J., R. E. Trajan, and J. R. Celoni, "Space Bounds for a Game on Graphs," Eighth Ann. Symp. on Theory of Computing, Hershey, PA, pp. 149-160, May 3-5, 1976.

9. Sethi, R., "Complete Register Allocation Problems," SIAM J. Comput., Vol. 4, No. 3, pp. 226-248, 1975.

10. Karp, R., "Reducibility amogg Combinatorial Problems, Complexity of Computer Computations, R. E. Miller and J. Thatcher eds., Plenum Press, New York, 1972.

11. Sahni, S. and T. Gonzales, "P-Complete Problems and Approximate Solutions," Proceedings of the 15th Annual Symposium on Switching and Automata Theory, pp. 28-32, New Orleans, 1974.

12. Paterson, M. S. and C. E. Hewitt, "Comparative Schematology," Proj. MAC Conf. on Concurrent Systems and Parallel Computation, Woods Hole, MA, pp. 119-127, June 2-5, 1970.

13. Cooley, J. M. and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," Math. Computation, Vol. 19, pp. 297-301, 1965.

14. Savage, J. E. and Swamy, S., Space-Time Tradeoffs on the FFT Algorithm, (to appear), IEEE Trans. on Information Theory, 1978.

15. Tompa, M., "Time-Space Tradeoffs for Computing Functions, Using Connectivity Properties of their Circuits," Proc. 10th Annual ACM Symposium on Theory of Computing, May, 1978.

16. Borodin, A., and Munro, I., The Computational Complexity of Algebraic and Numeric Problems, American Elsevier, New York, 1975.

17. Cook, S. A., "An observation on time-storage tradeoff," Proc. Fifth Annual ACM Symp. on Theory of Computing, (1973), pp. 29-33.

18. Chandra, A. K., "Efficient Compilation of Linear Recursive Programs," IBM Research Rept. RC4517, 10 pp. August 29, 1973.

19. Grigoriev, D. Yu, "An Application of Separability and Independence Notions for Proving Lower Bounds of Circuit Complexity," Notes of Scientific Seminars, Steklov Math. Inst., Leningrad Branch, Vol. 60, p. 38-48, 1976.

20. Manna, Z., Mathematical Theory of Computation, McGraw Hill, New York, 1974.

21. Guttman, A. J., "Programming Recursively Defined Functions in FORTRAN," Intl. Jour. of Computer and Info. Sci., Vol. 5, No. 2, 1976.

22. Gallagher, R. G., Information Theory and Reliable Communications, John Wiley and Sons, New York, 1968.

23. Bird, R. S., "Notes on Recursion Elimination," CACM, Vol. 20, No. 6, 1977, pp. 434-439.

24. Lamagna, E., "The Complexity of Monotone Functions," Brown University Technical Report CS-3, June 1975.

25. Valiant, L. G., "Graph-Theoretic Properties in Computational Complexity," JACM, Vol. 13, 1976, pp. 278-85.

26. Lipton, R. J., and Tarjan, R. E., "Applications of a Planar Separator Theorem," Proc. 18th Annual Symp. on Foundations of Comp. Sci., (1977), pp. 162-170.

27. Pippenger, N., "A Time-Space Tradeoff," Report RC6550, IBM, T. J. Watson Res. Ctr., Yorktown Heights, N.Y., 1977.

28. Cooley, J. M., P. A. Lewis, and P. D. Welch, "History of the Fast Fourier Transform," Proc. IEEE, Vol. 55, pp. 1675-1677, 1967.

29. Comprehensive Operating Supervisor for the GE-600 LINE (GECOS III), Ref. Manual, General Electric, May 1968, revised October 1970.

