70

AN OVERVIEW OF THE THEORY OF COMPUTATIONAL COMPLEXITY

by

J. Hartmanis

and

J.E. Hopcroft

Technical Report

No. 70-59

April 1970

Department of Computer Science
Cornell University
Ithaca, New York 14850

AN OVERVIEW OF THE THEORY OF COMPUTATIONAL COMPLEXITY*

by

J. Hartmanis and J.E. Hopcroft

Abstract

The purpose of this paper is to outline the theory of computational complexity which has emerged as a comprehensive theory during the last decade. This theory is concerned with the quantitative aspects of computations and its central theme is the measuring of the difficulty of computing functions. The paper does not attempt to give an exhaustive survey but instead presents the basic concepts, results and techniques of computational complexity from a new point of view from which the ideas are more easily understood and fit together as a coherent whole.

* This research has been supported in part by National Science Foundation Grants GJ-155 and GJ-96.

TABLE OF CONTENTS

I. Introduction

It is clear that a viable Theory of Computation must deal realistically with the quantitative aspects of computing and develo· to include a general theory which studies the properties of possible measures of the difficulty of computing functions. Such a theory must go beyond the classification of functions as computable and noncomputable, or elementary and primitive recursive, etc. This theory must concern itself with computational complexity measures which are defined for all possible computations and which assign a complexity to each computation which terminates. Furthermore, this theory must eventually reflect some aspects of real computing to justify its existence by contributing to the general development of computer science. During the last decade considerable progress has been made in the development of such a theory dealing with the complexity of computations. It is our conviction that by now this theory is an essential part of the Theory of Computation and that in the future it will be an important theory which will permeate much of the theoretical work in computer science.

The purpose of this paper is to outline the recently developed theory of computational complexity by presenting the central concepts, results and techniques of this theory. The paper is primarily concerned with the study of computational complexity measures defined for all computable partial functions and makes no attempt to exhaustively survey the whole field nor to present the material in historical order. Rather, the paper concentrates on exhibiting those results and techniques which we feel are important and presents them from a point of view from which they are most easily understood. In a way, the paper contains what we believe every computer scientist (or at least those working in the theory of computation) should know about computational complexity. On the other hand, he who wishes to do further research in this area may have to do considerably more reading. In particular he should study the results about specific complexity measures and relations between different measures which have motivated much of the general approach and remain a source for ideas and counter examples.

The first part of the paper motivates the definition of computational complexity measures and gives several examples. After that, some basic properties are derived which hold for all complexity measures. It is shown, for example, that in every complexity measure there exist arbitrarily complex zero-one functions and that there is no recursive relation between the size of a function and its complexity. On the other hand, it is shown that any two complexity measures bound each other recursively. In the following section we give a new proof of the rather surprising result which asserts that in any measure there exist functions whose computation can be arbitrarily speed up by choosing more and more efficient algorithms. This is later shown not to be true for every recursive function. Our proof is based on a direct diagonalization argument and does not rely on the Recursion Theorem which had to be used in the original proof. This is achieved by observing that the Speed-Up Theorem is measure independent; that is, if it holds in any measure it holds in all measures, and then proving it directly for the well-understood computational complexity measure of tape-bounded Turing machine computations. In this measure the proof loses much of its original difficulty. The Speed-Up Theorem has the strange implication that no matter which two universal computers we select (no matter how much faster and more powerful one of the machines is) there exist functions which cannot be computed any faster on the more powerful machine. This is so because for any algorithm which we use to compute such a function on the more powerful machine there exists another algorithm which is so fast that even on the slow machine it runs faster than the other algorithm on the faster machine.

Since there exist functions which have no "best" programs and thus we cannot classify functions by their minimal programs, we turn to the study of classes of functions whose computational complexity is bounded by a recursive function. For this study, we show for the first time here, that for any complexity class whose complexity is bounded by a recursive function f we can uniformly construct a strictly larger class whose complexity is given by a recursive function of the complexity of f (i.e. the running-time of f). The next result, the Gap Theorem, asserts that this is the best possible uniform result we can obtain by showing that there exist arbitrarily large "gaps" between the complexity classes. That is, for every recursive function r there exists an increasing recursive function t such that the class of all functions computable in the complexity bound t is identical to the class of functions computable in the complexity bound $r \circ t$. Thus we cannot always obtain larger complexity classes by applying a recursive function to the old complexity bound. This result has also the interesting implication that when we consider a universal computing machine, then no matter how much we increased the computation speed and no matter how many new operations we added, there exist infinitely many recursive complexity bounds in which the old and the new machine will compute exactly the same functions. That is, within infinitely many complexity bounds no advantage can be gained from the additional computing power and speed of the new machine over the old machine. This discussion is followed by another surprising result which shows that the complexity axioms admit complexity measures with complexity classes that cannot be recursively enumerated. Fortunately, this situation cannot prevail for large complexity classes and it is shown that in any measure all sufficiently large complexity classes are recursively enumerable.

The next section takes a more detailed look at the process of constructing new complexity classes by means of the diagonal process. Here we present a new approach which permits us to break down the "price of diagonalization" over any complexity class into the "price of simulation" and the "price of parallel computations". From this general formulation we can read-off the results about complexity classes for special measures once we know how difficult it is to "simulate" and to "parallel" two computations in a given measure. This is illustrated by deriving the three rather different looking results for the complexity classes of tape-bounded Turing machine computations as well as the results for time-bounded computations for one-tape and many-tape Turing machine models. In each case the differences in the structure of the result are traced back to the differences in the difficulty of "simulation" and "paralleling" computations for the three different complexity measures.

In the next section we look at the problem of "naming" complexity classes. First we prove the Union Theorem which asserts that the union of any recursively enumerable sequence of increasing complexity classes is again a complexity class. This implies that many previously studied subclasses of the recursive functions fit in naturally in many complexity measures. For example, there exists a recursive tape-bound $L(n)$ such that the class of functions computed by Turing machines whose tape length is bounded by $L(n)$ consists exactly of the primitive recursive functions. The second major result of this section, the Naming Theorem, takes some of the sting out of the Gap Theorem, by showing that in any measure there exists a (measured) set of functions which names all complexity classes without leaving arbitrarily large upward gaps. Unfortunately, it turns out that this naming of complexity classes may have arbitrarily large downward gaps. The Naming Theorem is a rather technical result and the proof is still quite difficult. The reader may want to skip this proof and proceed to the next section.

We conclude this overview with a discussion of the size of algorithms or machines in order to capture the notion of how complicated it is to describe an algorithm. We start by giving a formal definition of a size measure and then show that any two such measures are recursively related. The main result of this section shows that in any recursively enumerable list of algorithms there are arbitrarily inefficient representations. This result is then used to look at the economy of formalisms for representing various algorithms. For example, it is shown that when we use primitive recursive schema to represent primitive recursive functions, then there are large inefficiencies in this description of primitive recursive functions. That is, even among the shortest programs in this schema we can find programs which can be shortened by any desired amount by going to a general recursive schema. This asserts that, though we do not need a "go to" or "if" statement to compute primitive recursive functions, the use of these statements can shorten the length of our programs drastically and so clarifies their importance in programming languages.

The last section gives a very brief history of the research described in this paper and tries to indicate who did the original work. We have also included a short bibliography for possible further reading.

II. Computational Complexity Measures

The theory of computational complexity is concerned with measuring the difficulty of computations. In order to do this we must discuss what is meant by a computational complexity measure.

In this paper we are concerned with computational complexity measures which are defined for all possible computations, that is, for all partial reccursive functions mapping the integers into the integers. Therefore, to define a complexity measure we need an effective way of specifying all possible computations or algorithms (for the computation of partial recursive functions) and the complexity measure will then show how many "steps" it takes to evaluate any one of these algorithms on any specific argument.

For example, our list of algorithms or computing devices could be a standard enumeration of all one-tape Turing machines (which we know are capable of computing all partial recursive functions) and the complexity measure for a given machine M_i (or algorithm) working on an argument n could be the number of operations performed by M_i before halting on input n.

A different complexity measure is obtained when we consider (a recursive enumeration of) all ALGOL programs and again let the complexity of the i-th ALGOL program on argument n be defined by the number of instructions executed before the program halts on input n.

It should be noted that these complexity measures are associated with the algorithms and not directly with the functions they compute. The reason for this is that in computations we usually deal with algorithms which specify functions and for each computable function there are infinitely many algorithms which compute it. Furthermore, as it will be shown later, there exist functions which have no "best" algorithm and thus we can not talk of the complexity of a function as that of its best algorithm.

From the preceeding examples we see that a computational complexity measure consists of a recursive list of algorithms which compute all partial recursive functions to each of which is assigned a step-counting function which gives the amount of resource used by a given algorithm on a specific argument. The assignment of the step-counting function, furthermore, satisfies some conditions. If our list of algorithms is denoted by $\phi_1, \phi_2, \phi_3, \ldots$ and the corresponding step-counting functions by $\Phi_1, \Phi_2, \Phi_3, \ldots$, then we note that for our examples the following two conditions hold:

1. the algorithm $\phi_i(n)$ is defined if and only if $\Phi_i(n)$ is defined.
2. for any given number of steps m and any algorithm ϕ_i working on argument n we can determine (recursively) whether $\phi_i(n)$ halted in m steps, that is whether $\Phi_i(n) = m$.

In other words if the i-th Turing machine halts on input n then the number of steps it took before halting is well defined. On the other hand, if i-th machine does not halt on input n, then we cannot determine how complex the computation is since the measure is not defined. What we can do for each i and n is to determine whether the i-th machine halted on input n in m steps for any given n. Clearly, we achieve this, for our first example, by just performing m steps of the i-th computation on input n and noticing whether the computation halts on the last step.

One may impose additional conditions on the complexity measure to capture more completely some specific aspect of computational difficulty; but the conditions we have stated are so natural and basic to any notion of computational complexity that it is now generally accepted that they must hold for any computational complexity measure. The surprising fact is that they are sufficient to prove many interesting results about all complexity measures for which they hold. In a way, the rest of the paper will illustrate this although we will look at specific measures to strengthen our intuition and illustrate some special results.

At the same time it should also be observed that more applied computer scientists may be far more interested in results about the complexity of particular problems in specific measures. Nevertheless, the above outlined approach is sufficient to start developing the general theory.

We now make precise the notion of a computational complexity measure. Through out this paper we refer to a computable total function as a recursive function and we use the word algorithm for algorithmic procedure even though the procedure does not halt for all arguments.

Definition: A computational complexity measure Φ is an admissible enumeration of the partial recursive functions $\phi_1, \phi_2, \phi_3, \ldots$, to which are associated the partial recursive step-counting functions $\Phi_1, \Phi_2, \Phi_3, \ldots$, such that

1. $\phi_i(n)$ is defined iff $\Phi_i(n)$ is defined,

2. $M(i,n,m) = \begin{cases} 0 & \text{if} \quad \Phi_i(n) \neq m \\ 1 & \text{if} \quad \Phi_i(n) = m \end{cases}$ is a recursive function.

