In his first lecture, Professor Hopcroft discussed some topics relating to general techniques for the design of algorithms. He began by observing that the area of algorithms has attracted considerable attention during the past five years, and as a result, many new algorithms have been discovered and interesting insights obtained. However, the field probably appears to the outsider to be no more than a collection of results with very little coherence. Therefore, the purpose of the talk was an attempt to superimpose some unity on the field and to show that there are some general ideas underlying algorithm design. Several general techniques are discussed including Divide and Conquer, Dynamic Programming, Balance, and the choice of Data Structures; these are illustrated with simple examples of familiar algorithms. (Much of the material for these three talks was taken from Aho, Hopcroft, Ullman [1974] and details on examples can be found there.)

Most of the talk was concerned with asymptotic growth rates—that is, the rate at which computation time increases with the size of input to an algorithm as that size becomes very large. To justify this degree of attention, Professor Hopcroft presented the simple example of multiplication of two integers. Figure 1 illustrates the method of multiplication of two four-digit integers as taught in elementary school, where each dash represents one digit. In this method each digit of one multiplier is multiplied by the other multiplier and the results are added. The amount of work done to multiply this way is approximately \(n^2\) steps, where \(n\) is the number of digits of the multipliers. We express this by saying that the time is \(O(n^2)\).
Figure 1  Elementary school method of integer multiplication

\[
x = \begin{array}{c}
a \\ b
d\end{array} \quad y = \begin{array}{c}
c \\ d\end{array}
\]

\[
x = a^2 + b \
y = c^2 + d
\]

\[
xy = ac^2 + (ad + bc)^2 + bd
\]

\[
= ac^2 + [(a+b)(c+d) - ac - bd]^2 + bd
\]

Figure 2  Karatsuba-Ofman algorithm
meaning that the time is approximately proportional to \( n^2 \) for large \( n \) but that we are ignoring the magnitude of the constant of proportionality. (The precise value of this constant is very dependent upon the model of computation and style of computer being used, and consideration of it is beyond the scope of this discussion.)

Figure 2 shows a second well-known method for multiplying two integers. The \( n \)-digit integers, \( x \) and \( y \), are divided into two parts, each of \( n/2 \) digits, with components \( a, b, c \) and \( d \) as illustrated. (This example is of integers in the base 2, but clearly the method is independent of the base.) The product of \( x \) and \( y \) is as shown. In the rearranged form, there are exactly three multiplications of \( n/2 \)-digit integers, namely \((a+b)(c+d), a \times c, \) and \( b \times d\). The last two products are used twice but clearly need to be formed only once. The time, \( T(n) \), to form this product as a function of the number of digits is thus the time of three \( n/2 \)-digit products plus the time it takes to add all of the pieces together, which is \( O(n) \). This can be expressed by the recurrence relationship.

\[
T(n) = \begin{cases} 
\text{constant}, & n = 1 \\
3T(n/2) + kn, & n > 1 
\end{cases}
\]

Solving this, we find that \( T(n) \) is \( O(n^{1.69}) \).

Observe that this method is asymptotically faster than the elementary school method. So why is it not taught instead? Professor Hopcroft speculated that the real reason is not the difficulty in grasping the second method; it is instead that the constants involved are such that the first method is actually faster for \( n < 40 \) (approximately), and that the second method is not actually faster until \( n \) is greater than this. With paper and pencil computations, no one ever solves such large problems; so the asymptotic performance is not really an issue. But with a computer, the size of the problem increases and the second algorithm may be extremely relevant.

\[\text{† Ignoring details, such as the possibility that } n \text{ is odd, which does not substantially affect the results.}\]
Another justification for the study of asymptotic growth rates of algorithms is illustrated by Figure 3. This shows by way of example how the maximum size of problem which can be solved varies with the time complexity of the algorithm and the time available for a computation (or equivalently, the speed of the computer). For example, if a new generation of computers is 1,000 times faster than the present generation, it does not necessarily follow that we can solve problems which are 1,000 times larger. That is only true if the complexity of the algorithms (their growth rates) are linear. If the complexity is quadratic, the size of the problems which can be handled increases only by a factor of the square root of 1,000, that is, about 31; and for a cubic algorithm, the size increases only by a factor of 10. If the algorithm is exponential there is no multiplicative increase at all but only an additive increase of log 1,000, which is about 10.

<table>
<thead>
<tr>
<th>Time Complexity of algorithm</th>
<th>Maximum problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 sec</td>
</tr>
<tr>
<td>( n )</td>
<td>1000</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>140</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>31</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>10</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 3  Time to execute algorithms of various complexities as a function of problem size.

As computers get faster, we tend to work on harder problems—that is, ones farther down the table of Figure 3. Thus as we move across the table by increasing computing power, there is very little improvement. But if we move up the table by replacing, say, exponential algorithms with cubic or quadratic ones, there is a big improvement in the size of problem as a function of speed.
Professor Hopcroft conjectured that interest in asymptotic complexity, as illustrated by these examples, will increase as computers become faster. He then turned to the main subject of the talk, namely what are some of the techniques which can be used to design efficient algorithms.

Divide and Conquer

One of the first techniques is that of 'Divide and Conquer'. This is the technique of taking a problem of a size, say \( n \), and expressing its solution in terms of a small number of subproblems, each of a size a fraction of \( n \). The integer multiplication algorithm given in Figure 2 is an example of this, where a multiplication problem of size \( n \) was expressed as three subproblems of size \( n/2 \).

The Divide and Conquer technique has been used extensively to improve the asymptotic growth rate of many algorithms - for example, Strassen's algorithm expresses the multiplication of two \( n \) by \( n \) matrices as seven multiplications of \( n/2 \) by \( n/2 \) matrices (Strassen [1969]). Similarly, sorting sets of \( n \) items can be accomplished by sorting two subsets of \( n/2 \) items and merging the sorted subsets together. Other examples include the fast Fourier Transform and the fast Chinese remainder algorithms, binary search algorithms, and even various constructions in automata theory. The constructions in the last area are very similar to algorithms, and the techniques of algorithm design also apply to them. For instance, Savitch's result [1970] that anything which can be done in non-deterministic polynomial space can also be done in deterministic polynomial space is a good example of Divide and Conquer.

Let us consider one algorithm in detail, namely the Linear Median algorithm based on the work of Blum, Floyd, Pratt, Rivest and Tarjan [1973]. The problem is to find the median from a set of \( n \) items; it has the obvious solution of sorting the set and selecting the member in position \( n/2 \). But this method takes time \( O(n \log n) \), whereas it would be nice to have an algorithm which takes time \( O(n) \). The algorithm illustrated in Figure 4 is such an algorithm. In fact, it is slightly more general in that it selects the \( i \)th element from a set \( S \), the median being thus the special case of \( i = n/2 \).

The algorithm SELECT \((i,S)\) works as follows:

The set of \( n \) items is arranged in \( n/5 \) columns of five elements each, as illustrated by the dashes at the top of the figure. Each column
SELECT \( (i,S) \)

1. sort columns
2. find median \( m \) of middle row
3. construct
   \[
   S_1 = \{ x | x \leq m \}
   \]
   \[
   S_2 = \{ x | x > m \}
   \]
4. if \( |S_1| \leq i \) then SELECT \((i,S_1)\) else
   SELECT \((i-|S_1|,S_2)\)

Figure 4 Linear median finding algorithm
is sorted separately; and since each sort takes constant time, the whole set is processed in time $O(n)$. (We could not, of course, do the same thing with the rows, as that would take time $O(n/5 \log n/5)$.) Then the middle row - that is, the set of $n/5$ items consisting of the middle item from each column, circled with a solid line in the figure - is selected and the SELECT algorithm is applied recursively to find its median, $m$. We call $m$ the sample median for the whole set $S$.