30. Ginzburg, A., Algebraic Theory of Automata, Academic Press, New York, 1968.

31. Aho, A. V., J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

32. Wegner, P., Programming languages, information structures and machine organization, McGraw-Hill, 1968.

$$\begin{tikzcd}
\text{non-oblivious TM} \arrow[r, shift left, "{T, S\ [3]}"] \arrow[d] \arrow[dr, "{T\log_2 S,\ S\ [4,5,6]}"] & \text{non-oblivious RAM} \arrow[l, shift left, "\alpha\ ?"] \arrow[d, "\beta\ ?"] \\
\text{oblivious TM} \arrow[r, shift right, "{T, S\ [5]}"'] & \text{oblivious RAM} \arrow[l, shift right, "{T(\log_2 T)^3,\ T\log_2 T}"']
\end{tikzcd}$$

FIG. 2.1 TM-RAM TRANSFORMATIONS

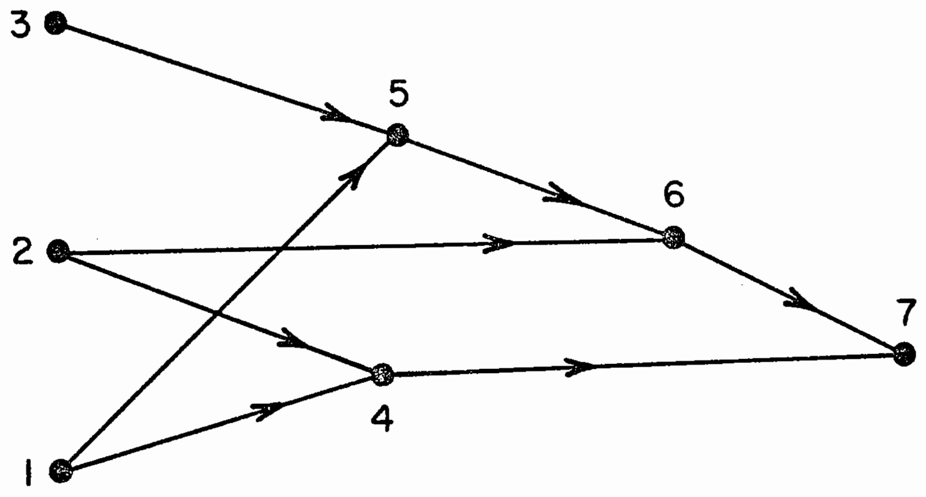


FIG. 2.2 A PEBBLE GAME ON A GRAPH

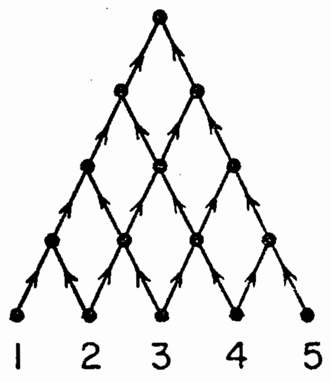


FIG. 2.3 COOK'S GRAPH WITH WIDTH = 5
SPACE REQUIREMENT = WIDTH + 1 =
DEPTH + 2

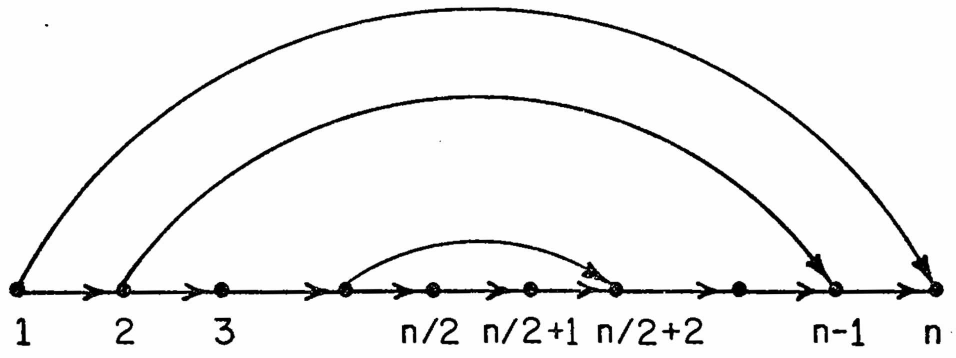
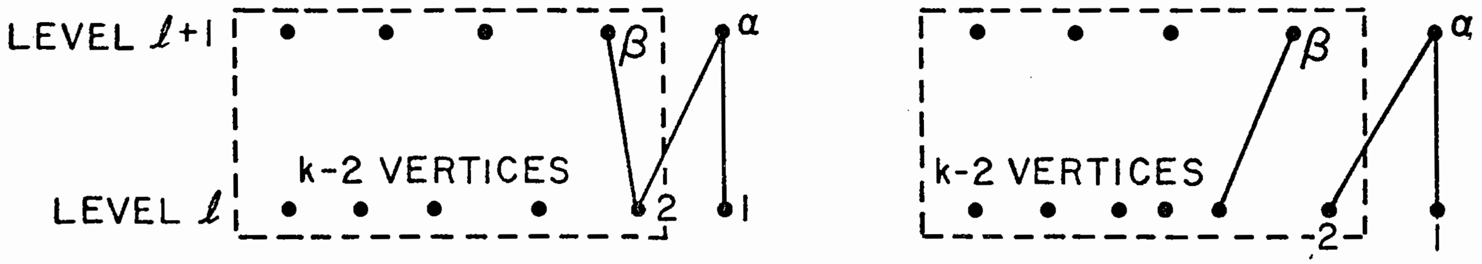