It was seen that the number of moves of a Turing machine can be used as a step-counting function to obtain a computational complexity measure. Similarly we can use the number of tape cells scanned by a Turing machine (provided we agree that if the machine does not halt the number of tape cells scanned is undefined) to define a measure. In fact most other natural measures which can be thought of do indeed satisfy the definition. Given a set of step-counting functions one can apply any recursive function $f(n)$, $f(n) \geq n$, or any recursive unbounded monotonic function to each step-counting function to obtain a new set of step-counting functions.

Nevertheless, it will be seen that the definition of computational complexity measures is restrictive enough to eliminate as step-counting functions, those functions which in no real sense measure the complexity of the computation. For the present it is instructive to consider several examples which do not form complexity measures.

A) The number of recursions used to define a function in a schema for primitive recursion cannot be used for step-counting functions since the schema is not capable of representing all partial recursive functions and thus we do not have an admissible enumeration of all algorithms.

B) The functions $\{\Phi_i(n)\}$ defined by $\Phi_i(n) = 0$ for each i and n fail to satisfy condition 1.

C) The functions $\{\Phi_i\}$ defined by

$$\Phi_i(n) = \begin{cases} 0 & \text{if } \phi_i(n) \text{ is defined} \\ \text{undefined otherwise} \end{cases}$$

do not satisfy condition 2 since for each i and n, $\phi_i(n)$ is defined if and only if $M(i,n,0) = 1$ and thus $M(i,n,m)$ cannot be recursive (otherwise we would be solving the Halting Problem).

Many results are implied by the definition of computational complexity measure. The first result is that for any measure there exist arbitrarily complex recursive functions. To establish this result we will show that for any recursive function f there exists a recursive function ϕ with the property that any possible way of computing ϕ requires more than $f(n)$ steps for infinitely many n. To construct ϕ we just have to formalize the procedure (diagonal process) which looks at each index i infinitely often with increasing n and sets

$$\phi(n) \neq \phi_i(n) \quad \text{iff} \quad \Phi_i(n) \leq f(n) .$$

<u>Notation</u>. We say that i is an index for the function ϕ provided $\phi_i(n) = \phi(n)$ for all n.

<u>Theorem 2.1</u>: Let Φ be a computational complexity measure and f any recursive function. Then there exists a recursive function ϕ such that for any index i for ϕ, $\Phi_i(n) > f(n)$ for infinitely many n.

<u>Proof</u>: Let $r(n)$ be a recursive function with the property that for all i, $i = 1,2,3,\ldots$, there are infinitely many n such that $r(n) = i$. Define

$$\phi(n) = \begin{cases} \phi_{r(n)}(n) + 1 & \text{if} \quad \Phi_{r(n)}(n) \leq f(n) \\ 0 & \text{otherwise} . \end{cases}$$

Since f and r are recursive functions (by the second condition on complexity measures) we can compute whether

$$\Phi_{r(n)}(n) \leq f(n)$$

and thus $\phi(n)$ is a recursive function. Furthermore, if j is an index for ϕ, then for the infinitely many n such that $r(n) = j$ we have that $\Phi_j(n) > f(n)$ as was to be shown.

By using a somewhat more complicated diagonal process, we next derive the stronger result which asserts that for any recursive f there exist recursive functions whose complexity exceeds f almost everywhere. To establish this result we just formalize the statement: "If the complexity $\Phi_i(n)$ of the i-th function is less than $f(n)$ for almost all n, then I am not the i-th function."

<u>Theorem 2.2</u>: Let Φ be a complexity measure. Then for any recursive function f there exists a recursive function ϕ such that for any index i for ϕ, $\Phi_i(n) > f(n)$ for almost all n.

<u>Proof</u>: Let f be any recursive function. To construct the function ϕ such that for any index j for ϕ, $\Phi_j(n) > f(n)$ for almost all n, we proceed as follows: for each n we look for the first function, smallest index i, such that $\Phi_i(n) \leq f(n)$ and make $\phi(n)$ different from $\phi_i(n)$, provided this has not been done before. More precisely, let $s(n)$ be the smallest integer less than n such that

$$\Phi_{s(n)}(n) \leq f(n)$$

and for no $m < n$ is

$$\Phi_{s(n)}(m) \leq f(m) \quad \text{with} \quad \phi_{s(n)}(m) \neq \phi(m) .$$

If no such integer exists $s(n)$ is undefined. Let

$$\phi(n) = \begin{cases} 0 & \text{if } \phi_{s(n)}(n) = 1 \\ 1 & \text{otherwise} . \end{cases}$$

Clearly $\phi(n)$ is a recursive function. Assume, $\Phi_j(n) \leq f(n)$ for infinitely many n and $\phi_j = \phi$. Eventually for some value of n, say n_o, the smallest integer k such that $\Phi_k(n_o) \leq f(n_o)$ and for no $m < n_o$ is $\Phi_i(m) \leq f(m)$ with $\phi_k(m) \neq \phi(m)$ will be j. Thus, by the definition of ϕ, $\phi(n_o) \neq \phi_j(n_o)$, a contradiction. Therefore, for any index i for ϕ, $\Phi_i(n) > f(n)$ for almost all n.

<u>Corollary</u>: There exist arbitrarily complex 0-1 valued functions in all measures.

By the previous theorem we see that there are arbitrarily complex bounded functions. From this we immediately conclude that there can be no recursive relation between functions and their complexities since such a relation would imply a bound on the complexity of any bounded function.

<u>Theorem 2.3</u>: There is no recursive function b such that for each i $b(\phi_i(n),n) \geq \Phi_i(n)$ almost everywhere.

Proof: Assume that such a b exists. Then the complexity $\Phi_i(n)$ of any zero-one function $\phi_i(n)$ must satisfy

$$\Phi_i(n) \leq b(0,n) + b(1,n) \quad \text{a.e.}$$

contradicting Theorem 2.2.

Although there is no recursive relation between the value of a function and its complexity there is a recursive relation between the complexity of an algorithm in any two measures. In other words a function which is "easy" to compute in one measure is "easy" to compute in all measures.

Theorem 2.4: Let Φ and $\hat{\Phi}$ be complexity measures. There exists a recursive r such that for any i,

$$r(n,\Phi_i(n)) \geq \hat{\Phi}_i(n) \quad \text{and} \quad r(n,\hat{\Phi}_i(n)) \geq \Phi_i(n)$$

for almost all n.

Proof: Let

$$r(n,m) = \max\{\Phi_i(n),\hat{\Phi}_i(n) \mid i \leq n \text{ and either } \Phi_i(n) = m \text{ or } \hat{\Phi}_i(n) = m\}.$$

Clearly

$$r(n,\Phi_i(n)) \geq \hat{\Phi}_i(n)$$

and

$$r(n,\hat{\Phi}_i(n)) \geq \Phi_i(n)$$

for all $n \geq i$. The function r is recursive since there is an effective procedure to determine if either $\Phi_i(n)$ or $\hat{\Phi}_i(n)$ is equal to m . Should either be equal to m , then both must be defined. Hence the maximum can be effectively computed.

Saying that two measures are recursively related does not mean much from a practical point of view since the recursive relation may be arbitrarily large. Furthermore, the recursive relation given by Theorem 2.4 may not be a tight bound since the relation given depends on the enumeration. Thus if we are comparing the number of steps of a single tape Turing machine to the number of tape cells used, the function r of Theorem 2.4 will depend on the order in which we enumerate Turing machines. However, as a consequence of the theorem, we note that the complexity of any class of functions which is bound in one measure (e.g. polynomials, primitive recursive functions, etc.) is bounded in every complexity measure.

Before we proceed to the study of more exciting results, we prove a technical lemma which shows that in any measure when we "combine" computations the complexity of the new computation is recursively bounded by the complexity of the component computations. This lemma is used repeatedly in the study of computational complexity.

<u>Lemma 2.1</u> (Combining Lemma): Let Φ be any measure and $c(i,j)$ a recursive function such that if $\phi_i(n)$ and $\phi_j(n)$ are

defined then so is $\phi_{c(i,j)}(n)$. Then there exists a recursive function h such that

$$\Phi_{c(i,j)}(n) \leq h(n,\Phi_i(n),\Phi_j(n)) \quad \text{a.e.}$$

<u>Proof</u>: Define

$$p(i,j,n,m,\ell) = \begin{cases} \Phi_{c(i,j)}(n) & \text{if} \quad \Phi_i(n) = m \quad \text{and} \quad \Phi_j(n) = \ell \\ 1 & \text{otherwise} . \end{cases}$$

This function is recursive and we obtain the desired h by setting

$$h(n,\Phi_i(n),\Phi_j(n)) = \max_{i,j\leq n} p(i,j,n,m,\ell) .$$

Clearly for $n > i,j$ we have

$$h(n,\Phi_i(n),\Phi_j(n)) \geq \Phi_{c(i,j)}(n)$$

and thus the inequality holds almost everywhere, as was to be shown.

III. Speed-Up Theorem and Applications.

We now give a new proof (without invoking the Recursion Theorem) of the rather surprising result that there exist functions which have no best algorithms. In fact we show that for any recursive function $r(n)$, there exists a recursive function $\phi(n)$ such that to every index i for ϕ there corresponds an index j for ϕ with $\Phi_i(n) \geq r(\Phi_j(n))$ for sufficiently large n.

For example, if we choose $r(n) = 2^n$, then there exists a recursive function ϕ such that if $\phi_i = \phi$ then for some other index j of ϕ we have

$$\Phi_j(n) \leq \log \Phi_i(n) \quad \text{a.e.}$$

Furthermore, this process can be repeated and we conclude that there exists an index k for ϕ such that

$$\Phi_k(n) \leq \log \log \Phi_i(n) \quad \text{a.e.}$$

and this logarithmic speed-up can be iterated arbitrarily often.

Rather than prove the most general case directly we shall first establish the result for a specific well-understood measure by a straightforward diagonalization argument and then use the fact that all measures are recursively related to obtain the general theorem. The complexity measure which we will use is based on the amount of tape used by a one-tape Turing machine. The machines are so modified that they never cycle on a finite segment of their tape and thus this measure satisfied the two conditions of our definition. Furthermore, we modify them so that $L_i(n)$ (the number of tape cells used by the i-th Turing machine on input n) is larger than the length of the description of the i-th Turing machine. Finally, we will use an enumeration of these Turing machines with the property that the i-th Turing machine has at most i different tape symbols. The advantage of selecting the amount of tape used as our complexity measure stems from the fact that tape can be reused several times for different computations which we will carry out in the desired computation of ϕ .

We say that a recursive function $f(n)$ is <u>tape-constructable</u> if and only if there exists a one-tape Turing machine which uses exactly $f(n)$ tape cells for input n , $n = 1,2,...$.

The basic idea of the following proof is quite simple. We will construct a function ϕ which cannot be computed fast by "small"machines by running a diagonal process in which the stringency of conditions will decrease with the size of the machine. More explicitely, for a properly chosen function h we will formalize the following procedure for the construction of ϕ , "if the i-th machine computes $\phi_i(n)$ on fewer than $h(n - i)$ tape cells, then ϕ is not ϕ_i" , and then show that this diagonal process is sufficiently simple that for any k we can compute ϕ on $h(n - k)$ tape cells by a sufficiently large machine.