The set $S$ is now partitioned into two subsets $S_1$ and $S_2$, where $S_1$ consists of all elements less than the sample median and $S_2$ consists of all elements greater than it. By construction and the transitivity of the 'less than' relationship, $S_1$ and $S_2$ each have at least one-quarter of all elements of $S$. This is illustrated by the subsets enclosed by dotted lines in Figure 4, these representing the 'smallest' possible sizes of $S_1$ and $S_2$. This permits a division of the problem, for by determining the sizes of $S_1$ and $S_2$, it is possible to determine which contains the actual desired element. Then the algorithm is applied to that subset, which can have no more than $3/4 n$ elements in it.

The running time of this algorithm is given by the following recurrence relationship

$$T(n) = \begin{cases} 
c & \text{small } n \\
(c n + T(n/5) + T(3n/4)) & \text{large } n,
\end{cases}$$

where $c$ is a constant.

This has the solution

$$T(n) = 20 c n,$$

which is linear because the sum of the sizes of the two subproblems, $n/5 + 3n/4$, is less than $n$ itself.

The Divide and Conquer technique leads naturally to recursive algorithms, such as the previous example. Another example, shown in Figure 5, is an algorithm for numbering the vertices of a tree in order. It divides the problem into those of numbering the left subtree (if there is one), numbering the root, and numbering the right subtree (if there is one). The first procedure gives a natural recursive representation of this algorithm. However, there is no reason why
(Recursive)

procedure INORDER(VERTEX):
begin
if LEFTSON[VERTEX] ≠ 0 then
   INORDER(LEFTSON[VERTEX]);
NUMBER[VERTEX] = COUNT;
COUNT = COUNT + 1;
if RIGHTSON[VERTEX] ≠ 0 then
   INORDER(RIGHTSON[VERTEX])
end

(Iterative)

procedure INORDER(VERTEX):
begin
   COUNT = 1;
   VERTEX = ROOT;
   STACK = empty;
   left: while LEFTSON[VERTEX] ≠ 0 do
      begin
         push VERTEX onto STACK;
         VERTEX = LEFTSON[VERTEX]
      end;
   center: NUMBER[VERTEX] = COUNT;
      COUNT = COUNT + 1;
   if RIGHTSON[VERTEX] ≠ 0 then
      begin
         VERTEX = RIGHTSON[VERTEX];
         goto left
      end;
   if STACK not empty then
      begin
         VERTEX = top element of STACK;
         pop STACK;
         goto center
      end
end.

Figure 5
Divide and Conquer algorithms have to be written in a recursive fashion. In cases where a non-recursive algorithm is preferred, an equivalent iterative one such as the second procedure can be developed. These two algorithms are essentially identical and have identical running times (assuming a reasonable implementation of recursion). The only difference is that in the iterative version, certain information about the state of the algorithm has to be kept explicitly in the stack, whereas in the other version it is implicitly encoded in the recursion. However, the recursive algorithm is much easier to work with, to design, to analyse, to prove correct, and to understand. The designer can concentrate his attention on the algorithm itself rather than get lost in the bookkeeping details.

One final observation to be made about Divide and Conquer algorithms concerns running times. It is frequently the case that the running time $T(n)$ can be expressed as a recurrence equation of the form

$$T(n) = \begin{cases} \begin{align*} b & \text{ if } n = 1, \\ a \frac{T(n)}{c} + b_n & \text{ if } n > 1, \end{align*} \end{cases}$$

where $a$ is the number of subproblems, $n/c$ is the size of each subproblem, and $b_n$ is the cost of putting the subproblem solutions together to obtain the final result. This recurrence equation has the following solutions

$$T(n) = \begin{cases} \begin{align*} 0(n), & \text{ if } a < c, \\ 0(n \log n), & \text{ if } a = c, \\ 0(n^{\log c a}), & \text{ if } a > c. \end{align*} \end{cases}$$

This makes it convenient to design algorithms quickly because the running time is known as soon as the size of the subproblems is known.

**Dynamic Programming**

Sometimes we encounter problems which have no natural division into subproblems, and thus we cannot use Divide and Conquer techniques. An alternative technique is the use of Dynamic Programming. Some examples of algorithms developed this way include the Younger-Kasami algorithm of 1967 for recognising strings from context-free languages, an algorithm for discovering the order for multiplying matrices.
together to minimise the number of operations, and the optimisation of binary search trees. In this talk, we will consider one of these, namely context free language recognition, in a little more detail in order to illustrate the technique.

Figure 6 depicts, at the top, a context-free grammar in Chomsky Normal Form and, in the middle, an input string. We require an algorithm to answer the question of whether or not the input string can be derived from the given grammar, where S is the start symbol. Our first thought is Divide and Conquer. But we immediately observe that it is not appropriate to divide the grammar — for example, by taking only one half of the productions at one time — because the size of the problem is in terms of the length of the string. (If we had a problem which was a function of the size of the grammar, then it might be appropriate to divide the grammar.) What we want to do is to divide the string; but if we knew where to divide it, our problem would be already solved, that is, we would divide the string at the last place where it came together as a result of applying one of the productions with S as left-hand side. But then we would already have a parse and would not need to do the recognition.

Thus the only obvious way to break the string into substrings is at every possible position. The first people who did this ended up with exponential algorithms. However, that is unreasonable because the total number of subproblems (that is, substrings) cannot exceed \( n^2 \). Thus the only way we use exponential time is to solve some subproblem over and over again many times. This immediately suggests that it would be reasonable to solve every subproblem and store the result in a table. Then whenever the answer to a subproblem is required it can be obtained by table look-up.

This is exactly the algorithm which has been designed for parsing context free languages in time \( n^3 \). A table like that at the bottom of Figure 6 is constructed, in which an entry at coordinate \((x,y)\) represents all variables of the grammar from which we can derive the substring of length \( y \) starting at position \( x \) of the input. For example, entry \((2,3)\) states that \( A \) is the only variable of the grammar from which the substring "aba" of the input can be derived. Finally, we observe that if
input string = a a b a b b

<table>
<thead>
<tr>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

Figure 6  Context-free language recognition.
S is contained in position (1,n), then the input string can be derived from S and thus conclude that it is a sentence of the grammar. The table itself is filled in from top to bottom, each entry taking work of time $O(n)$, given that the results of all earlier entries are already filled in.

This, in general, is the notion of Dynamic Programming, that is, the construction of a table of results of all possible subproblems in order to obtain the result desired. This algorithm occurs many times in many guises. It may appear to be something special in the case of context free languages, but it is not.

![Diagram](image.png)

Figure 7   Effect of order of associating matrices on number of operations.
Another example is illustrated by Figure 7. Given a string of n matrices to be multiplied together, it makes a big difference in which order the product is formed. For example, the four matrices of the figure can be multiplied from right to left in a total of 125,000 operations (assuming an $n^3$ multiplication algorithm). But if we are clever and multiply them in the order shown on the top of the figure, it only requires 2200 operations. Thus, in a package or language which provides matrix multiplication, a little optimisation is appropriate to decide the order of associating the operands.

Once again, the Divide and Conquer technique does not work because there seems to be no obvious way to divide the problem into some small number of subproblems. What has to be done is to break the problem down in every possible location and calculate the cost of computing each half and the cost of multiplying the two results together. This again is a Dynamic Programming problem, where for each i and j ($i<j$) the cost of multiplying the matrices between position i and position j is entered into a table. There are $O(n^2)$ entries and each entry can be filled in, in time n, from those above it. The minimum cost is found in the (i,n) position. The actual order of multiplying the matrices can be obtained by storing with each entry pointers to the two subproblems into which the problem should be divided.

Another example of an application of Dynamic Programming is the problem of constructing optimum binary search trees. Given a number of words which we want to place in a tree, where the probabilities of the words are known, we want to place them so as to minimise the expected path length. Once again, there does not seem to be a simple way to divide the problem. The best known algorithm is the modified Dynamic Programming approach of Knuth [1971].