FIG. 2.4. A GRAPH THAT NEEDS n/2 PEBBLES IF EACH VERTEX IS PEBBLED ONCE



CASE a.

CASE b.

FIG. 2.5.1

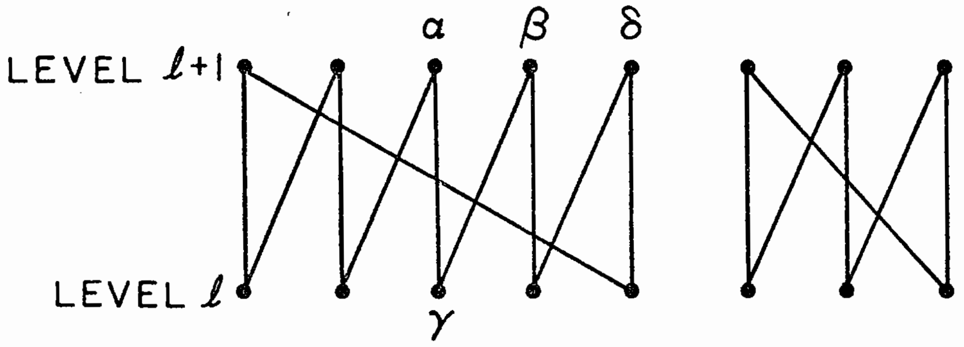


FIG. 2.5.2 PEBBLING A LEVEL CONTAINING WIDTH (G) VERTICES

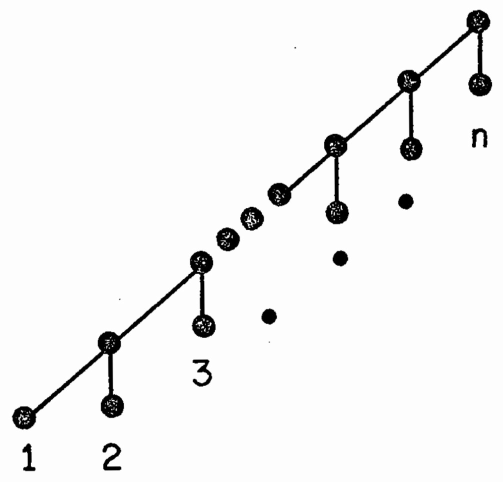


FIG. 2.6 A GRAPH THAT NEEDS ONLY 3 PEBBLES TO PEBBLE EVERY VERTEX.