Lemma 3.1: Let $r(n)$ be any recursive function. Then there exists a recursive function $\phi(n)$ such that for each i for which $\phi_i(n) = \phi(n)$ there exists a j with

$$\phi_j(n) = \phi(n) \quad \text{and} \quad L_i(n) \geq r(L_j(n))$$

for sufficiently large n.

Proof: Without loss of generality we can assume that $r(n)$ is a strictly increasing tape constructable function. (Otherwise replace $r(n)$ by $\hat{r}(n)$ where $\hat{r}(n)$ is a strictly increasing tape constructable function and $\hat{r}(n) \geq r(n)$ for all n.) Define $h(n)$ by $h(1) = 1$ and for $n > 1$ set

$$h(n) = r(h(n-1)) + 1 ,$$

Then

$$h(n) > r(h(n-1))$$

for all $n > 1$ and clearly h is tape constructable since $r(k)$ and $r(k) + 1$ are tape constructable and thus by induction is $h(n)$.

We will define the function $\phi(n)$ so that

(1) $\phi_i(n) = \phi(n)$ implies that $L_i(n) > h(n - i)$ a.e.,

(2) for each k there exists an index j such that $\phi_j(n) = \phi(n)$ and $L_j(n) \leq h(n - k)$.

This will insure that given an index i for ϕ there exists an index j for ϕ with

$$L_i(n) \geq r(L_j(n)) \quad \text{a.e.}$$

To achieve this, select j so that

$$L_j(n) \geq h(n - i - 1) .$$

Then

$$L_i(n) > h(n - i) > r(h(n - i - 1)) > r(L_j(n)) .$$

<u>Construction of $\phi(n)$</u>:

$$\text{Set} \quad \phi(1) = \begin{cases} \phi_1(1) + 1 & \text{if } L_1(1) < h(1) \\ 0 & \text{otherwise} . \end{cases}$$

If $L_1(1) < h(1)$, cancel first Turing machine from the list of Turing machines. For $n = 2,3,4,\ldots$, set

$$\phi(n) = \begin{cases} \phi_i(n) + 1 & \text{where } i \text{ is the smallest index not already cancelled such that } L_i(n) < h(n - i) \text{ and cancel index } i . \\ 0 & \text{if no such } i \text{ exists} . \end{cases}$$

Clearly if the i-th Turing machine computes ϕ, then

$$L_i(n) > h(n - i) \quad \text{a.e.}$$

since for sufficiently large n_o, each j which will eventually be cancelled has already been cancelled and hence if

$$L_i(n) < h(n - i)$$

for any $n > n_o$, then

$$\phi(n) = \phi_i(n) + 1 \quad \text{and hence} \quad \phi_i(n) \neq \phi(n) .$$

Furthermore, for each k there exists a Turing machine j which computes ϕ in

$$L_j(n) \leq h(n - k)$$

tape cells. Turing machine j operates as follows. For each i, $i \leq k$, which ever gets cancelled, gets cancelled for a value of $n < v$. For each $n \leq v$, j has stored in its finite control the value of $\phi(n)$ and simply prints out the appropriate value. For $n > v$, j computes the smallest i not already cancelled as follows:

1) j lays off $h(n - k)$ tape cells.
2) j has stored in finite control, the values of i cancelled on input n, $n \leq v$.
3) for $v < m \leq n$, j simulates each Turing machine i, $v < i \leq n$, determines which machine gets cancelled at each value of $m < n$, and then finds the smallest uncancelled i such that

$$L_i(n) < h(n - i)$$

and sets

$$\phi(n) = \phi_i(n) + 1 .$$

If no i exists $\phi(n)$ is set equal to zero. The simulations can be carried out in $h(n - k)$ tape cells since j simulates only machines with indices greater than v and simulates the machine i only until it exceeds $h(n - i)$ tape cells. Since machine i has at most i tape symbols, the simulation requires at most

$$ih(n - i) < h(n - k)$$

tape cells.

Next we consider arbitrary measures and show that a speed up theorem exists for all measures.

<u>Theorem 3.1</u> (Speed-Up Theorem): Let Φ be a complexity measure and $r(n)$ a recursive function. There exists a recursive function $\phi(n)$ so that for each i such that $\phi_i(n) = \phi(n)$ there exists a j for which

$$\phi_j(n) = \phi(n) \quad \text{and} \quad \Phi_i(n) \geq r(\Phi_j(n)) \quad \text{a.e.}$$

<u>Proof</u>: Since all measures are recursively related there exists a strictly monotonic R such

$$L_i \leq R(\Phi_i) \quad \text{and} \quad \Phi_j \leq R(L_j) \quad \text{a.e.}$$

provided Φ_i and L_j grow faster than n. Set

$$\hat{r}(n) = R(r(R(n))) .$$

Select ϕ by Lemma 3.1 so that for each i such that $\phi_i = \phi$ there exists an index j for ϕ with

$$L_i \geq \hat{r}(L_j) \quad \text{a.e.}$$

Then

$$R(r(\Phi_j)) \leq R(r(R(L_j))) \leq L_i \leq R(\Phi_i) \quad \text{a.e.}$$

But

$$R(r(\Phi_j)) \leq r(\Phi_i) \quad \text{a.e.}$$

and R strictly monotonic implies

$$r(\Phi_j) \leq \Phi_i \quad \text{a.e.} ,$$

as was to be shown.

In the next section we will show that not all recursive functions can be speeded-up by proving that in any measure there exists an increasing recursive function h such that for each sufficiently large running-time Φ_i there exists a function f with complexity Φ_i which cannot be speeded-up by the factor h . That is, f is computable in running-time Φ_i but not in running-time Φ_j if

$$h \circ \Phi_j(n) < \Phi_i(n) \quad \text{a.e.}$$

Thus in any measure many functions have h-best programs.

One should also observe that the speed-up is not effective in that the value of v in the construction of Lemma 3.1 is not effectively determined. This immediately raises the questions as to whether there is some other construction and another function f with an effective speed up. If one wishes a small speed up, say a linear speed up for tape bounded Turing machines, then there is an effective procedure. (Small here, of course, depends on the measure and must be less than the previously mentioned h .) Large speed ups, however, can never be effective. Since effectiveness of speed up is measure independent, we consider the amount of tape used by single tape Turing machines and show that in this measure large speed-ups are not effective. Again the outlined proof makes use of the fact that we can reuse the tape many times for different computations. Let ϕ be a function and $i_1, i_2, \ldots$ be a list of programs for ϕ with the property that if $\phi_j = \phi$, there exists a k so that $r(\Phi_{i_k}) < \Phi_j$ a.e. Here r is reasonably large and we are faced with the following problem if we assume that the list is recursively enumerable. Consider an algorithm which for input n marks off two tape cells, enumerates as much of the list $i_1, i_2, \ldots$ as will fit on half of the amount of tape marked off and then simulates successively each algorithm $\phi_{i_1}, \phi_{i_2}, \ldots$ until the simulation tries to exceed the tape marked off. If no algorithm has yet computed $\phi(n)$, then two more tape cells are marked off and the process is repeated. This way every algorithm ϕ_{i_j} on the list is tried eventually and, since the simulation of ϕ_{i_j} by our algorithm does not require more than a constant times the amount of tape used by ϕ_{i_j}, we conclude that this algorithm runs in approximately as little space (almost everywhere) as any algorithm on the list. Hence a program running in considerably less space cannot appear on the list and therefore we have no r-speed-up for the program we constructed. Thus we cannot recursively enumerate any list of algorithms for f which contain arbitrarily manifold r-speed-ups.

Since the speed-up theorem applies to all complexity measures, it can be applied to speed up computer programs.. However, in a computer program one usually wishes to compute a value of a function on a specific input or for some finite range of inputs not on all possible inputs. Reducing the asymptotic running time <u>almost everywhere</u> is not precisely what a practical computer scientist is interested in. However, the theorem does tell us that we cannot classify functions by their complexity since some functions have no intrinsic minimal complexity. What we shall do instead is to define complexity classes and study them in the next section.

Before proceeding with the study of complexity classes we will discuss one application of the speed-up theorem. Let us consider two different computers both of which can compute all partial recursive functions. One of these computers is assumed to have a very rich set of operations and, say, it can perform $(10!^{10!})!$ operations per second; the other computer is assumed to have only a few of the operations of the first machine and it can perform only one operation every hundred years. We can now use these two computers and their programs to define two computational complexity measures based on their running time measured in seconds. Furthermore, we know from Theorem 2.4 that the running times of these two machines are related by a recursive function h. Clearly, h will be a very large function, indeed. Nevertheless, the Speed-Up Theorem asserts that if we pick functions which have an r-speed-up with $r > h$, then we cannot gain any speed advantage by using the faster machine to compute these functions. Since for every program which is used on the fast machine, there exists another program computing the same function faster (for large values) on the slow machine. Note that the conclusion holds only for sufficiently large values of n and that there is no effective way of finding the program for the slow machine, nevertheless, we know that it exists.

Later we will find a related result, the Gap Theorem, which asserts that we can also exhibit an arbitrarily large recursive function such that the set of functions computed in this time bound is identical for both machines. In this case, as it will be seen, the proof does not rely on the fact that the slow machine is using better programs.

IV. Complexity Classes

In the first part of this paper we postulated two properties which any computational complexity measure must have and then derived several results about complexity measures utilizing only these postulates. We saw that in any measure there exist arbitrarily complex computable functions, that there can not exist a recursive relation between the size of computable functions and their computational complexity, that any two complexity measures are recursively related (they bound each other recursively), and finally that there exist functions whose computation can be "speed up" arbitrarily.

We now turn to the study of the classes of functions whose computational complexity is bounded by recursive functions. More precisely for any complexity measure Φ we define, for any recursive function ϕ_i, the complexity class

$$C^{\Phi}_{\phi_i} = \{\phi_j \mid \Phi_j(n) \leq \phi_i(n) \quad \text{a.e.}\} .$$

Thus $C^{\Phi}_{\phi_i}$ or C_{ϕ_i} consists of all computable functions whose complexity is bounded by ϕ_i almost everywhere.

We consider first the problem of constructing for any given C_{ϕ_i}, ϕ_i-recursive, a new complexity class which contains some new function not contained in C_{ϕ_i}.

The construction of the new class will be achieved by diagonalizing over the total functions which are computed in $\phi_i(n)$ steps. It should be observed that in our diagonalization we should look at each index infinitely often since the function ϕ_j is in C_{ϕ_i} provided $\Phi_j(n) \leq \phi_i(n)$ for sufficiently large n. Thus even if we find that for some n, $\Phi_j(n) > \phi_i(n)$, we still should check later whether the inequality is not reversed and whether we should not set $\phi \neq \phi_j$.

This is done by first selecting any recursive function r with the property that for all i, $r(n) = i$ for infinitely many n. Then define

$$\phi(n) = \begin{cases} \Phi_{r(n)}(n) + 1 & \text{if } \Phi_{r(n)}(n) \leq \phi_i(n) \\ 1 & \text{otherwise .} \end{cases}$$

If $\phi_j \in C_{\phi_i}$ then for a sufficiently large n, $\Phi_j(n) \leq \phi_i(n)$ and $r(n) = j$, which implies that $\phi \neq \phi_j$ by construction. Thus ϕ is not in C_{ϕ_i}. On the other hand by using the Combining Lemma, we know that for some recursive h and index j for ϕ

$$\Phi_j(n) \leq h[n, \Phi_{r(n)}(n), \Phi_i(n)] \quad \text{a.e.}$$

Thus we have obtained the following result.