Balance

As we consider various algorithms and their running times, it quickly becomes apparent that one very important requirement in algorithm design is to keep things balanced. For example, using Divide and Conquer, we try to find subproblems of approximately equal sizes. If we have a data structure based on trees, it is
helpful to keep the size of the subtrees balanced. If the problem
requires, say, two operations to be performed, it often happens that
some possible choices of data structure facilitate one operation at
the expense of making the other very time consuming. This sort of
problem is balanced by choosing a compromise data structure in which
neither operation is very simple nor too complex.

The importance of balance can be illustrated by several examples.
The problem of sorting \( n \) items for instance, is solved by merge-sort
in time \( O(n \log n) \) by dividing it into two subproblems of size \( n/2 \).
Suppose instead that we try to divide slightly differently — say into
one problem of size one and one problem of size \( n-1 \). This can be done
in several ways, one being to take an element out of the set, sort the
remaining \( n-1 \) items, then insert the one item back in the correct
place. This could lead to a fast algorithm if we are very, very
sophisticated at inserting, but it will usually turn out to be an
\( n^2 \) algorithm. Another possible subdivision of the problem is to
select a specific element, say the minimum one. Then sort the
remaining \( n-1 \) elements and add the selected one to the beginning.
This also takes time \( O(n^2) \), that is the solution of the recurrence
relation

\[
T(n) = n + T(n-1).
\]

Another example is that of the dictionary, illustrated by
Figure 8. We represent the dictionary by a binary tree and provide
the operations INSERT, DELETE, and FIND. FINDing a particular item
can be done very quickly by comparing that item to the root, and if
it is smaller looking in the left subtree, otherwise looking in the
right subtree. If the tree is balanced, a FIND takes \( O(\log n) \)
operations. Once we start doing INSERT and DELETE operations,
however, the tree can become unbalanced very quickly and a FIND can
become very expensive. For example, if BAKER, CALDWELL, DOUGLAS and
MARTIN are deleted and Ze, Zee, and Zebra are added, FINDs will take
an average time \( O(n) \).

What is needed is a method for keeping the tree balanced. A
very simple method is based on a structure called a 2-3 tree,
illustrated by Figure 9. This is a tree in which every path from
the root to a leaf is the same length and each vertex is of either
Figure 8 Representations of a Dictionary
Figure 9  Illustration of a 2-3 tree
degree 2 or degree 3. Notice that when performing the operation FIND in this structure, some vertices require two comparisons in order to decide which son to select. However, the tree has the advantage that it can always be rebalanced easily.

Suppose, for example, we wish to remove node B from the top tree of the figure. This can be done by simply deleting the node, and the remaining tree is still a 2-3 tree. But if we subsequently wish to remove node C, we must rebalance because the remaining tree is not a 2-3 tree. In this case, we place node D under its father's brother, then delete its father. The tree at the bottom of Figure 9 is the result, and this again is a 2-3 tree. If it had turned out that the father's brother node already had three sons, then it would be sufficient to move one of those in place of the deleted node. The worst case of deletion is the case in which there is only one brother node left; this must be moved and the father node deleted. This situation is then repeated at each level up the tree, that is, at most log n times. Insertion is simply the reverse process, and it could require that a root with three sons be split into two roots each with two sons. Thus n INSERT, DELETE, and/or FINDs take time $O(n \log n)$ with rebalancing, whereas it could take up to $O(n^2)$ without rebalancing.

**Data Structures**

Another technique in efficient algorithm design concerns the choices of data structures and data representation. With this technique, a problem is considered in terms of a set of mathematical or abstract objects and a set of operations upon those objects. Then it is appropriate to ask what data structure should be used to represent the objects in order to minimise the worst case asymptotic complexity of an arbitrary sequence of those operations. This approach is particularly useful because the same abstractions often form the basis for the solutions of a lot of different problems. Thus if good solutions to the mathematical problem can be found, they can be used to speed up a number of algorithms.
Some problems which have to be formulated in these terms include:

(1) A QUEUE of objects, with operations INSERT and DELETE (first-in, first-out);
(2) A STACK, with the operations INSERT and DELETE (last-in, first-out);
(3) A HEAP, with INSERT, DELETE, and MIN;
(4) A DICTIONARY, with INSERT, DELETE and FIND;
(5) A MERGEABLE HEAP, namely a HEAP with the additional operation of forming the UNION of two HEAPS; and
(6) A PRIORITY QUEUE, namely a QUEUE with the additional operation of combining two QUEUES according the common ordering of their elements.

An example which we can consider in more detail is the disjoint set union problem which has solutions faster than is normally expected. In this problem, we are initially given n singleton sets of integers which we represent by:

\[ \{1\}, \{2\}, \ldots, \{n\}. \]

For simplicity, we name each set by the integer in it. The operations required are UNION, which forms the union of two sets and gives it an arbitrary name, and FIND, which determines the name of the set currently containing a given integer. Figures 10a and 10b shows two obvious solutions to this problem. In Figure 10a, sets are represented as trees. The FIND operation consists of following the pointers from the given integer up to the root of the tree; and UNIONS are formed by attaching the root of one tree as the son of the other. By the principle of balance, we always form a union by attaching the smaller tree to the larger tree rather than vice versa, for this guarantees that no path ever exceeds \( \log n \) for a tree of \( n \) elements. In this representation, UNIONS take a constant time and FINDs take up to \( O(\log n) \) operations. Thus, an arbitrary sequence of \( n \) operations would take time \( O(n \log n) \).

Figure 10b shows an alternate representation. The sets are represented by the entries of an \( n \)-element array. Each integer indexes an entry which gives the name of the set containing it. A FIND operation is very simple - i.e., just index into the array to
A = \{2, 4, 7\} \quad B = \{1, 3, 6, 9\}

Figure 10a. Representation of a set by a tree

Figure 10b. Representation of a set by an array
find the answer - and it has constant cost. A UNION is a bit more complicated because it requires the names of the entries of one set to be changed to those of the other set. This takes time $O(\log n)$. Thus, a sequence of $n$ operations in the worst case still takes $O(n \log n)$.

![Diagram of trees and nodes](image)

Figure 11  Executing a find operation

Figure 11 shows another, faster solution for solving this problem. This algorithm, though known for a long time, has only been successfully analysed recently. In this case, a set is given represented as a tree, and UNIONs are done as in Figure 10a, applying the principle of balance. However, FINDs are implemented slightly differently. Suppose we wish to FIND element E of Figure 11. We follow the path to the root as before; but as we do, we collect each subtree we encounter and move it up to the root. That is, we make nodes E, D, C, and B sons of node A. This will make life simpler for us in the future.

An analysis of this algorithm shows that a sequence of $n$ operations requires only time $O(n A^{-1}(n))$, where $A(n)$ is Ackermann's function in one variable. Ackermann's function grows so rapidly -

$$A(0)=0, A(1)=1, A(2)=2+2, A(3)=3+3, A(4)=4^4, A(5)=5^{5^5}$$

- that its inverse is very nearly constant. For all practical purposes, the algorithm of Figure 11 thus requires time $O(5n)$.
This also suggests another strategy. Given a sequence of these operations, we normally apply them in the order specified. However, once we consider the problem in the general form, we have some options. It may be that in the problem, we know the complete sequence of operations to be applied before any answer is required. In this case, there is no need to execute these operations in the order in which they occur. Instead, they could be sorted or executed in another order, storing the answers in a list. Then the answers could be presented in the order requested. This technique is called the off-line solution to the problem. Similarly, if we have to execute operations in the order specified because we require answers before we know what the future operations are, we call it an on-line algorithm. This distinction is useful because it turns out that the worst case off-line solutions is faster than the best known worst case behaviour of on-line solution.