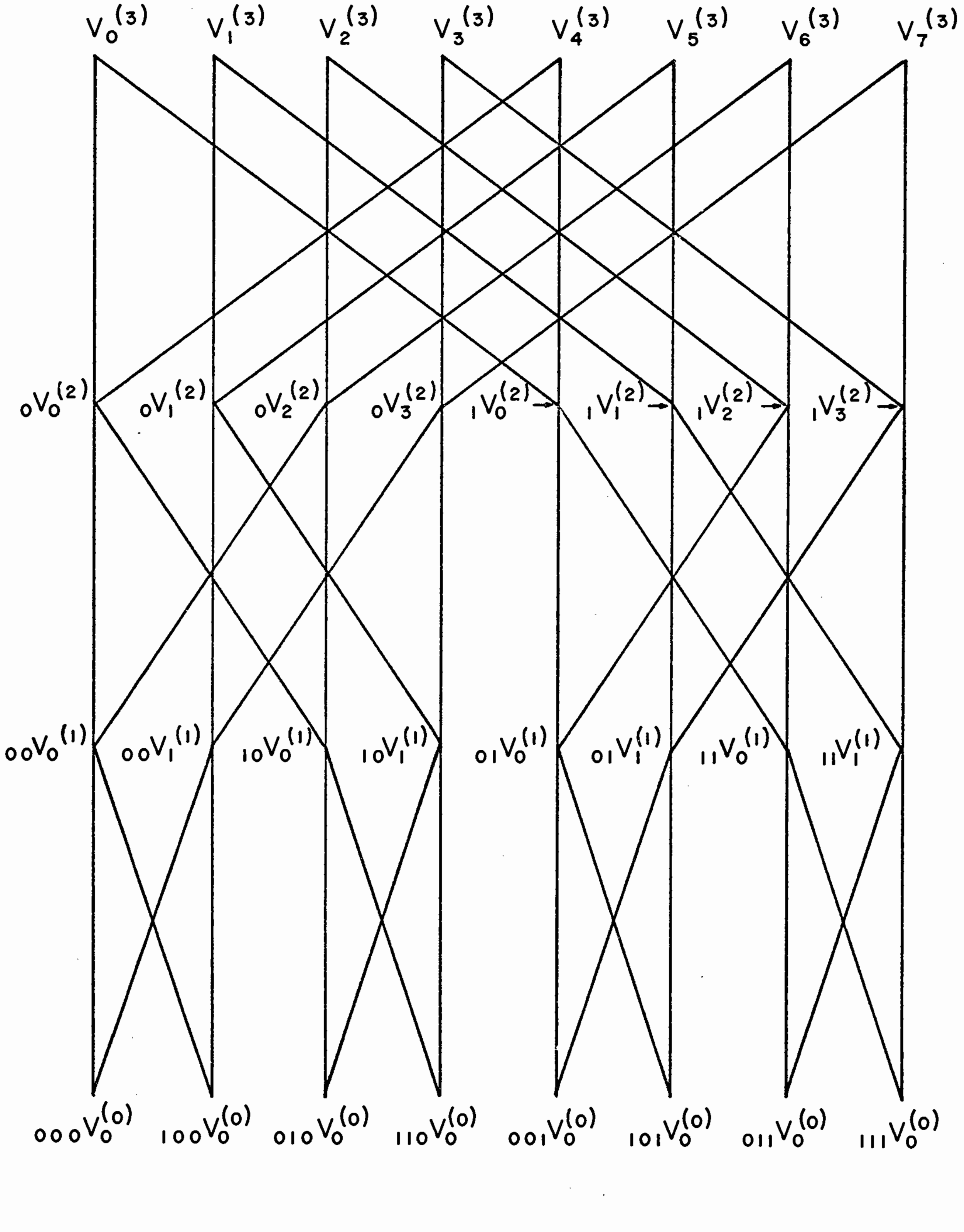


FIG. 3.1a FFT GRAPH WITH EIGHT INPUTS

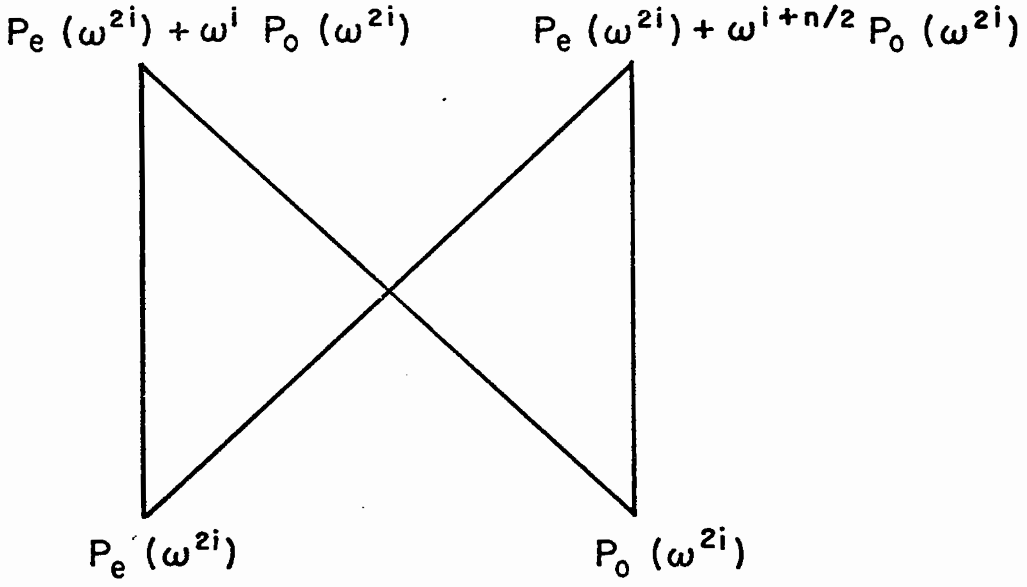

$P_e(\omega^{2i}) + \omega^i P_o(\omega^{2i})$
$P_e(\omega^{2i}) + \omega^{i+n/2} P_o(\omega^{2i})$
$P_e(\omega^{2i})$
$P_o(\omega^{2i})$