<u>Theorem 4.1</u>: In any measure Φ there exists a recursive function H such that for each total ϕ_i there exists a function f_i,

$$f_i \notin C_{\phi_i} \quad \text{and} \quad f_i \in C_{H[n,\Phi_i(n)]}$$

The above formula can be simplified by deleting the first **argument** of $H(n,\Phi_i(n))$ for $\Phi_i(n) \geq n$. We cannot do this in general since if one of our measures is the number of tape cells used by an off-line Turing machine and $\Phi_i(n)$ is constant (in reality the Turing machine is a finite automaton), then $H(\Phi_i(n))$ would be a constant and could not bound, say, the number of **steps** performed by a Turing machine whose complexity grows at least linearly with the size of the input.

<u>Corollary</u>: In any measure Φ there exists a recursive function k such that for all total $\phi_i(n)$ such that $\Phi_i(n) \geq n$ there exists an $f_i(n)$ such that

$$f_i \notin C_{\phi_i} \quad \text{and} \quad f_i \in C_{k \circ \Phi_i}$$

By a slight modification of the previous proof we can obtain a new complexity class which properly contains the old class.

<u>Corollary</u>: In any measure Φ there exists a recursive function R such that for all ϕ_i total

$$C_{\phi_i} \underset{\neq}{\subset} C_{R[n,\Phi_i(n)]} \ .$$

Again if we consider sufficiently difficult, recursive functions ϕ_i, i.e. $\Phi_i(n) \geq n$, then we can drop the n from the above equation and obtain $C_{\phi_i} \underset{\neq}{\subset} C_{R\circ\Phi_i}$.

The previous results show that in any complexity measure we can obtain new complexity classes uniformly in the running time of any recursive ϕ_i, that is

$$C_{\phi_i} \underset{\neq}{\subset} C_{R\circ\Phi_i} \ .$$

The proof of this result was quite simple and we saw that the running time Φ_i entered the expression because we had to compute ϕ_i in order to bound the number of simulation steps of $\phi_{r(n)}(n)$. If we could bound the size of the running time Φ_i recursively to the size of the function ϕ_i, then we would have a result which would yield new complexity classes uniformly in ϕ_i instead of in the complexity Φ_i. On the other hand we know that the complexity of the function cannot be recursively bounded by the size of the function and this leads us to suspect that there is no recursive q such that

$C_{\phi_i} \subsetneq C_{q \circ \phi_i}$ for all i . The next result proves this suspicion, establishing the previous result as the strongest uniform result which we can get when we consider all total functions.

The Gap Theorem which we prove next can be viewed as an assertion that for any complexity measure the step-counting functions are sparse relative to recursive functions. The reason for this is that the second condition for complexity measures permits us to decide whether $\Phi_i(n) \leq m$ for all n , m and i . This is so strong a condition that using it we can construct for every recursive r a recursive t such that no step-counting function falls infinitely often between t and $r \circ t$. Thus insuring that whatever can be computed in bound $r \circ t$ can also be computed in bound t .

<u>Theorem 4.2 (Gap Theorem)</u>: In any complexity measure Φ for any recursive function r there exists a recursive monotonically increasing function t such that

$$C_t = C_{r \circ t} .$$

<u>Proof</u>: Let $\Phi_1, \Phi_2, \Phi_3, \ldots$ be the enumeration of the running times of the complexity measure Φ . We define the desired t inductively; let

$$t(1) = 1$$

$$t(n+1) = t(n) + m_n \text{ , where}$$

$$m_n = \min \{m \mid \text{for each } i \le n \text{ either } \Phi_i(n+1) \ge r \circ [t(n) + m] \text{ or } \Phi_i(n+1) \le t(n) + m\} .$$

The last predicate is recursive and thus we can test it for $m = 1,2,3,\ldots$. To see that this procedure will find the desired m_n , note that there are only a finite number of $\Phi_i(n+1)$, $i \le n$, and thus there exists an m so that

$$\Phi_i(n+1) \le t(n) + m$$

for all i , $1 \le i \le n$, such that $\Phi_i(n+1)$ is defined; if $\Phi_i(n+1)$ is not defined, then clearly

$$\Phi_i(n+1) \ge r \circ [t(n) + m] \quad \text{for all } m .$$

Thus a desired m exists and we conclude that $t(n)$ is a recursive function.

To have $C_t \subsetneq C_{r \circ t}$ we must find a Φ_j such that $\Phi_j(n) \le r \circ t(n)$ a.e. and not $\Phi_j(n) \le t(n)$ a.e. This is impossible by construction of t since $\Phi_j(n) \le r \circ t(n)$ a.e. implies $\Phi_j(n) \le t(n)$ a.e. Thus

$$C_t = C_{r \circ t} .$$

The Gap Theorem shows that there are arbitrarily large gaps among the recursive functions which contain no running times (infinitely often). The complexity classes defined by the functions bounding such a gap are identical. This also implies that we cannot have a recursive way of increasing every total function ϕ_i to $r \circ \phi_i$ to get a computation bound for some computation not in C_{ϕ_i}.

At the same time it is worth recalling that there exists a recursive way of increasing the running time Φ_i of any total function ϕ_i to get a bound for a computation which is not in C_{ϕ_i}, i.e. $C_{\phi_i} \subsetneq C_{R[n,\Phi_i(n)]}$. We shall now use this result to see for what subclasses of the recursive functions we can recursively increase the complexity bound to obtain new computations.

We first observe that for tape bounded computations the running times or tape bounds realized by Turing machines are computable in their own bounds. More precisely there exists a recursive function σ such that for every tape bound L_i we have

$$\phi_{\sigma(i)}(n) = L_i(n) \quad \text{and} \quad \phi_{\sigma(i)} \in C_{L_i} .$$

The function σ just produces a new machine from M_i which checks how much tape M_i used and then converts the number of tape squares into the proper (say, binary) output form.

Under proper output conventions the same is true for the time bounded computations of multitape Turing machines which suggest that measures with self-computable running times deserve special attention.

<u>Definition</u>: A computational complexity measure Φ is said to be <u>proper</u> if there exists a recursive function σ such that for all i

$$\phi_{\sigma(i)} = \Phi_i \quad \text{and} \quad \phi_{\sigma(i)} \in C_{\Phi_i} .$$

This leads us to our next result.

<u>Corollary</u>: In any proper measure Φ there exists a recursive function R such that for all i

$$C_{\Phi_i} \subsetneq C_{R[n,\Phi_i(n)]} .$$

<u>Proof</u>: Since Φ_i is Φ_i computable we can replace ϕ_i by Φ_i in the second Corollary of Theorem 4.1.

The previous result can easily be extended to all measures if we note that in any measure the size of the step-counting functions bound recursively their computational difficulty.

<u>Lemma 4.1</u>: For any measure Φ there exist two recursive functions σ and r such that for all i , $\Phi_i(n) = \phi_{\sigma(i)}(n)$ and $r[n,\Phi_i(n)] \geq \Phi_{\sigma(i)}(n)$ a.e.

Proof: The fact that for all i, m and n we can decide whether $\phi_{\sigma(i)}(n) = m$ permits us to give a proof very similar to that of the Combining Lemma. We set

$$p(i,n,m) = \begin{cases} \Phi_\sigma(i) & \text{if } \phi_{\sigma(i)}(n) = m \\ 1 & \text{otherwise} \end{cases}$$

and then let

$$r[n,m] = \max_{i \leq n} p(i,n,m)$$

It is seen that r gives the required bounding function.

Combining this result with Theorem 4.2 we get

Theorem 4.3: For any measure Φ there exists a recursive function h such that for all i

$$C_{\Phi_i} \subsetneq C_{h[n,\Phi_i(n)]} .$$

Proof: We just observe that the result of Lemma 4.1 substituted in the result of the last Corollary to Theorem 4.2 yields the desired relation.

When we look back at the last result, we see that the running times can be recursively increased to obtain bounds for new computations. This result relied primarily on the fact that we could enumerate the running times Φ_i and determine whether $\Phi_i(n) = m$. We now generalize this observation.

Definition: A set of functions $\{\gamma_i\}$ which can be recursively enumerated and for which we can decide for all i, m, n whether $\gamma_i(n) = m$, is called a measured set of functions.

Note that the running times of any complexity measure form a measured set and furthermore the property of being a measured set is not dependent on the complexity measure.

The above definition permits us to state a more general result.

Theorem 4.4: Let $\{\gamma_i\}$ be a measured set. Then in any complexity measure Φ there exists a recursive function r such that

$$C_{\gamma_i} \subsetneq C_{r[n,\gamma_i(n)]} .$$

Proof: We observe that for any measured set there exists a recursive σ and h such that

$$\phi_{\sigma(i)} = \gamma_i \quad \text{and} \quad \Phi_{\sigma(i)}(n) \leq h[n,\phi_{\sigma(i)}(n)] \quad \text{a.e.} ,$$

and then invoke the last Corollary of Theorem 4.2.

Next we take a look at the problem of enumerating all the functions in a complexity class.

We say that the complexity class C_t^Φ is recursively enumerable if there exists a recursively enumerable set of indices which contains an index for every function in C_t^Φ

and only indices for functions in C_t^Φ. Note though, that we do not insist that the algorithms named in this enumeration run in the bound t, we only require that they name functions for which there exists some algorithm running in the bound t (almost everywhere).

It turns out that the three specific complexity measures discussed in this paper have recursively enumerable complexity classes. We will show that this is the case in any complexity measure for sufficiently large complexity classes (bounds). On the other hand, we will also show that there are complexity measures for which some complexity classes cannot be recursively enumerated. This is rather surprising since it implies that there is no effective way of describing what functions are contained in these classes. It suggests that these complexity measures are rather pathological and that additional conditions should be imposed on complexity measures to eliminate these cases.

First, we observe that for any recursively enumerable set of recursive functions we can bound (a.e.) their running times.

<u>Lemma 4.2</u>: Let Φ be any measure and A a recursively enumerable set of total functions. Then there exists a recursive t such that $A \subseteq C_t$.

<u>Proof</u>: Let $i_1, i_2, i_3, \ldots$ be a recursive enumeration of A,

then define

$$t(n) = \max \{\Phi_{i_j}(n) | j \leq n\} .$$

Clearly, $t(n)$ is recursive and for each j, $\Phi_{i_j}(n) \leq t(n)$ a.e. Thus $A \subseteq C_t$.

In our next result we will use the set of functions of finite support,

$\mathcal{F} = \{\phi_i | \phi_i(n)$ is total and $\phi_i(n) = 0$ almost everywhere$\}$.

The set $\mathcal{F}$ is easily seen to be recursively enumerable and therefore in any measure there exists a recursive t such that $C^{\Phi}_{\mathcal{F}} \subseteq C^{\Phi}_t$. We now show that all complexity classes which contain C_t are recursively enumerable.