Professor Hopcroft concluded his first lecture by presenting a solution to the off-line minimum problem. In this problem, we are presented with a sequence of INSERT operations and MIN operations. INSERT merely places a new integer into a set, while MIN selects and deletes the smallest integer seen so far from the set. The top line of Figure 12 illustrates a sequence of these operations – that is, INSERT 2, 3, and 7 into a set; find and delete the MIN, 2; then INSERT 1 and 4; find and delete the MIN, 1; etc. Doing this on-line requires time $O(n \log n)$.

However, the bottom two lines of Figure 12 illustrate an off-line solution which only takes time $O(n A^{-1}(n))$. Each integer and each MIN is given a pointer to the next MIN in the sequence. Then by considering the integers in order, it is possible to compute the MINS not in the order requested but in a much faster way – that is, we first find where 1 is selected by following the pointer to its MIN; this information is stored in a table. Then we merge the set of pointers to its MIN with those to the next MIN, as illustrated by the second line of the Figure. This process is repeated for 2, as illustrated by the third line, etc. Merging the pointers is done by the set union algorithm of the Figure 11 in time $O(n A^{-1}(n))$. Having considered all integers in order, we know which one each MIN selects; then we
Figure 12  Illustration of the off-line min algorithm
can output the required results for the given sequence by looking up the information we stored in the table.

Lecture 2  Unification of Algorithms

Professor Hopcroft began his second lecture with the simple example of computing the reciprocal of a polynomial.

One way to compute the reciprocal of a degree $n$ polynomial is to apply the Divide and Conquer technique, and to break it into two polynomials each of degree about $n/2$:

$$P = P_1 x^{n/2} + P_2$$

Then

$$\frac{1}{P} = \frac{1}{P_1 x^{n/2} + P_2} = \frac{1/P_1 x^{n/2}}{1 + \frac{P_2}{P_1 x^{n/2}}}$$

Performing a Taylor expansion on the denominator produces:

$$\frac{1}{P_1 x^{n/2}} \left[ 1 - \frac{P_2}{P_1 x^{n/2}} + \left( \frac{P_2}{P_1 x^{n/2}} \right)^2 + \ldots \right]$$

The final term shown here has $x^{-n}$ as its highest power, and so if only the terms up to $x^{-n}$ are of interest, it may be ignored together with succeeding terms. Thus:

$$\frac{1}{P} = \frac{1}{P_1 x^{n/2}} \left[ 1 - \frac{P_2}{P_1 x^{n/2}} \right]$$

This requires that we compute the reciprocal of $P_1$, a polynomial of degree about $n/2$, multiply by $P_2$, subtract from unity and multiply by $1/P_1$ again. The time to compute the reciprocal of a degree $n$ polynomial is equal to the time to compute the reciprocal of degree $n/2$, plus that for two multiplications:

$$T(n) = T(n/2) + M(n)$$

where $M(n)$ is the time to compute the product of two $n^{th}$ degree polynomials.
Professor Hopcroft observed that the above reasoning was simplistic but that the formula was nevertheless correct. On solving the recurrence equation the solution is found to be dominated by the last term, and

\[ T(n) = O(M(n)). \]

This shows that the problem of computing the reciprocal of a polynomial can be reduced to that of multiplying two polynomials. The example demonstrates that division is the same process as multiplication in the sense that an algorithm for multiplications can be converted to one for division which works in a linearly related time. In addition the algorithm given will also work for integers, as these may be considered polynomials. Each "divide and conquer" step here breaks the integer into two parts, namely the first \( n/2 \) bits and the last \( n/2 \) bits.

It also happens that multiplication can be reduced to division. First, reduce multiplication to squaring using the identity

\[ ab = \left( (a+b)^2 - (a-b)^2 \right)/4 \]

Squaring can then be reduced to division using the expression:

\[
a^2 = \frac{1}{\frac{1}{a} - \frac{1}{a+1}} - a
\]

In the integer case one has to be careful with round-off, since only the first \( n \) bits are taken in each reciprocal, but this only requires a finite correction at the end. This shows that the two problems of multiplications and division are essentially the same. Professor Hopcroft then gave a number of example to demonstrate that working with integers and working with polynomials were also essentially the same.

1. **Polynomial Evaluation**

Polynomial evaluation corresponds, in the integer case, to the computation of the remainder when one integer is divided by another. Consider evaluating the polynomial:

\[ p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \text{ at point } x_0. \]
If we divide $P$ by $x-x_0$, and write

$$p(x) = (x-x_0) q(x) + r$$

then setting $x=x_0$, so that the first term vanishes we find that

$$p(x) \mid_{x=x_0}$$

is the remainder when $p(x)$ is divided by $x-x_0$.

The form of the remainder is:

$$\ldots \left( (a_n x_0 + a_{n-1} x_0 + a_{n-2}) \ldots \right)$$

as can be verified by synthetic division, and this can be recognised as Horner's Rule for evaluating a polynomial. Thus evaluating a polynomial is essentially the same process as finding the residue of an integer.

Taking the example one step further, we can consider the evaluation of a polynomial at the $n$ points $x_1, x_2, \ldots, x_n$. This is again achieved by Divide and Conquer, starting with the original polynomial to be evaluated at $n$ points, and reducing the problem to that of evaluating two polynomials of degree $n/2$ at $n/2$ points.

Take:

$$p(x) = q_1(x) \prod_{i=1}^{n/2} (x-x_i) + r_1(x)$$

$$p(x) = q_2(x) \prod_{i=n/2+1}^{n} (x-x_i) + r_2(x)$$

We observe that for $1 \leq i \leq n/2$, evaluating $p(x)$ at $x=x_i$ is the same as evaluating the remainder $r_1(x)$ at $x=x_i$, since one of the products in the first term will vanish. Thus to evaluate $p(x)$ at $n$ points, it is necessary to evaluate $r_1$ at the first $n/2$ points, and $r_2$ at the remaining $n/2$ points. These remainders will have degree at most $n/2$, since the divisions are of degree $n/2$. Thus:

$$\text{for } 1 \leq i \leq n/2 \quad p(x) \mid_{x=x_i} = r_1(x) \mid_{x=x_i}$$

$$\text{for } n/2 \leq i \leq n \quad p(x) \mid_{x=x_i} = r_2(x) \mid_{x=x_i}$$

The recurrence equation, giving the time, is

$$T(n) = 2T(n/2) + n \log n$$

where the last term gives the amount of arithmetic for the step.

The reason for choosing this example is that it is related to the fast Fourier Transform. Here we are given a sequence $a_1, a_2, \ldots, a_n$,
and we wish to compute a new sequence whose terms are of the form:

\[ \gamma_{i-1} = \sum_{i=0}^{n-1} a_i \omega^i \]

where \( \omega \) is the \( n^{th} \) root of unity.

Now the terms in this sequence can be produced by evaluating the polynomial.

\[ p(x) = \sum_{i=0}^{n-1} a_i x^i \text{ at the points } x=1, \omega, \omega^2 \ldots \omega^n \]

Having derived a fast algorithm for this problem (polynomial evaluation), we might expect that the Fast Fourier Transform is the same algorithm, and indeed it is, but specialized to the case where the polynomial is evaluated at the roots of unity rather than at arbitrary points. This accounts for the fact that the Fast Fourier Transform runs in time \( n \log n \), while the evaluation algorithm runs in \( n \log^2 n \). The difference is explained by the fact that when computing the products of the \( x=x_i \), we can arrange them in a special manner, illustrated in Figure 13, and no cross-product terms arise. It is therefore a little faster to compute the products in this tree than the products of arbitrary polynomials.

![Figure 13](image)

This tree can be built in time \( n \), rather than \( n \log n \), and the total computation time is \( n \log n \) rather than \( n \log^2 n \).