FIG. 3.1b A TWO INPUT - TWO OUTPUT FFT USED IN THE CONSTRUCTION OF FFT GRAPHS ON 2^d INPUTS, $d \geq 1$

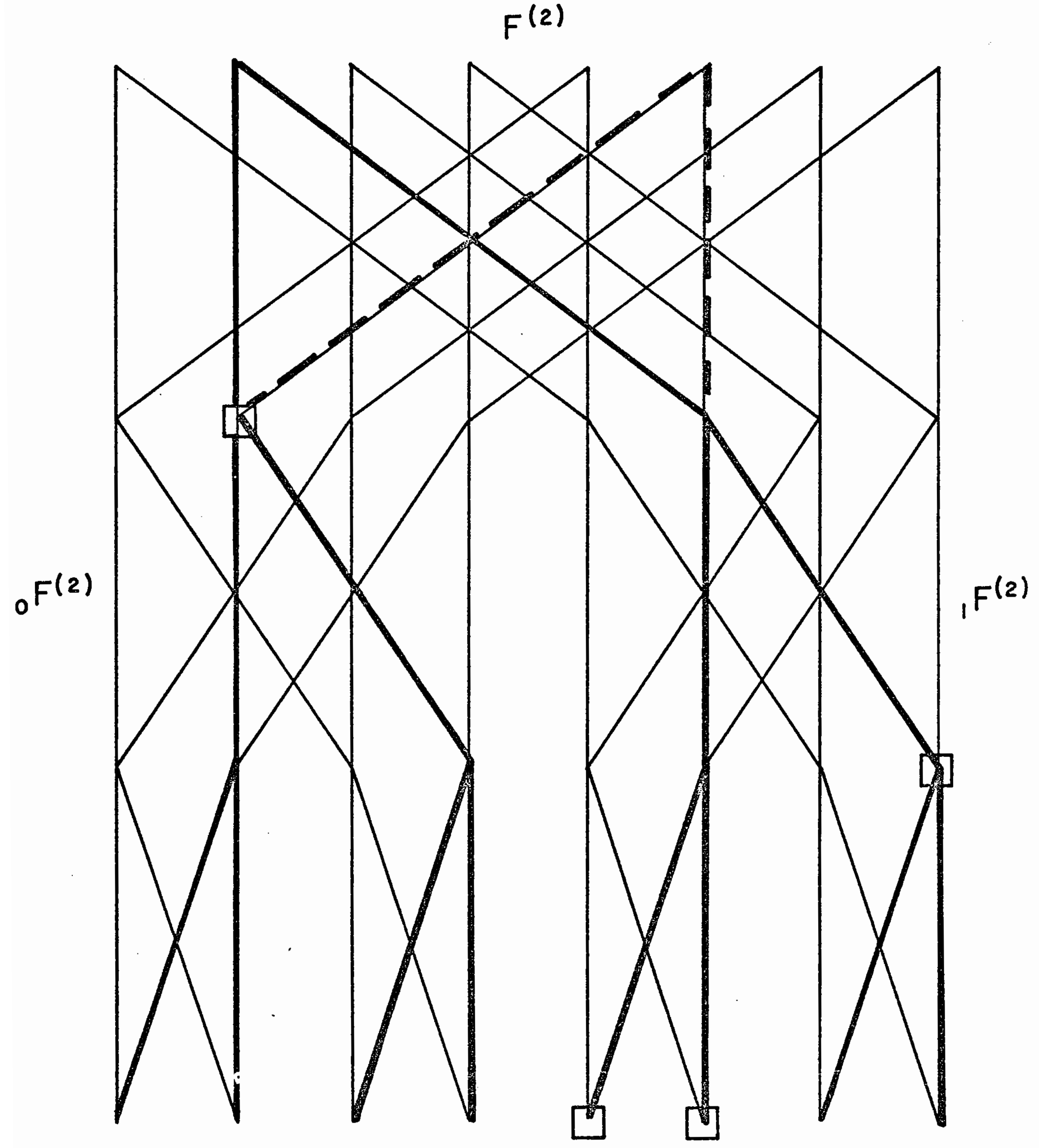