<u>Theorem 4.5</u>: Let Φ be a complexity measure and t such $\mathcal{F} \subseteq C^{\Phi}_t$. Then for each recursive $\phi_j(n) \geq t(n)$ the complexity class $C^{\Phi}_{\phi_j}$ is recursively enumerable.

<u>Proof</u>: Let $\phi_j \geq t$ and consider the recursively enumerable set $\{\phi_{p,i} | p = 0,1,2,\ldots , i = 1,2,3,\ldots\}$ with

$$\phi_{p,i}(n) = \begin{cases} \phi_i(n) & \text{if for each } k , p \leq k \leq n , \Phi_i(k) \leq \phi(k) \\ 0 & \text{otherwise.} \end{cases}$$

Thus $\phi_{p,i}$ is equal to ϕ_i if the complexity of $\Phi_i \leq \phi_j$ for all $n > p$ and is of finite support otherwise.

If

$$\Phi_i(n) \leq \phi(n) \quad \text{a.e.}$$

then for some p,

$$\phi_{p,i} = \phi_i$$

and thus every ϕ_i in $C^{\Phi}_{\phi_j}$ appears in our enumeration. Furthermore, since all functions of finite support are in $C^{\Phi}_{\phi_j}$ we can conclude that we have only functions from $C^{\Phi}_{\phi_j}$. Thus we have the desired enumeration.

The following result shows that there exist complexity measures in which "small" complexity classes cannot be recursively enumerated.

<u>Theorem 4.6</u>: There exist complexity measures Φ and recursive t such that C^{Φ}_t is not recursively enumerable.

<u>Proof</u>: Let $\phi_{i_1}, \phi_{i_2}, \ldots$ be a recursive enumeration of the constant functions such that $\phi_{i_j}(n) = j$. Define the measure Φ as follows: for all $k \neq i_j$, $j = 1,2,\ldots$, let $\Phi_k(n) \geq n$; and let

$$\Phi_{i_j}(n) = \begin{cases} 0 & \text{if } M_j(j) \text{ does not halt in } n \text{ steps} \\ n & \text{otherwise .} \end{cases}$$

Thus C_o consists of all those constant functions $\phi_{i_j}(n) = j$ for which the j-th Turing machine M_j does not halt on input j . Therefore, if $\phi_{k_1}, \phi_{k_2}, \phi_{k_3}, \ldots$ is an enumeration of C_o , then $\phi_{k_1}(1), \phi_{k_2}(1), \phi_{k_3}(1), \ldots$ is an enumeration of $\{j | M_j(j)\}$ does not halt$\}$. This is a contradiction, since this set is known (and can easily be shown) not to be recursively enumerable.

It is interesting to observe that this proof relied heavily on the fact that in this measure a finite change in ϕ_i could create an infinite change in its complexity Φ_i . If this is not the case then all complexity classes are recursively enumerable. The following result captures some of this observation.

<u>Corollary</u>: Let Φ be a measure with the property that $\phi_i = \phi_j$ a.e. implies $\phi_i \in C_t$ if and only if $\phi_j \in C_t$. Then all complexity classes of Φ are recursively enumerable.

The proof of this result is similar to the proof of Theorem 4.5.

V. Simulation and Parallelism

In this section we will take a more detailed look at the diagonalization process over complexity classes and express the complexity of this process in terms of the complexity of "simulation" and running two computations in "parallel". This approach will permit us to gain more insight into this process and to derive easily some results about specific complexity measures.

In the previous section we considered the following diagonalization process which worked for any i, provided ϕ_i was a total function, yielding a function not in C_{ϕ_i} :

$$\phi_{d(i)}(n) = \begin{cases} \phi_{r(n)}(n) + 1 & \text{if } \Phi_{r(n)}(n) < \phi_i(n) \\ 1 & \text{otherwise .} \end{cases}$$

where $r(n)$ was any computable function with the property that for each i there exist infinitely many n for which $r(n) = i$. We now change this process slightly to obtain the results previously derived for specific measures.

Instead of checking whether the computation of $\phi_{r(n)}(n)$ has used up more than $\phi_i(n)$ steps, that is, whether $\Phi_{r(n)}(n) \leq \phi_i(n)$, we construct a machine (find an algorithm) which computes $\phi_{r(n)}(n)$ and then put a bound on how long this algorithm is allowed to "sumulate" $\phi_{r(n)}(n)$. Thus in the first diagonal process we bounded the <u>number of steps simulated</u> in the computation of $\phi_{r(n)}(n)$. In the new process we will bound the <u>number of simulation steps</u> used in the computation of $\phi_{r(n)}(n)$. More precisely, let

$$\phi_S(n) = \phi_{r(n)}(n) + 1$$

and

$$\phi_{D(i)}(n) = \begin{cases} \phi_S(n) & \text{if } \Phi_S(n) < \phi_i(n) \\ 1 & \text{otherwise .} \end{cases}$$

Next we express the complexity of the simulation and the shutting off of the simulation, $\Phi_{D(i)}$, in terms of the complexity of $\Phi_{r(n)}(n)$ and $\Phi_i(n)$.

<u>Lemma 5.1</u>: In any measure Φ there exist recursive functions S and P such that

$$S[n, \Phi_{r(n)}(n)] \geq \Phi_S(n)$$

and

$$P[n, \Phi_i(n)] \geq \Phi_{D(i)}(n)$$

for all i and almost all n .

Proof: Let

$$S[n,m] = \begin{cases} \Phi_S(n) & \text{if } \Phi_{r(n)}(n) = m \\ 1 & \text{otherwise} . \end{cases}$$

To construct P, proceed as in the Combining Lemma by letting

$$P(i,n,m) = \begin{cases} \Phi_{D(i)}(n) & \text{if } \Phi_i(n) = m \\ 1 & \text{otherwise} \end{cases}$$

and then let

$$P[n,m] = \max_{i \leq n} P(i,n,m)$$

The next result gives a uniform way of constructing new complexity classes.

Theorem 5.1: For any measure there exist two recursive functions S and P such that for all recursive ϕ_i and ϕ_j

$$S[n,\phi_j(n)] < \phi_i(n)$$

implies that there exists a recursive function f such that

$$f \notin C_{\phi_j} \quad \text{but} \quad f \in C_{P[n,\Phi_i(n)]} .$$

Comment: Intuitively, the result asserts that if the time lost in simulating the ϕ_j bounded computations does not exceed ϕ_i , that is $S[n,\phi_j(n)] < \phi_i(n)$, then we can diagonalize over these computations in time ϕ_i . Furthermore, the cost to shut-off the simulator after it has used up $\phi_i(n)$ steps is related to the difficulty of computing $\phi_i(n)$, namely $\Phi_i(n)$, and is given by $P[n,\Phi_i(n)]$ which describes the difficulty of attaching the shut-off mechanism (or putting it in "parallel" with the simulation). Thus the complexity of the diagonal process does not exceed $P[n,\Phi_i(n)]$ and therefore there exists a function $f \in C_{P[n,\Phi_i(n)]}$ such that $f \notin C_{\phi_j}$.

<u>Proof</u>: Let S and P be total functions which satisfy the previous lemma and let S be nondecreasing in its second variable (i.e. $k > \ell$ implies that for all i , $S[i,k] \geq S[i,\ell]$). Let $f(n) = \phi_{D(i)}(n)$ and note that

$$P[n,\Phi_i(n)] \geq \Phi_{D(i)}(n) \quad \text{a.e.}$$

and therefore $\phi_{D(i)}(n) \in C_{P[n,\Phi_i(n)]}$.

To show that $\phi_{D(i)}(n)$ is not in C_{ϕ_j} consider any ϕ_k in C_{ϕ_j} whose complexity is bounded by ϕ_j , $\Phi_k(n) \leq \phi_j(n)$ a.e. By Lemma 5.1 we know that for sufficiently large n

$$\Phi_S(n) \leq S[n,\Phi_{r(n)}(n)] \text{ .}$$

Furthermore, by construction of r for arbitrarily large n, $r(n) = k$ and therefore $\phi_k(n) = \phi_{r(n)}(n)$ and $\Phi_k(n) = \Phi_{r(n)}(n)$. Thus for a sufficiently large n

$$\Phi_S(n) \leq S[n, \Phi_k(n)] ,$$

ans since $\Phi_k(n) \leq \phi_j(n)$ we see that

$$S[n, \Phi_k(n)] \leq S[n, \phi_j(n)] .$$

But by the hypotheses of the theorem

$$S[n, \phi_j(n)] < \phi_i(n)$$

and therefore

$$\Phi_S(n) < \phi_i(n)$$

which implies that

$$\phi_S(n) = \phi_{r(n)}(n) + 1 = \phi_k(n) + 1 .$$

But then $\phi_k(n) \neq \phi_{D(i)}$. Hence, recalling that $f(n) = \phi_{D(i)}(n)$, we have that f is in $C_{P[n, \Phi_i(n)]}$ and f is not in C_{ϕ_j}, as was to be shown.

By a slight modification of the proof of the proceeding theorem we obtain our next result.

<u>Corollary</u>: In any measure there exist two recursive functions S and P' such that for all recursive ϕ_i and ϕ_j

$$S[n, \phi_j(n)] < \phi_i(n)$$

implies that

$$C_{\phi_j} \subsetneq C_{P'[n, \Phi_i(n)]} .$$

This result establishes a sufficient condition for one complexity class to be properly contained in another. Unfortunately, again these results are not uniform in ϕ_i but only uniform in Φ_i and the Gap Theorem asserts that this is the best we can do.

To strengthen our intuition and grasp of this general approach we will now look at several specific measures. As it will be seen, for specific complexity measures we can often derive very tight bounds for the simulation time, S , and for the cost of putting two processes in parallel, P .

The specific measures under consideration are based on Turing machine computations. To simplify our reasoning and permit us to derive the results in their original form we will make some minor modifications in our complexity measures. We will view the Turing machines as recognizers of input sequences (thus they compute zero-one functions) and the parameter of the input will be its length ℓ (not the size of the number represented by the input).

A. First we outline the specific measure based on time bounded Turing machine computations.

<u>Definition</u>. A set of sequences R is $T(\ell)$ acceptable if and only if there exists a multitape Turing machine which accepts the set R and for inputs of length ℓ uses no more than $T(\ell)$ operations. To indicate that we are dealing with multitape Turing machines, we denote the class of all T-computable sets by C_T^M .

[It should be observed that the step-counting measure based on the length of the input sequence (over a k-symbol alphabet, $k \geq 2$) does not strictly satisfy the complexity axioms because M may halt on some input of length ℓ and and not on some other. Thus we cannot assign a unique running time based only on the length of the input. Our definition overcomes this by looking only at C_T^M , for recursive T and insisting that all inputs of length ℓ are processed in no more than $T(\ell)$ operations.

The purist can get out of these troubles by restricting the input to a one symbol alphabet and representing n by a sequence of $n + 1$ symbols and defining $\ell = n + 1$. Under this input convention, the following results remain unchanged.]

To utilize our previous result we must now determine a good simulation bound S and a good shut-off price P . In simulation the difficult problem is to simulate machines with arbitrarily many tapes on a machine with a fixed number of tapes. Fortunately, there exists a clever simulation method which yields a good result [1]. The proof of this result is quite hard and since it is used only once in this paper, we will not include it here.