2. Polynomial Interpolation

This example is that of fitting a polynomial through a set of points, and corresponds to the problem of reconstructing an integer given its residues modulo certain other integers. This problem occurs when converting a number from a variable radix representation
to positional arithmetic or vice versa. Suppose we are given the residues of an unknown number \( x \) modulo \( p_1 \) and modulo \( p_2 \) so that

\[
x = \begin{cases} r_1 \mod p_1 \\ r_2 \mod p_2 \end{cases}
\]

First, we mentally subtract \( r_1 \) from \( x \) so that

\[
x - r_1 = \begin{cases} 0 \mod p_1 \\ r_2 - r_1 \mod p_2 \end{cases}
\]

and then subtract a multiple of \( p_1 \), say \( a p_1 \), so that the second residue is also zero:

\[
x - r_1 - ap_1 = \begin{cases} 0 \mod p_1 \\ r_2 - r_1 - ap_1 \mod p_2 = 0 \end{cases}
\]

then: \( x = r_1 + ap_1 \)

If \( r_2 - r_1 - ap_1 = 0 \) then \( a = \frac{1}{p_1} (r_2 - r_1) \) where \( \frac{1}{p_1} \) is the multiplicative reciprocal mod \( p_2 \) (that is, if \( x = \frac{1}{p_1} \), then \( x \cdot p_1 = 1 \mod p_2 \)). This enables an integer to be reconstructed from two residues, and can be extended for \( n \) residues using "divide and conquer". Thus, to find \( x \) so that:

\[
x = r_i \mod p_i \text{ for } 1 \leq i \leq n,
\]

we find first:

\[
x_1 = r_i \mod p_i \text{ for } 1 \leq i \leq n/2
\]

\[
x_2 = r_i \mod p_i \text{ for } n/2 < i \leq n
\]

by applying the algorithm recursively, and then compute \( x \) using:

\[
x = \begin{cases} x_1 \mod \prod_{i=1}^{n} p_i \\ x_2 \mod \prod_{i=n/2+1}^{n} p_i \end{cases}
\]

Although this algorithm has been explained in terms of integers, it also works for polynomials, replacing \( r \mod p_i \) by the remainder when dividing \( p(x) \) by \( x - x_i \).

This example illustrates that it is not only in the case of division and multiplication that the analogy between polynomials and integers holds. However, there are operations on polynomials which do not seem meaningful on integers. For instance, to determine whether a
polynomial is square-free, one can compute the greatest common divisor of the polynomial and its derivative. The greatest common divisor is unity if and only if the original polynomial is square free. It is difficult to see how to convert this to an integer algorithm without an equivalent notion of an integer derivative. Notions of integer derivatives do exist, but suffer from the problem that there seems to be no fast algorithm for computing them, and the concept cannot be used in this case.

3. **Graph algorithms**

Another area where apparently dissimilar algorithms turn out to be the same concerns the following three problems:

1. Computing a regular expression from a state-graph of a finite-state automaton.
2. Computing the shortest path in a directed graph.
3. Computing the transitive closure of a graph.

Consider the problem of computing a regular expression for an automaton. In Figure 14 the states are numbered with consecutive integers 1, 2... n with n=4.

![Figure 14](image)

We denote:

\[
R_{i,j}^k = \text{the set of paths from node } i \text{ to node } j, \text{ using only vertices from } \{1, 2, \ldots, k\} \text{ as intermediates.}
\]

Then \( R_{i,j}^0 \) are the direct transitions, and \( R_{i,j}^k \) is needed as a result. The following relation enables \( R_{i,j}^k \) to be derived from \( R \) for lower values of \( k \):

\[
R_{i,j}^k = R_{i,j}^{k-1} + R_{i,k}^{k-1} (R_{k,k}^{k-1}) R_{k,j}^{k-1}
\]
The last term in the expression represents a path from node i to node k (not using node k), followed by a path from node k to itself, followed by a path from node k back to j, again without using node k.

This algorithm can be used for several problems. In the first case, that of finding all paths, the $R_{ij}$ represent sets of strings, the + sign representing set union, the · representing set product, and * representing set closure. To compute shortest paths, each edge is represented by its cost, and $R_{ij}$ is a minimum cost path not using a node greater than k. Here union corresponds to selecting the minimum cost, and concatenation to the addition of costs. Assuming all costs are positive, the shortest path will be loop-free, and closures may be set to zero. For the transitive closure, the $R_{ij}$ represent Boolean values indicating whether a path exists between i and j not using a node greater than k. Here union is replaced by Boolean 'or', concatenation by 'and', and closure by 'true'. The relationship between these algorithms is tabulated in Figure 15.

<table>
<thead>
<tr>
<th>all paths</th>
<th>shortest path</th>
<th>transitive closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic symbols</td>
<td>cost</td>
<td>Boolean variables</td>
</tr>
<tr>
<td>+</td>
<td>min</td>
<td>or</td>
</tr>
<tr>
<td>·</td>
<td>+</td>
<td>and</td>
</tr>
<tr>
<td>*</td>
<td>( ) = 0</td>
<td>( ) = true</td>
</tr>
</tbody>
</table>

Figure 15

4. **Boolean Matrix Multiplication**

Figure 16 illustrates how the multiplication of Boolean matrices may be reduced to transitive closure. If a larger matrix, say $X$, is formed from given matrices A and B, then the transitive closure of $X$ contains the product $AB$ in the upper right hand corner as shown.
\[ X = \begin{bmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{bmatrix} \quad X^* = \begin{bmatrix} 1 & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix} \]

where \( X^* = I + X + X^2 + \ldots + X^n \)

**Figure 16**

If the closure can be computed in time \( T(N) \), and provided \( T(N) \) is well-behaved, \( T(3N) \) will be some constant multiplied by \( T(N) \). Consequently the time for computing a product is linearly related to that for computing a transitive closure. This is hardly surprising since we think of a closure as being more complex than a product, but this can be demonstrated not to be the case since a transitive closure can be reduced to an algorithm for a product.

Figure 17 shows a matrix whose transitive closure is required. It is partitioned into four sub-matrices \( A, B, C \) and \( D \). Matrix \( A \) gives edges which connect vertices between 1 and \( n/2 \); matrix \( B \) gives edges which connect vertices between \( n/2 + 1 \) and \( n \), and \( C \) and \( D \) have similar roles. This is illustrated in Figure 18.

**Figure 17**
The closure is then given by the following matrix:

\[
E \equiv (A + BD^*C)^*
\]

To justify this, consider paths from vertices in 1 to \( n/2 \), ending in the same region, namely the upper left-hand corner of the transitive closure. These paths may consist of a path solely in this region, that is some sequence of edges in \( A \), or a path to the other region (via \( B \)), followed by a sequence of paths in the higher region, followed by a path back (via \( C \)). This gives rise to the expression denoted by \( E \). The expression for the other sub-matrices can be justified in a similar way. The closure matrix is written in a non-symmetric manner in order to minimise the number of "starred" expressions, which have been reduced to two, namely \( D^* \) and \( (A+BD^*C)^* \).

Thus if \( T(N) \) is the time to compute the closure of an \( N \times N \) matrix, then

\[
T(N) = 2T(N/2) + 6M(N/2).
\]

The last term reflects the fact that six multiplications are also needed, but this factor has no influence on the result. Also some additions are required, but since these are no more difficult than multiplications, the coefficient 6 can be increased to account for them. Solving the recurrence equation, assuming \( M \) behaves properly,
leads to the result: $T(N)$ is $O(M(N))$. Thus an algorithm for multiplying can be converted to one for transitive closure, running in a linearly related time.

5. Matrix multiplications and Inversion

By analogy with the case of Boolean matrices, it is possible to convert an inversion algorithm to one for matrix multiplication. It is also possible to use a multiplication algorithm for inversion, as can be demonstrated using a slightly modified form of Strassen's algorithm.