FIG. 3.2 A CRITICAL EVENT ON AN FFT GRAPH USING S_{min} PEBBLES

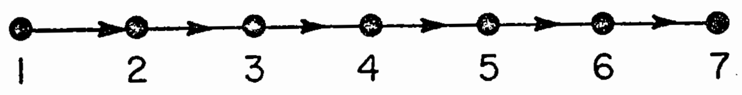


FIG. 4.1 THE CHAIN L_n FOR $n = 7$

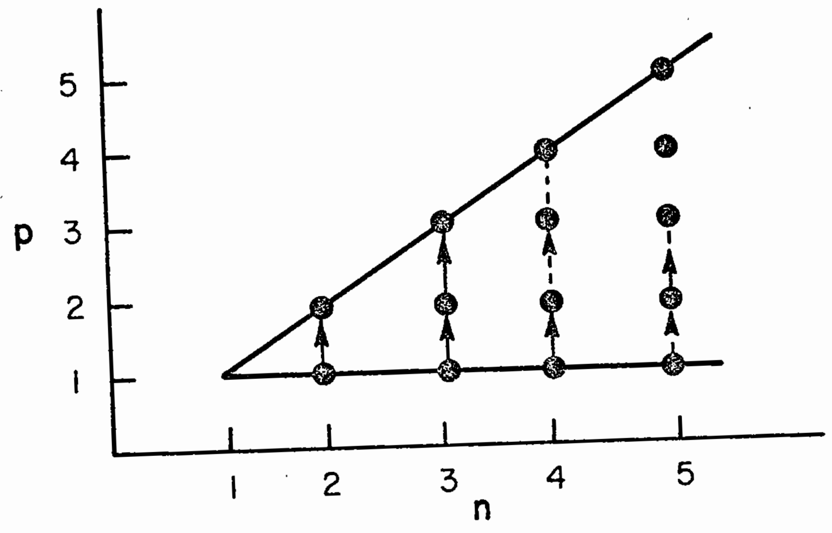