Lemma 5.2: There exist two computable functions $r(n)$ and $c(n)$, and a two-tape Turing machine M such that

$$M(n) = M_{r(n)}(n) + 1$$

and if $M_{r(n)}(n)$ halts in t operations then $M(n)$ halts in no more than

$$c(n)\ t \log t + c(n) \text{ operations.}$$

Comments: 1. Thus for this model there exists a simulation function S such that

$$S[n,t] < c(n)\ t \log t + c(n)$$

2. The function $r(n)$, say for a three symbol input alphabet $\{0,1,a\}$, can be so chosen that it depends only on the binary prefix up until the first "a" marker and that this prefix is interpreted as a very direct encoding of a Turing machine's state table. Thus for every M_i there exists an x, $x \in (0 + 1)^*$ such that for all $y \in (0 + 1 + a)^*$

$$r(xay) = i .$$

This decoding function r has the advantage that whenever M_i is simulated its state table description is in the same form and the operations required to start and carry out a simulation step depends only on the prefix x and not on the length of the whole input. (For the "purist's" case when we use a unary input alphabet we are forced into using more subtle decoding techniques but with a bit of thought they can be supplied for the three models under consideration.)

<u>Theorem 5.2</u>: Let $T(\ell)$ be the running time of some multi-tape Turing machine. Then for every total function ϕ

$$\lim_{\ell\to\infty} \frac{\phi(\ell)\log\phi(\ell)}{T(\ell)} = 0 \quad \text{implies that} \quad C^M_{\phi(\ell)} \subsetneqq C^M_{T(\ell)} .$$

<u>Proof</u>: We outline the proof to explain the limit condition and the use of the running time. The limit condition implies that for any $\ell > 0$ and for sufficiently large

$$c\phi(\ell) \log \phi(\ell) < T(\ell)$$

Thus $C^M_\phi \subseteq C^M_T$ and for some sufficiently large n

$$S[n,\phi(\ell)] < T(\ell)$$

(where ℓ is the length of n and n is used to compute $\phi_{r(n)}(n)$)
But then we can diagonalize over all R in C^M_ϕ in $T(\ell)$ operations and, since $T(\ell)$ is a running time of some Turing machine M_T, we can run M_T on separate tapes in parallel with the simulator and shut-off the process when M_T halts. Thus $P[n,T(\ell)] = T(\ell)$ and we conclude that $\phi_{D(i)} \in C^M_T$ but $\phi_{D(i)} \notin C^M_\phi$, as was to be shown. (Viewing $\phi_{D(i)}$ as the characteristic function of a set of sequences).

B. Next we look at complexity classes defined by time-bounded one-tape Turing machines.

Definition. A set of sequences R is $T(\ell)$-acceptable by a one-tape Turing machine if and only if there exists a one-tape Turing machine which accepts R and uses no more than $T(\ell)$ operations to process inputs of length ℓ . The class of all $T(\ell)$-acceptable sets is denoted by C_T^1 .

For one-tape Turing machines the simulation problem is considerably simpler than for many-tape machines. As a matter of fact, for simple $r(n)$ functions the simulation can be carried out on one machine within a constant time of the machine simulated, that is

$$S[n,t] \leq c(n)t + c(n) .$$

This is achieved by always keeping a copy of the description of $M_{r(n)}$ near the place of the simulation (say on a separate track of the tape). Since the length of the description is fixed as is the number of tape symbols $M_{r(n)}$ can use, we see that each step of the $M_{r(n)}$ computation can be simulated in a fixed number of steps of M (including the moving of the $M_{r(n)}$ description along).

On the other hand, the shutting-off of the simulation process after $T(\ell)$ simulation operations is more difficult than for the many-tape model (where we just ran the shut-off counter in parallel on separate tapes). The difficulty comes from the fact that we have to run two independent computations on the same tape with one head. One way of doing this is to run the two processes on different tracks of the tape and move one of them, if necessary, to insure that the "head positions" of the two processes do not separate. If we do this, then we are interested in making sure that the computation which we have to move is not too long. This is achieved by choosing $T(\ell)$ to be a running time of a one-tape Turing machine using no more than $\log T(\ell)$ tape squares. For such $T(\ell)$ it can easily be seen that

$$P[n,T(\ell)] \leq T(\ell) \log T(\ell) .$$

Thus we obtain the corresponding result for one-tape machines.

<u>Theorem 5.3</u>: Let $T(\ell)$ be the running time of a one-tape Turing machine which computes $T(\ell)$ on $\log T(\ell)$ tape squares. Then for any total ϕ

$$\lim_{n\to\infty} \frac{\phi(\ell)}{T(\ell)} = 0 \text{ implies that } C_\phi^1 \subsetneq C_{T\log T}^1 .$$

Note how the two results differ in structure because for many-tape machines simulation was expensive and parallelism free, whereas for the one-tape model simulation was cheap and parallelism expensive.

C. We conclude by a look at tape-bounded computations.

<u>Definition</u>. A set of sequences R is $L(\ell)$-tape acceptable if and only if there exists a Turing machine M which accepts

R and which uses no more than $L(\ell)$ tape squares to process inputs of length ℓ. The set of all L-tape acceptable sets is denoted by C_L^T.

For tape bounded computations it can easily be shown that simulation costs only a constant times more, thus

$$S[n,\ell] \leq c(n)\ell + c(n)$$

and parallelism is free, that is

$$P[n,L(\ell)] \leq L(\ell) \quad \text{(provided } L(\ell) \text{ can be computed on } L(\ell) \text{ tape).}$$

Thus for tape bounded computations we get the following result.

<u>Theorem 5.4</u>: If $L(\ell)$ is computable on $L(\ell)$-tape then

$$\lim_{n\to\infty} \frac{\phi(\ell)}{L(\ell)} = 0 \text{ implies that } C_\phi^T \subsetneq C_L^T .$$

Thus, we see the structure of this result reflects the fact that simulation is cheap and parallelism is free for tape-bounded computations.

VI. Naming of Complexity Classes

In this section we study two related problems. The first problem arises naturally when we look at some well-known subclasses of the recursive functions, like the primitive recursive functions, and try to locate them among the complexity classes of a given measure. Usually these subclasses of the recursive functions are defined by the structure of their algorithms and it is quite reassuring that they fit in naturally among the complexity classes. We show in fact, as an application of the Union Theorem, that for many complexity measures Φ there exist recursive t such that C_t^{Φ} is exactly the set of primitive recursive functions.

The second problem of this section arises when we ask for "good" ways of naming complexity classes. Recall that the Gap Theorem asserted that in any measure for any recursive r there exists a recursive t such that $C_t = C_{r \circ t}$. Thus we can construct functions which "name" the same complexity class but which are as far apart as we wish. This seems to imply that we have chosen improper functions to name the complexity classes. It turns out that this is the case and that we can do much better. We cannot always name all complexity classes with the step-counting functions of the measure but we do show that there exists a measured set of functions which names all complexity classes.

First we show that the union of any recursively enumerable hierarchy of complexity classes (sequence of increasing complexity classes) is itself a complexity class. Let $\{f_i | i = 1,2,\ldots\}$ be a recursively enumerable set of functions such that for each i and n,

$$f_i(n) < f_{i+1}(n) .$$

It suffices to show that there exists a recursive function which is greater than $f_i(n)$ for each i and almost all n but is infinitely often less than each step counting function which is greater than $f_i(n)$ for each i and almost all n. Then the complexity class defined by the function will be $\bigcup_i C_{f_i}$.

Clearly the function $t(n) = f_n(n)$ is greater than $f_i(n)$ for each i and almost all n. However, there may exist a ϕ_j for which

(1) $\Phi_j(n) < t(n)$ for almost all n and

(2) for each i and almost all n, $\Phi_j(n) > f_i(n)$.

Thus ϕ_j is in C_t but not in the union of C_{f_i} . The way to avoid this difficulty is to guess for each j that some f_{i_j} majorizes Φ_j . If we detect that for some n,

$\Phi_j(n) > f_{i_j}(n)$, then we assign a value to $t(n)$ which is less than $\Phi_j(n)$ and guess that some larger f_i majorizes $\Phi_j(n)$. If ϕ_j is in $\bigcup_i C_{f_i}$, eventually we will find an f majorizing Φ_j and t will be greater than Φ_j almost everywhere. On the other hand, if ϕ_j is not in the union, then t will be less than Φ_j infinitely often and thus ϕ_j will not be in C_t . We formalize this intuitive idea in the proof of the Union Theorem.

<u>Theorem 6.1</u> (Union Theorem): Let $\{f_i | i = 1,2,\ldots\}$ be a recursively enumerable set of functions such that for each i and n , $f_i(n) < f_{i+1}(n)$. Then there exists a function $t(n)$ such that $C_t = \bigcup_i C_{f_i}$.

<u>Proof</u>: Construct t such that

1) for each i , $t(n) \geq f_i(n)$ a.e.

2) if for each j , $\Phi_i(n) > f_j(n)$ i.o., then $t(n) \leq \Phi_i(n)$ i.o.

In the construction of t we will maintain a list of indices $i_1, i_2, i_3, \ldots$. The list will be repeatedly updated and at the n-th step when we compute $t(n)$ the interpretation is as follows: $i_j = k$ means that currently we are guessing that $f_k \geq \Phi_j$ almost everywhere. To compute $t(n)$ we check if all our guesses are correct and if this is so then we set $t(n) = f_n(n)$ and enter an additional guess that $\Phi_n \leq f_n$ almost everywhere by setting $i_n = n$. The checking of our guesses proceeds as follows: we start with the smallest k such that $k = i_j$, and determine whether $f_k(n) \geq \Phi_j(n)$, if this condition is satisfied for all k , then we set $t(n) = f_n(n)$ and enter a new guess that $f_n \geq \Phi_n$ a.e. by setting $i_n = n$. On the other hand, if one of our guesses is wrong, $f_k(n) < \Phi_j(n)$, then we set $t(n) = f_k(n)$ and change our guess to $\Phi_j \leq f_n$ a.e. by setting $i_j = n$. In this case we also add the new guess $i_n = n$ and repeat the process for $n = n + 1$. This process is summarized more formally below:

Construction of t :

Initially i_j is undefined for each j , i.e. the list is empty. Set $n = 1$. Go to step n .

Step n: Let k be the smallest integer such that there exists a j for which the j-th item on the list is k (i.e. $i_j = k$) and $f_{i_j}(n) < \Phi_j(n)$. If more than one such j exists, select the smallest. Define $t(n) = f_{i_j}(n)$ and set $i_j = n$ and $n = n + 1$. Go to step n . If no such j exists define $t(n) = f_n(n)$. Set $i_n = n$ and set $n = n + 1$. Go to step n .

Proof that $C_t = \bigcup_i C_{f_i}$:

(1) ϕ_g in $\bigcup_i C_{f_i}$ implies that there exists an i such that

ϕ_g in C_{f_i} and therefore

$$\Phi_g < f_i \quad \text{a.e.}$$

But $t \geq f_i$ almost everywhere since eventually for each j, i_j will take on a value greater than i or i_j is such that $f_{i_j}(n)$ majorizes $\Phi_j(n)$. From this point on $t \geq f_i$. Therefore $\phi_g \in C_t$.