Strassen expressed a matrix, decomposed as shown as the product of three matrices:

$$
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix} = \begin{pmatrix}
I & 0 \\
A_{21} & I
\end{pmatrix}^{-1} \begin{pmatrix}
A_{11} & 0 \\
0 & \Delta
\end{pmatrix} \begin{pmatrix}
I & A_{11}^{-1} A_{12} \\
0 & I
\end{pmatrix}
$$

where $\Delta = A_{22} - A_{21} A_{11}^{-1} A_{12}$.

To compute the inverse, the following identity can be derived:

$$
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}^{-1} = \begin{pmatrix}
I & -A_{11}^{-1} A_{12} \\
0 & I
\end{pmatrix} \begin{pmatrix}
A_{11}^{-1} & 0 \\
0 & \Delta^{-1}
\end{pmatrix} \begin{pmatrix}
I & -A_{11}^{-1} I \\
0 & I
\end{pmatrix}
$$

Thus to invert a matrix of size $n$, we must invert two matrices of size $n/2$, namely $A_{11}$ and $\Delta$, and perform a large number (say $k$) of multiplications. Thus

$$I(n) = 2I(n/2) + kM(n).$$

Solving the recurrence equation leads to:

$$I(n) = \Theta(M(n)).$$

This algorithm does not always work however, since although the original matrix might be invertible, it may happen that one of the partitions ($A_{11}$ or $\Delta$) is not invertible. This occurs for example with a matrix whose only non-zero elements are on the cross-diagonal. However if the original matrix is upper or lower diagonal, then no singular matrices will be encountered.
There are two known solutions to this problem. The first is an unpublished result due to Schonhage, which applies if the elements of the matrix are drawn from an ordered field. If the matrix is multiplied by its transpose, the result is positive-definite-symmetric. During the inversion, each partition also creates positive-definite-symmetric matrices to be inverted, and no unexpected singular matrices arise. The identity: $A^T (AA^T)^{-3} = A^{-3}$ may then be used to produce the required inversion.

This process fails in a binary field with arithmetic modulo two; for example

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Although the original matrix is non-singular, $AA$ produces a singular matrix on its first decomposition.

The second method is called fast LUP decomposition. Since the aim of matrix inversion is frequently to solve a set of simultaneous equations, an LUP decomposition can be produced, and the equations then solved by back-substitution.

Professor Hopcroft then quickly sketched how normal Gaussian elimination could be modified to use matrix multiplication to zero blocks of the matrix at a time, with the effect that the time taken is reduced from $n^3$ to that for multiplication, say $n^{2.81}$ using Strassen's algorithm (Bunch and Hopcroft [1974]).

6. Context-Free Language recognition

Referring to the first lecture, Figure 5 illustrates a method for recognising a context-free language. Valient [1974] expressed the table slightly differently, using the vertical axis to represent the starting position and the horizontal axis for the ending position. This alters the order in which the elements are entered. The initial entries, strings of length one, rather than occupying the top row, occur down the diagonal. This is shown in Figure 19. Performing the
first step in the process then consists of multiplying the matrix with itself, where the element multiplication is somewhat unusual. Given two sets, say \([A,C]\) and \([A,C]\), the cartesian product is formed, namely \([AA, AC, CA, CC]\), and then each of these is matched with the right-hand-side of the productions, whose left-hand-sides form the result set, in this case \([S]\). The recognition process then reduces to that of computing the closure \(M^*\) of the original matrix \(M\), where \(M^*\) is defined as:

\[
\]

Here terms like \(M.M^2 + M^2.M\) correspond to the concatenation of a string of length one and a string of length two or a string length two and a string of length one forming one of length three.

Valient reduced this closure to multiplication, and was able to make use of Strassen's algorithm, thereby reducing the time for context-free recognition below \(n^3\). It should be evident that the same process works for many "dynamic programming" algorithms, that is they can be reduced to computing a closure. However, the underlying structure (usually a ring) will be different for each problem, and a quick way of performing the multiplication is needed. For example, with the shortest path problem, where the operations are "min" and "plus", there seems no way to embed these in Strassen's algorithm.
Professor Hopcroft summed up by suggesting that there were in fact only a few basic algorithms, for instance sort, binary search, set union, partitioning, and matrix multiplication (or transitive closure). He said that given a new problem one can often get considerable insight by asking into which of these types it can be mapped.
Lecture 3  The Role of Theory in Computer Science

One of the concerns of the seminar is the problems involved in Computer Science education. The following examples suggest that any Computer Science course should have a large theoretical content. Although nobody denies that theory is important, there is a great reluctance to believe in it.

Most of the algorithms used today are very simple; there are only a few complex algorithms. Algorithms, produced by a trained and a relatively untrained person, tend to be remarkably similar. In many fields of computing there is an absolute absence of deep results. Although the topics covered by Dr. Rabin [this proceedings] appear to be rather deep, they are more of a branch of mathematics or logic than computing.

At the moment, we may be on the threshold where Computer Science becomes a "deep" subject. In the future, computer scientists are going to have to be well-trained, in order that they may be able to use the available tools to solve a particular problem. The following algorithms have the property that various results have been combined to improve the efficiency of the algorithm and probably would not be obtained by anyone not trained in the theoretical aspects of computer science.

Paths in a graph

The problem is to find all of the paths between two sets of vertices in a graph. For example to find all of the paths between \{L1,L2\} and \{R1,R2\} in Figure 20.

Figure 20
Nearly all of the previous algorithms have attempted to trace out physical paths. An algorithm based upon Strassen's matrix multiplication algorithm however does something far more interesting.

The edges (or lines) \( a_{11}, a_{12}, a_{21}, a_{22} \) are thought of as elements of a matrix, similarly for \( b_{11}, b_{12}, b_{21}, b_{22} \). For this particular graph the problem is equivalent to matrix multiplication. Strassen's algorithm evaluates expression of the form

\[
(a_{11} + a_{12}) b_{11} + a_{12} (-b_{11} + b_{21})
\]

The first term represents two paths. The first path is comprised of the edge \( a_{11} \) followed by the edge \( b_{11} \). The second path (which is shown by a dotted line in Figure 21) is rather interesting being composed of the edges \( a_{12} \) and \( b_{11} \).

![Figure 21](image)

This path is not a continuous path, but is instead a "fictitious" path. Strassen's algorithm makes the manipulation of the graph easier by adding in these "fictitious" paths. At the end of the manipulation, these "fictitious" paths are removed in order to end up with real paths only. In this example, the term \( a_{12} b_{11} \) will remove the path \( a_{12} b_{11} \).

This example suggests that future algorithms in graph theory will no longer work on the physical paths. Instead the paths are going to become symbols in an algebra, with algebraic manipulations being used to obtain the required results.
String Recognition

The next example also suggests that algorithms are going to become more sophisticated. The algorithm will use a 2-way deterministic push down automaton as shown in Figure 22.

![Diagram of a two-way pushdown automaton]

This model has the ability to move two ways along the input tape. Because of this, the model can recognise languages of the form 'ww' which are not context-free. To more fully understand the working of a 2-way deterministic pushdown automaton, consider the problem of the recognition of strings in the following language:

\[ \{ w \# x \mid x \text{ is a substring of } w, \# \text{ is a marker separating the two strings } w \text{ and } x \} \]

Such a pattern recognition occurs in languages like SNOBOL, where there is an operation to determine if 'x' is a substring of 'w'. The automaton starts with the input head to the left of the first symbol in the string 'w\#x'.
The automaton will proceed to move past each symbol in the string 'w' until the input head is reading the 'Ą' symbol. Next, the automaton reverses direction and moves left, placing the symbols of w onto the pushdown store, until the input head is to the left of the first symbol in w. The input head then moves past each symbol in w and past the 'Ą' marker. In this position the machine is in the state shown in Figure 24, with the input head reading the first symbol in the string x.