FIG. 4.2 BOUNDARIES AND INDUCTION SEQUENCE FOR THEOREM 1

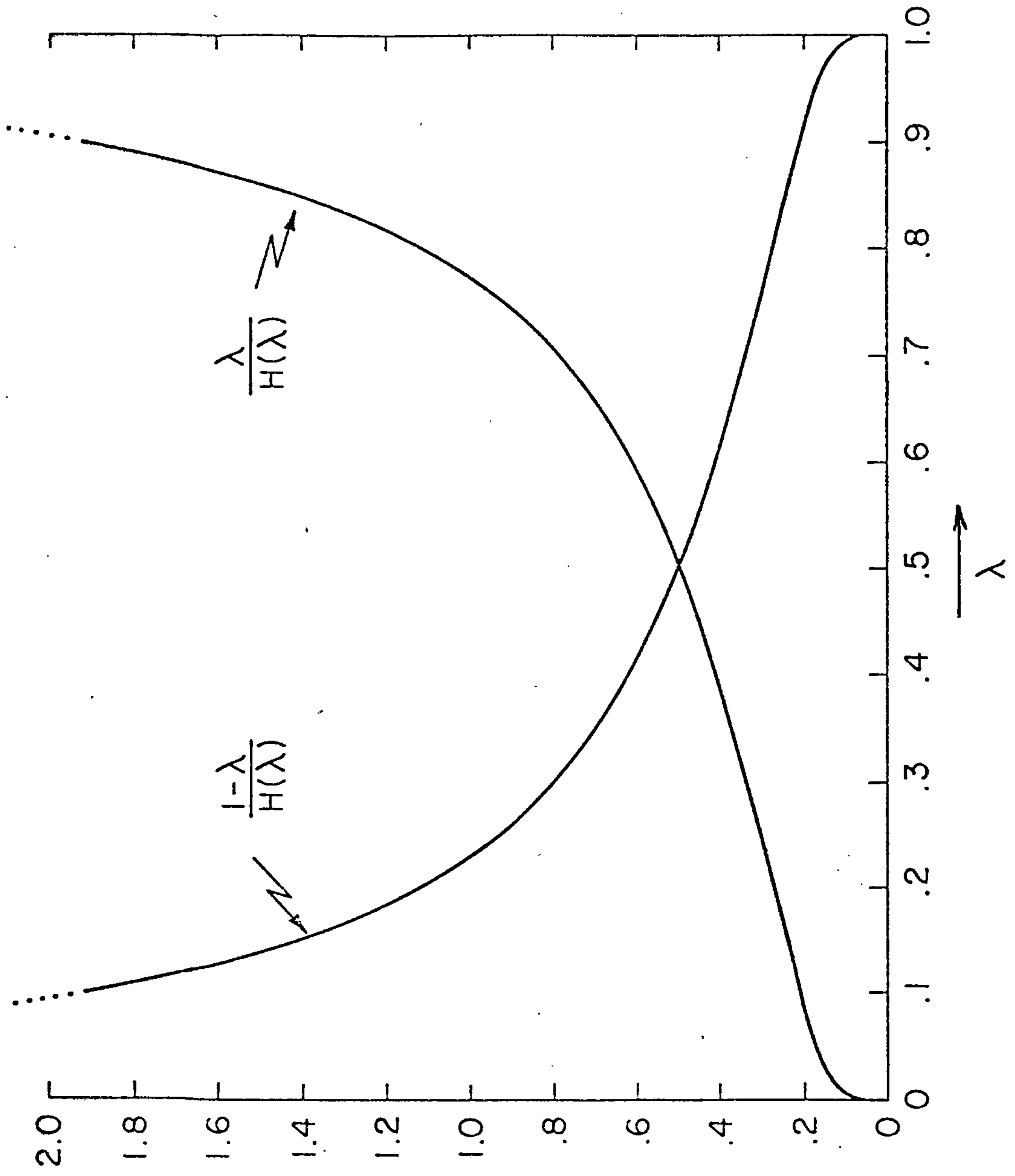


FIG. 4.3 THE FUNCTIONS $\lambda/H(\lambda)$ AND $(1-\lambda)/H(\lambda)$