(2) $\phi_g \in C_t$ implies that $\Phi_g \leq t$ a.e. and thus there exists an f_k such that $f_k \geq \Phi_g$ a.e., else infinitely often i_g would be the smallest number on the list such that $f_{i_g} < \Phi_g$ and t would infinitely often be less than f_{i_g}, a contradiction. But g in C_{f_k} implies that g in $\bigcup_i C_{f_i}$, as was to be shown.

Consider now the primitive recursive functions and the complexity measure which counts the steps of a single tape Turing machine. We claim that g primitive recursive implies there exists a primitive recursive t such that g is in C_t. The reason for this is that the successor function, the zero function and projection function are in a complexity class determined by a p.r. function; the recursive function bounding the complexity of composition and recursion is p.r. and the class of p.r. functions is closed under composition.

We further claim that any complexity class determined by a p.r. function contains only p.r. functions. The reason for this is that primitive recursion is sufficient to simulate a Turing machine for a primitive recursive number of steps.

Note that we need only show that there exists a recursively enumerable sequence of primitive recursive functions such that every p.r. function is majorized in order to show that there is a time complexity class consisting of precisely the p.r. functions. If $g_1, g_2, \ldots$ is such a sequence, then $f_1, f_2, \ldots$ where

$$f_i(n) = \max \{g_1(n), g_2(n), \ldots, g_i(n)\}$$

is a sequence satisfying the Union Theorem. The desired result follows immediately and we state is as a corollary of the Union Theorem.

<u>Corollary</u>: There exists a recursive function t such that the set of functions computable on a one-tape Turing machine in the time bound t is exactly the set of primitive recursive functions. The same result holds for many-tape time-bounded Turing machines as well as for tape-bounded Turing machines.

Note that for any measure which is related by p.r. function to the number of steps on a single-tape Turing machine, the p.r. functions form a complexity class. Another interesting observation is that the complexity class consisting of precisely the primitive recursive functions cannot be named by a primitive recursive function. For if it were named by a p.r. $t(n)$, then $t(n)$ would lie in some level of the Grzegorczyk hierarchy and thus C_t would not contain higher levels. This suggests that the function naming the complexity class is very compliecated.

Furthermore, we saw from the Gap Theorem that the same complexity class can be named by radically different functions. Namely, in any measure for any recursive function r we can construct a recursive function t such that $C_t = C_{r \circ t}$. In all these cases the functions turn out to be very complicated in that their complexity differs widely from their size. This leads us to the problem of trying to name all complexity classes with functions which are not too complex.

The above observations lead us to the problem of finding a recursively enumerable set of functions which name all complexity classes of a measure and have the property that their complexity is recursively bounded by their size, that is an honest set of functions. The next result, the Naming Theorem, assures that this is always possible in that all complexity classes can be named by a measured set. Our strategy consists of picking for every recursive ϕ_t a $\phi_{t'}$ in a measured set so that $C_{\phi_t} = C_{\phi_{t'}}$. Setting $\phi_{t'} = \max\ (\Phi_{t'}, \phi_t)$ insures that $\phi_{t'}$ is honest. However, it may be the case that for some i , ϕ_i is a member of $C_{\phi_{t'}}$ but not a member of

C_{ϕ_t} . To resolve this difficulty $\phi_{t'}$ can be decreased below $\Phi_i(n)$ for infinitely many n . The values of n are selected so that $\Phi_{t'}(n)$ is also reduced to keep the function honest. We must insure in decreasing $\phi_{t'}(n)$ below $\Phi_i(n)$ that $\phi_{t'}$ does not infinitely often dip below some Φ_j which is almost everywhere less than ϕ_t . Otherwise ϕ_j is a member of C_{ϕ_t} but not a member of $C_{\phi_{t'}}$. The next theorem is proved by formalizing the above ideas.

<u>Theorem 6.2</u>: For each measure Φ **there exists a measured set** naming every complexity class.

<u>Proof</u>: We must show that there exist an $r(n)$ such that for each ϕ_t we can construct $\phi_{t'}$ with

1) $r(n,\phi_{t'}(n)) \geq \Phi_{t'}(n)$ a.e. and

2) $C_{\phi_t} = C_{\phi_{t'}}$ (i.e. $\Phi_i \leq \phi_t$ a.e. $\iff$ $\Phi_i \leq \phi_{t'}$ a.e.)

Two lists are used in the construction of $\phi_{t'}$. List 1 contains functions ϕ_i for which we discover there exists an n such that

$$\phi_t(n) < \Phi_i(n) \leq \phi_{t'}(n) .$$

The function ϕ_i is removed from this list when we assign

$$\phi_{t'}(m) \leq \Phi_i(m)$$

for some m. At stage k List 2 will contain each ϕ_j, $j \leq k$, which is not on List 1 and thus will contain each ϕ_j for which

$$\Phi_j(m) < \phi_t(m)$$

and for which we set

$$\phi_{t'}(m) < \Phi_j(m)$$

in removing some i from list 1.

For $k = 1,2,3,\ldots$ perform the following computation. Place k on list 2 with priority k (priority 1 will be highest, 2 next, and so on.)

<u>Test to determine if $\Phi_i(n) > t(n)$</u>

For each $i \leq k$ and each $n \leq k$ such that $\Phi_t(n) \leq i$ compute $\phi_t(n)$ unless computed for a smaller value of k. Place i on list 1 if not already on list 1 and assign priority i (remove from list 2 if on list 2) if $\Phi_i(n) > \phi_t(n)$.

<u>Try to force $\phi_{t'}$ below some i with high priority index on list 1</u>

Let $\sigma(k) \leq k$ be a function which takes on each integer infinitely often. Let $m = \sigma(k)$. If $\phi_{t'}(m)$ is already defined go to stage $k + 1$. Otherwise find i of highest priority on list 1 such that no index on list 2 of higher

priority actually takes as many steps on input m as ϕ_i .
If an i is found, remove i from list 1 and place on list 2 with priority k . Go to stage $k+1$. However, if attempt to find either i requires more than k steps, go to stage $k+1$, or (2) requires more than Φ_t , set $\phi_{t'} = \max\{\Phi_t, \phi_t\}$ and go to stage $k+1$.

To see that $C_{\phi_t} = C_{\phi_{t'}}$ consider:

1) ϕ_i in C_{ϕ_t} but not in $C_{\phi_{t'}}$

In this case

$$\Phi_i(n) \leq \phi_t(n) \text{ a.e. and } \Phi_i(n) > \phi_{t'}(n) \text{ i.o.}$$

The index i can be placed on list 1 at most a finite number of times (only once for each n such that $\Phi_i(n) > \phi_t$ n)). Eventually, each index both on list 1 and on list 2 of higher priority which will ever be removed will be removed. Then i will be removed from list 1 unless there always exists a higher priority index j on list 2 which takes longer to compute. If such an index always exists, $\phi_{t'}$ cannot dip below Φ_j and hence infinitely often cannot dip below ϕ_i since $\Phi_i \leq \Phi_j$. If no higher priority index on list 2 takes longer to compute, then i gets removed from list 1 and is placed on list 2. Eventually all higher priority indices on list 1 which will ever be removed are removed and then ϕ_t cannot dip below Φ_i . Thus $\phi_{t'} > \Phi_i$ a.e. ; a contradiction.

2) ϕ_i in $C_{\phi_{t'}}$ but not in C_{ϕ_t} .

In this case

$$\Phi_i(n) \leq \phi_{t'}(n) \quad \text{a.e.} \quad \text{and} \quad \Phi_i(n) > \phi_t(n) \quad \text{i.o.}$$

Thus index i is on list 1 for infinitely many steps. Either i is infinitely often removed from list 1 (in which case $\phi_{t'} < \Phi_i(n)$ i.o. ; a contradiction) or i remains on list 1 forevermore. Eventually, all indices on list 1 of priority higher than i which will ever be removed are removed. Similarly any index on list 2 of higher priority which will ever be removed will have been removed. Then $\Phi_i < \phi_t$ forevermore (contradicting $\Phi_i \geq \phi_t$ i.o.) since some index on list 2 of higher priority which is never removed from list 2 takes more steps.

To see that $\phi_{t'}$ is honest, if $\phi_{t'}(n)$ is assigned a value because more than Φ_t steps are needed, then $\phi_{t'}(n) = \Phi_t(n)$, otherwise $\phi_{t'}$ is independent of ϕ_t .

VIII. Size of Machines

We conclude the study of computational complexity by establishing some relations between the size of algorithms or machines and their efficiency. Just as we abstracted the notion of complexity of an algorithm we can abstract the notion of the size of an algorithm. What we have in mind is to capture the notion of how complicated it is to describe an algorithm. The size of a computer program might be measured by the number of statements and the size of a Turing machine by the state-tape symbol product.

Definition. Let s be a recursive mapping of integers into integers. We say that s is a measure of the size of machines for an admissible enumeration of all partial recursive functions $\phi_1, \phi_2, \phi_3, \ldots$, provided that

1) for each j there exist finitely many indices i such that $s(i) = j$;
2) there exists a recursive function giving the size of each algorithm; and
3) there exists a recursive function giving the number of algorithms of each size.

Consider representing algorithms by strings of symbols from some finite alphabet and let the size of the algorithm be the number of symbols. The first axiom captures the fact that there are a finite number of strings of symbols of any given length. The second axiom corresponds to counting the number of symbols in a string and the third axiom captures the notion that we can check the format of a string of symbols to determine if it represents an algorithm. Thus the number of syntactically correct strings of any given length can be computed. The three axioms are equivalent to an effective enumeration of algorithms in order of increasing size (among representations of the same size, order is unimportant).

It is easily shown that all measures of size are recursively related.

Theorem 7.1: Let s and $\hat{s}$ be two measures of size. There exists a recursive function g such that

$$g(s(i)) \geq \hat{s}(i) \quad \text{and} \quad g(\hat{s}(i)) \geq s(i)$$

for all i .

Proof: Let

$$g(m) = \max_{j \in S_m} \{s(i), \hat{s}(i)\}$$

where

$$S_m = \{i \mid \text{either } s(i) \leq m \text{ or } \hat{s}(i) \leq m\} .$$

Since S_m is a finite set for each m and since s and $\hat{s}$ are recursive, g is a recursive function. Clearly

$$g(s(i)) \geq \hat{s}(i) \quad \text{and} \quad g(\hat{s}(i)) \geq s(i)$$

for all i .

The reason for considering size of algorithms is to study the economy of various formalisms for representing algorithmic processes. In writing computer programs for functions which arise in practical situations one can dispense with conditional transfer statements and write a program where the flow of execution is determined by a very simple nested loop structure. Furthermore, the running time does not differ much from that of an arbitrary program. This raises the question as to why use conditional transfers at all. The answer lies in efficiency of representation. As an example, compare the size of the representation of a primitive recursive function using a primitive recursive schema versus the representation by means of a Turing machine which computes the function. Given an arbitrary recursive function f, we can exhibit a primitive recursive ϕ such that the minimum number of symbols in any primitive recursive schema for ϕ is larger than $f(m)$ where m is the number of symbols used to describe a certain Turing machine computing ϕ.

To obtain the result we first prove that in any infinite sequence of algorithms there are inefficient representations.

<u>Theorem 7.2</u>: Let g be a recursive function with infinite range (g enumerates indices of an infinite sequence of algorithms). Let f be a recursive function. There exist i and j such that

1) $\phi_i = \phi_{g(j)}$
2) $f(s(i)) < s(g(j))$.

NOTE: The intuitive idea behind the theorem should be transparent. Since there are a finite number of algorithms of any given size, it follows that in any infinite r.e. sequence of algorithms, there is an infinite r.e. sequence where the size of algorithms grows as rapidly as we like. Let g enumerate the rapidly growing subsequence. Given k, $\phi_{g(k)}(n)$ can be computed by a fixed size program. Namely

$$\phi(k,n) = \phi_{g(k)}(n) .$$

Thus, the size of programs to compute $\phi_{g(k)}$ need only grow at the rate needed to compute k at the same time the corresponding programs on the r.e. list grow very rapidly and the difference in the length of the two representations becomes arbitrarily large.

Proof: Since size reduction is measure independent (i.e. if there is arbitrary size reduction in one measure, then there is arbitrary size reduction in all measures) we need only prove the theorem for the case where $s(i)$ is the state symbol product of the i-th Turing machine. Without loss of generality, assume

$$s(g(n+1)) > f(s(g(n))) .$$

(Since there exist only a finite number of machines of any size, simply delete machines from the sequence determined by g until a large enough machine comes along.) Consider the algorithm $\phi_{i(k)}$ which writes k on its tape, computes $g(k)$ and then computes $\phi_{g(k)}$. The size of $i(k)$ is a constant plus k (i.e. size of Turing machine computing g plus size of universal machine to simulate ϕ_g plus states to store k.) Now $\phi_{i(k)} = \phi_{g(k)}$. Increasing k by 1 increases size of $i(k)$ by 1 and $g(k)$ by f. Thus, for sufficiently large k

$$f(s(i(k))) < s(g(k)) .$$

Let $j = k$ and $i = i(k)$ to complete the proof.

As a consequence of Theorem 7.2, there exists a primitive recursive function whose smallest primitive recursive schema is much larger than a general recursive algorithm for computing the function. Each primitive recursive function has at least one smallest primitive recursive definition. The smallest definitions are recursively enumerable. (First enumerate the smallest scheme. Start evaluating the function computed by the i-th schema on input n for larger and larger i and n. Enumerate a schema whehever it is discovered that it computes a function which differs from all functions of smaller schema.) Let g enumerate the smallest schema. Let $f(n) = n^2$ and applying Theorem 7.2 we get a primitive recursive function ϕ whose smallest primitive recursive schema has the square of the number of symbols of some general recursive algorithm for ϕ. Note that we could have selected any r.e. class of recursive functions instead of the primitive recursive functions and obtained the same result.

VIII. Historical Notes

One can detect interest in the difficulty of computations in much of mathematics where we can find algorithms defined, analyzed and compared for their efficiency. On the other hand, hardly any of this constituted a systematic attempt to develop a theory of computational complexity which would study the quantitative problems in computing. The complexity problems were originally not well-defined but even during the rapid development of constructive mathematics in the first part of this century, they were not viewed as a separate problem area. During this time several classifications of subclasses of recursive functions were defined and investigated but the main interest was in uniformly constructing larger and larger subsets of the recursive functions rather than the study of the intrinsic computational complexity of functions. The emergence of electronic computing and the general developments of computer science no doubt emphasized the need for a quantitative theory of computing and recursive function theory provided the formalism and initial models for this theory.

The first systematic attempts to define complexity measures for all computable functions and to start the development of a theory of computational complexity were initiated by Rabin [2], Hartmanis and Stearns [3] and Cobham [4]. Rabin's unpublished report, the Hartmanis and Stearns paper, and Cobham's conference paper clearly stated the importance of this subject and derived enough results to be considered a "call to arms" for computational complexity and named the field as well.

The general axiomatic approach to computational complexity used in this overview was formulated by Blum [5] who also derived most of the results described in Section II. The Speed-Up Theorem is also due to Blum though the proof of this result given in Section III is new. It differs from the original proof in that no use is made of the Recursion Theorem which (we believe) obscured the simplicity of the central diagonal process. The observation expressed in Theorem 4.1 is new and is used further in Section V. The Gap Theorem is due to Borodin [6] and it shows that Theorem 4.1 cannot be improved. The Gap Theorem thus gives a beautiful justification for Blum's use of measured sets of functions which enter Theorems 4.3 and 4.4. The recursive enumerability of complexity classes was studied by Hartmanis and Stearns [3] and Young [7]. The proof that in some measures there exist complexity classes which are not recursively enumerable is due to F. Lewis [8] and Robertson and Landweber [9].

The material about simulation and parallelism of Section V is new and the results about time-bounded and tape-bounded complexity classes in this section are due to Hennie and Stearns [1], Hartmanis and Stearns [3], Hartmanis [10], and Stearns, Hartmanis and Lewis [11].

The Union Theorem and the Naming Theorem in Section VI are due to McCreight and Meyer [12]. The result that the Naming

Theorem still leaves arbitrarily large downward gaps is due to Constable [13]. The material on the size of machines was derived by Blum [14].

The development of computational complexity has further been influenced by many papers and results which have not been explicitly used in this section. Some of them are listed in our short bibliography.

IX. References and Bibliography

A. **References:**

[1] Hennie, F.C., and R.E. Stearns, "Two-tape simulation of multi-tape Turing machines," JACM 13 (1966), 533-546.

[2] Rabin, M.O., "Degrees of difficulty of computing a function and a partial ordering of recursive sets," Technical Report 2, Hebrew University, Jerusalem, Israel, 1960.

[3] Hartmanis, J., and R.E. Stearns, " On the computational complexity of algorithms," Transactions of the American Mathematical Society 117 (1965), 285-306.

[4] Cobham, A., "The intrinsic computational difficulty of functions," Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science, ed. Y. Bar-Hillel, North-Holland, Amsterdam, 24-30, 1964.

[5] Blum, M., "A machine-independent theory of the complexity of recursive functions," JACM 14 (1967), 322-336.

[6] Borodin, A., "Complexity classes of recursive functions and the existence of complexity gaps," Conference Record of ACM Symposium on Theory of Computing, 67-78, 1969.

[7] Young, P.R., "Toward a theory of enumerations," JACM 16 (1969), 328-348.

[8] Lewis, Forbes, "Unsolvability considerations in computational complexity," Second Annual ACM Symposium on Theory of Computing, 1970.

[9] Robertson, E.L., and H.H. Landweber, " On recursive properties of complexity classes," Second Annual ACM Symposium on Theory of Computing, 1970.

[10] Hartmanis, J., "Computational complexity of one-tape Turing machine computations," JACM 15 (1968), 325-339.

[11] Stearns, R.E., J. Hartmanis, and P.M. Lewis II,"Hierarchies of memory limited computations," 1965 IEEE Conference Record on Switching Circuit Theory and Logical Design, 179-190, 1965.

[12] McCreight, E.M., and A.R. Meyer, "Classes of computable functions defined by bounds on computation: preliminary report," Conference Record of ACM Symposium on Theory of Computing, 79-88, 1969.

[13] Constable, R.L., "Upward and downward diagonalization over axiomatic complexity classes," Technical Report No. 69-32, Department of Computer Science, Cornell University, 1969.

[14] Blum, M., "On the size of machines," Information and Control 11 (1967), 257-265.

B. Bibliography:

Axt, P., "Enumeration and the Grzegorczyk hierarchy," Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik 9 (1963), 53-65.

Becvar, J., "Real-time and complexity problems in automata theory," Kybernetika 1 (1965), 475-497.

Blum., M., "On effective procedures for speeding up algorithms," Conference Record of ACM Symposium on Theory of Computing, 43-53, 1969.

Borodin, A., R.L. Constable, and J.E. Hopcroft, "Dense and non-dense families of complexity classes," IEEE Conference Record of 1969 Tenth Annual Symposium on Switching and Automata Theory, 7-19, 1969.

Cobham, A., "On the Hartmanis-Stearns problem for a class of tag machines," IEEE Conference Record of 1968 Ninth Annual Symposium on Switching and Automata Theory, 51-60, 1968.

Constable, R.L., "The operator gap," IEEE Conference Record of 1969 Tenth Annual Symposium on Switching and Automata Theory, 20-26, 1969.

Fischer, P.C., "Multi-tape and infinite-state automata - a survey," CACM 8 (1965), 799-805.

Fischer, P.C., " The reduction of tape reversals for off-line one-tape Turing machines," Journal of Computer and System Sciences 2 (1968), 136-147.

Fischer, P.C., J. Hartmanis and M. Blum, "Tape reversal complexity hierarchies," IEEE Conference Record of the 1968 Ninth Annual Symposium on Switching and Automata Theory, 373-382, 1968.

Grzegorczyk, A., "Some classes of recursive functions," Rozprawy Matematyczne 4 (1953), Warsaw, 1-45.

Hartmanis, J., "Tape reversal bounded Turing machine computations," Journal of Computer and System Sciences 2 (1968), 117-135.

Hennie, F.C., "One-tape, off-line Turing machine computations," Information and Control 8 (1965), 553-578.

Hennie, F.C., "Crossing sequences and off-line Turing machine computations," 1965 IEEE Conference Record on Switching Circuit Theory and Logical Design, 168-172, 1965.

Hopcroft, J.E., and J.D. Ullman, "Relation between time and tape complexities," JACM 15 (1968), 414-427.

Hopcroft, J.E., and J.D. Ullman, "Some results on tape bounded Turing machines," JACM 16 (1969), 168-188.

Karp, R., "Some bounds on the storage requirements of sequential machines and Turing machines," JACM 14 (1967), 478-489.

Lewis, II, P.M., R.E. Stearns, and J. Hartmanis, "Memory bounds for recognition of context-free and context-sensitive languages," 1965 IEEE Conference Record on Switching Circuit Theory and Logical Design, 191-202, 1965.

McCreight, E.M., "Classes of computable functions defined by bounds on computation," Doctoral Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., 1969.

Meyer, A.R., and D.M. Ritchie, "The complexity of loop programs," Proceedings of 22nd National Conference, Association for Computing Machinery, 465-469, 1967.

Rabin, M.O., "Real time computation," Israel Journal of Mathematics 1 (1964), 203-211.

Ritchie, R.W., "Classes of predictably computable functions," Transactions of the American Mathematical Society 106 (1963), 139-173.

Savitch, W.J., "Deterministic simulation of non-deterministic Turing machines (detailed abstract)," Conference Record of the ACM Symposium on Theory of Computing, 247-248, 1969.

Trakhtenbrot, B.A., "Turing computations with logorithmic delay," Algebra i Logika 3,4 (1964), 33-48 (in Russian).

Yamada, H., "Real-time computation and recursive functions not real-time computable," IRE Transactions on Electronic Computers EC-11 (1962), 753-760.