The automaton then checks to see if the string x is a substring of the string w, starting at the first symbol of the string w. This is done by reading each symbol of the string x. At the same time as a symbol of the string x is read, the topmost symbol is 'popped' off the store and the two symbols are checked to see if they are the same. Should the string x match the first m symbols on the top of the store, then x is a substring of w.
However should the current symbol of $x$ not match the symbol at the top of the pushdown store, then $x$ is not a substring of $w$ starting at the first symbol of $w$. The automaton then backs past each symbol in $x$ and $w$, rewriting those symbols on the pushdown store that were erased. The symbol $w_i$ at the top of the store is then erased, putting the automaton in the state shown in Figure 25.

\[ \emptyset \ w_1 \ w_2 \ \ldots \ \ w_n \neq \ x_1 \ x_2 \ \ldots \ x_n \ \$ \]

![Diagram](image)

**Figure 25**

The matching process is then repeated to see if the string $x$ matches the symbols $w_2, w_3, \ldots, w_n$, on the top of the pushdown store. If the two strings match then $x$ is a substring of $w$, starting at the second symbol of $w$. Should the process fail, the pushdown store is restored, $w_d$ is erased and the matching process is repeated.

Although this method works, in general it will take the same time as the standard algorithm which checks if $x$ is a substring of $w$. However S. Cook [1971] has proved the following theorem:

**Theorem** If set $S$ can be recognised by a 2-way input head, deterministic pushdown automata, (PDA) then $L$ can be recognised in linear time by a random access register machine.

Linear time means that even though the PDA in the original example made $n$ moves, the random access machine requires only $kn$ moves. This theorem is even more powerful, for if the PDA requires
time exponential in \( n \), the simulation of the PDA by the random access machine will still only require linear time. The algorithm for the PDA can be inefficient, but because of Cook's Theorem the equivalent algorithm on the random access machine is efficient.

Indeed the problem of 'is \( x \) a substring of \( w \)' can be solved in linear time on the random access machine. Although it is not easy to find the 'linear algorithm', Pratt and Knuth used Cook's Theorem to construct the linear pattern matching algorithm.

The algorithm constructs a skeletal machine for the pattern; here the states of the machine are equivalent to positions in the pattern. For the pattern 'aabbaab' the skeletal machine is shown in Figure 26.

![Figure 26](image)

When the machine is in a particular state and the next input symbol matches the symbol on the outgoing arrow, it goes into the next state down the chain. Should the symbols not match, the machine is brought back to the last state where the initial portion of the pattern matches a portion of the string up to the last symbol of the string which has just been matched and continues the search from there. These backward moves are shown as dotted lines in Figure 27.

![Figure 27](image)

This algorithm is linear in time bounded by \( 2n \). When the machine moves forward it uses up a symbol from the input string, the machine can only go forward \( n \) times. Although when the machine goes back it does not use up an input symbol, it can only go backwards
as many times as it had gone forward.

**Pattern matching with 'don't care' symbols**

Consider the pattern matching problem, where several positions of the pattern are marked by "don't care" entries. For example, matching brackets, where we are not concerned with what is between the brackets. In this example the strings are

\[
\text{string } w = w_1 \ w_2 \ldots \ w_n \\
x = x_1 \ x_2 \ldots \ x_z = x_1 \ \emptyset x_2 \ \emptyset x_3 \ldots
\]

where any symbol in \( w \) will match a \( \emptyset \) in the pattern \( x \). The previous algorithms will not work, because whenever a don't care entry is met, information is thrown away. The problem is finding all \( i \) such that the pattern \( x \) is a substring of the string \( w \) starting at the \( i \)th position of \( w \):

\[
x_1 x_2 \ldots x_z = w_1 w_1 + 1 \ldots w_1 + z - 1
\]

This problem is equivalent to the AND-OR multiplication problem, in the alphabet \{0,1\}. The problem is to find

\[
z_i = \bigvee_{i=1}^{n} w_i \land x_{i-1}
\]

Construction of the string \( Z \) is equivalent to finding all \( i \) where \( x \) is a substring of \( w \) starting at the \( i \)th position of \( w \). Although multiplication normally takes time of \( O(n^2) \), there is a result due to Schonhage and Strassen [1971] which does integer multiplication in time \( O(n \log n \log \log n) \). In the integer multiplication problem there are 'carries'. Because there are at most \( n \) additions of one bit the carry is at most \( \log n \) bits. To make the Boolean multiplication problem a integer multiplication problem, \( \log n \) zeros are inserted between every bit. Therefore the pattern matching problem can be solved in time \( O(n \log n \log \log n) \).

The multiplication algorithm is of interest in its own right apart from the pattern matching application. To multiply together integers \( x \) and \( y \), the algorithm views \( x \) and \( y \) as strings.
These strings are converted to $\hat{x}$ and $\hat{y}$ respectively, using a fast Fourier transform. Having used a pairwise multiplication on $\hat{x}$ and $\hat{y}$, to get $\hat{x} \times \hat{y}$, an inverse Fourier transform is used to find the value of $x \times y$.

This method can be used to multiply together two polynomials. By evaluating the polynomials at $n$ points to get sample values of the product which are passed to interpolation routines. The time taken has been reduced from $n^2$ to $n \log n$, although there are slight technical problems as Fourier transforms use various roots of unity which are not rational numbers.

**Disjoint set union:**

Initially there are $n$ singleton sets, with the $i$th set containing the integer $i$, and two operations (1) form the union of two sets and (2) find the name of the set which currently contains the value of $i$. This problem arose out of Fortran programming, in finding out which identifier pairs are equivalent at any point in time. An algorithm is required which will compute the transitive closure of these sets of pairs. There are various algorithms which solve this problem. The first algorithm was due to Galler and Fischer (1964) and had an $O(n^2)$ running time. The obvious algorithm using trees or lists and merging the smaller set into the larger requires $O(n \log n)$ time.

This last algorithm was thought to be the best possible solution to the problem. However Stearns and Rosenkrantz gave a solution which ran in time $O(n \log \log n)$. What they did was to represent a set by header, each header had a number of subheaders to which the elements of the set were attached to. To find which set an element belonged to, the algorithm indexes into the element and follows the path via the subheader to the header. The structure of two typical sets is shown in Figure 28.
These trees were kept so that no new subheader was added, until all of the subheaders of the tree had \( \log n \) elements. To merge two sets the header element with the smaller number of subheaders if selected, its subheaders are moved across to the other header, thereby at most doubling its number of subheaders. Should there be two incomplete subheaders; these are rebuilt to form only one incomplete subheader. An analysis of the algorithm shows that the running time is \( O(n \log \log n) \), which is obtained as follows:

- Maximum number of times that a complete subheader is moved: \( \log n \)
- Maximum number of complete subheaders: \( n/\log n \)
- Time spent in moving complete subheaders: \( n \)
- Number of times an element is moved before ending up under complete subheader: \( \log \log n \)
- Number of elements: \( n \)
- Time spent in moving elements: \( n \log \log n \)
- Total time \( n + n \log \log n \), that is, \( O(n \log \log n) \)

If we add a third level with \( \log \log n \) elements per 'sub subheader' we can reduce the time spent in moving subheaders down to linear time as the number of subheaders would be \( n/\log \log n \). The time in
merging the sets would be largely spent down at the bottom level and would be \( n \log \log \log n \). It is possible to add on further levels, provided the number of levels is kept independent of \( n \). At this point, it is worth noting the following:

**Theorem (Blum):** There exist functions for which there is no best program.

It would be interesting to find a natural problem which has no best program.
Perhaps the disjoint set union is such a problem, there being a sequence of algorithms with increasingly better running times.

\[
\begin{align*}
\text{n log n} \\
\text{n log log n} \\
\text{n log log log n} \\
\vdots \\
\vdots
\end{align*}
\]

However Blum has also shown that given a recursively enumerable sequence of faster and faster programs, one can construct a program which is faster than any program in the enumeration. Since the above construction to solve the disjoint set union problem yields better and better solutions, there must be a construction which is even better. Armed with this knowledge we are leading to considering trees with fan out dependent on the distance of the vertex from the root as shown in Figure 29. Vertices of height one have two sons, vertices of height 2 have \( 2^2 \) or 4 sons, vertices of height 3 have \( 2^4 \) or 16 sons and so on.

An algorithm using this data structure has an asymptotic running time of \( nG(n) \) where \( G(n) \) grows more slowly than any finite number of logarithms of \( n \).
This example illustrates the way in which apparently worthless theorems can suggest really interesting results.

**Tape Sort**

If there are three tapes on which to sort $n$ items, it is possible by using divide and conquer techniques to construct an algorithm running in time $O(n \log n)$. Initially the $n$ items to be sorted are put on tape one, with the other two tapes blank. Half of the items on tape one are copied onto tape two, the rest onto tape three. After the last item on tape three a marker is placed which indicates that tape three after that marker is blank. Having also set tape one blank, tape two is split onto tape one and the blank portion of tape three. By a recursive application of this division, it is possible to get down to a few items which are easily sorted. When the items are sorted into small sets, the sets are recursively merged to end up with the sorted set of all of the items on tape one. If $T(n)$ is the running time to sort $n$ items, then the following relation holds.
$T(n) = 2T(n/2) + n.$

Thus $T(n) = O(n \log n).$

Since a one-tape sort can only be done in time $O(n^2)$ the question to be asked is, what is the running time for the two-tape sort problems? The early algorithms to solve the two-tape sorting problem had a running time $O(n^3)$. However Hennie and Stearns [1966] discovered the following theorem.

**Theorem** (Hennie-Stearns): If set $S$ can be recognised in time $T(n)$ on a $k$-tape Turing machine, then $S$ can be recognised in time $T(n) \log (T(n))$ on $2$-tape Turing machine.

Therefore the three-tape sorting algorithm converted by the Hennie-Stearns construction into a two-tape sorting algorithm has a running time of $O(n \log^2 n)$.

**Planar Graph Isomorphism**

A planar graph can be broken down into connected, bi-connected and tri-connected components. If the isomorphism problem can be solved for the tri-connected components, it is possible to combine these parts together to solve the problem for the whole graph. A triply connected graph is a graph, where if any vertex is removed the graph is still connected. Triply connected planar graphs have a unique representation on a sphere.

One of the hardest parts of the isomorphism problem is to find the mapping between the graphs. If the fact that triply connected planar graphs have unique representation on a plane is used it is easier to find the mapping. This is done by fixing one line in each graph and reducing the problem to verifying that the implied correspondence is an isomorphism.

However the triply connected parts are similar to state graphs for finite automata. It is possible to construct finite automata from these triply connected graphs in the following way:

1) each edge in a given direction is made into a state of a finite automaton,

2) two symbols $L$ and $R$ (with the meaning of Left and Right) are
associated with each vertex. Then each vertex has the form of Figure 30:

![Figure 30](image)

A move of the automata is then of the form

\[
\delta (S, R) = t \\
\delta (S, L) = u,
\]

where if the automaton is in state \( s \) and scans an \( R \) on the input tape, it goes into state \( t \). This means that a graph \( G \) has been converted into an automaton \( M_1 \), similarly a graph \( G_2 \) is converted to an automaton \( M_2 \). It has been proved that the two graphs \( G_1 \) and \( G_2 \) are isomorphic, if and only if the two automata \( M_1 \) and \( M_2 \) are equivalent. The problem of planar graph isomorphism has therefore been reduced to the equivalence of finite automata. When this result was discovered the two problems had equivalent running times, i.e. \( O(n^2) \). However an algorithm was found which reduced the running time of the equivalence of finite automata to \( O(n \log n) \) and this reduced the running time for the planar graph isomorphism problem.

Consider an automaton, which if it scans a 0 in state \( i \) (except state \( n \) it goes into state \( i+1 \). If it sees a 1 it stays in the same state. Early algorithms for equivalence of automata, partitioned states into final and non-final states. For this automaton the partition is

\[
\begin{align*}
[ 1 & 2 & 3 & \ldots & n-1 & n ]
\end{align*}
\]

The algorithm then partitions the non-final state set into states whose next state entries are out of this set and those whose next state entries are not out of the set. This partition in the automaton just described would give
This algorithm gives a running time of $n^2$ in this example, which is the worst possible case. By doing the refinement in the reverse order, it is possible to reduce the worst case running time to $O(n \log n)$.

Rather than ask where is the next state entry, as the last algorithm did, the quicker algorithm, by computing the inverse, asks where is the previous state(s) entry. If the set \{n\} is refined in this way, the previous states are \{n-1,n\}. Since n is already in the set, but n-1 is not then n-1 is a refinement of the larger set and can therefore be taken out. The new algorithm has produced the refinement shown in Figure 31, but has had to do less calculation to obtain the refinement. Both algorithms will produce the same refinements from the initial configuration. The choice of which set to refine does not matter, so the best choice is always to refine from the smallest set, by doing this the running time of $O(n \log n)$ for the algorithm is obtained.

The planar graph isomorphism problem has subsequently been improved to $O(n)$ time.

A Vector Machine

Theory allows us to ask what is going on, and to suggest what notation to use and the questions to ask. The programmer, when he want to know the running time of an algorithm, expects the answer to reflect reality. If multiplication of arbitrary length integers could be done in unit time, then the programmer tries to use this in unforeseen ways. By coding an n-by-n matrix into such an integer and by careful use of the multiplication, the running time of the matrix multiplication routine could be reduced to $n^2$. In which case he should add on a logarithmic cost function to take care of the manipulation of the data.

Pratt, Stockmeyer and Rabin have constructed a device called 'a vector machine', which is a random access machine with a facility for performing Boolean operations on arbitrarily long vectors and a shift
operation which can be used to perform multiplication. This machine is interesting as it allows the arbitrary length integers previously mentioned. Pratt et. al. were able to show that on this machine, anything that could be done in nondeterministic polynomial time could be done in deterministic polynomial time.

A deterministic device can simulate a nondeterministic device in the same time. The following method shows how the simulation of nondeterministic Turing machine can be performed. Let \( C_1 \) be the initial configuration of a Turing machine, which on any one move has exactly two options.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Contents of the register</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 \leftarrow C_1 )</td>
<td>( C_1 )</td>
<td>Store the length of the initial configuration</td>
</tr>
<tr>
<td>( R_2 \leftarrow 2^{\left</td>
<td>C_1 \right</td>
<td>} )</td>
</tr>
<tr>
<td><strong>LOOP:</strong> ( R_3 \leftarrow R_1 \ast R_2 + R_1 )</td>
<td>( C_1 C_3 )</td>
<td>Duplicate the configuration of the machine</td>
</tr>
<tr>
<td>( R_2 \leftarrow R_3 \ast C_1 )</td>
<td>( 2^{\left</td>
<td>C_1 \right</td>
</tr>
</tbody>
</table>

by sequence of masks simulate move of \( T_m \), place result in \( R_1 \)

Goto LOOP

The deterministic machine then goes through the following configurations which are stored in \( R_1 \).

Initially \( C_1 \)

After the first loop \( C_2 C_2' \) (two possible configurations after one move of \( T_m \))

After the second loop \( C_2 C_2' C_2'' C_2''' \)
By running all possible choices in parallel, it is possible to deterministically simulate a non-deterministic machine in the same time. This example gives some insight into what operations we can and cannot do, when the correct running times of algorithms are required. In particular it suggests then allow bit vector operations on arbitrarily long bit vectors in one step will yield unrealistic results.

In conclusion, Professor Hopcroft said the following:

"I hope that I have made the case, that the days when the bright young person who is untrained can come into Computer Science and essentially do as well as the people that we were training, are probably drawing to a close and that in the future we are going to start to build on results that we have obtained before. Unless people are trained rigorously in these theoretical concepts and can work with them, I think that they are going to be left out of the really interesting aspects of Computer Science in the future".
References:


