stichting
mathematisch
centrum

MC

AFDELING ZUIVERE WISKUNDE ZW 7/71 AUGUST

P. VAN EMDE BOAS

A NOTE ON THE McCREIGHT-MEYER NAMING THEOREM IN THE THEORY OF COMPUTATIONAL COMPLEXITY

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

Contents

Abstract: The naming theorem of McCreight & Meyer stating that in any complexity measure all complexity classes are named by functions taken from a measured set is proved without relying on the fallacious "equivalence" between the notions of honestness and measuredness as suggested by earlier proofs (see [1] and [2]).
The different "time bounds" used in the algorithm are shown to be independent of the abstract measure dealt with in the theorem. The explicit construction of the algorithm as represented by an ALGOL program indicates that the transformation of the old timebound to the new one is an effective transformation of programs. The earlier described algorithms are shown to be not completely correct.

Acknowledgements: I am indebted to prof.dr. J. Hartmanis and the faculty of the Computer Science Department of Cornell University for the opportunity they have given to me to study at their institute and for the valuable discussions we had at that time, which have directed me into the research covered by this report. I am also indebted to Mr. L.G.L.Th. Meertens for a large number of valuable suggestions and remarks which have greatly influenced the presentation of sections 3b) and 3c) and Appendix 1.

§0. Introduction

This paper discusses the so called naming theorem of McCreight and Meyer which states that in any complexity measure there exist a measured set naming all complexity classes. This theorem is formulated in [1] and [2]; in both papers a sketch of a proof is given. A complete proof is given in the Ph. D. thesis of E.M. McCreight.

In both proofs a construction is described which produces for every time bound t a new time bound t' such that

i) $C^{\Phi}_{t} = C^{\Phi}_{t'}$

ii) there exists a recursive function R(-,-) such that for almost all x we have

$$\Phi_{t'}(x) \leq R(x, t'(x))$$

ii) means in fact that the family of time bounds t' is an R-honest family. From this it is concluded that the family of t' is measured - a conclusion which could be motivated by the contents of fact (6,7) from [1].

The proof in [2] gives a clear argument for i) but does not indicate now to prove ii); in [1] the construction is described more explicitly but the proof of i) and ii) is left to the reader.

The present paper presents the following critical remarks:

A): The statement in fact (6,7) in [1] is incorrect. We construct an example of a family of functions which is 0 - honest but not measured. A weaker form of the statement is formulated and proved (section 2).

B): The construction of the time bound t' as given in [1] and [2] uses does not yield in general that $C_t = C_{t'}$. This is shown by a counterexample (section 4). We give however a modification of the construction which does work (section 3).

The modification is based on a careful analysis of the role played by the notion of complexity at different stages in the proof. We indicate three places where time bounds play an essential role. The construction gives however a complete freedom for the complexity measure used at two of these three places. The proofs in [1] and [2] are repaired by either replacing a certain time bound and/or by replacing the measure with respect to which the bound is applied.

Further the paper presents an ALGOL 60 program for the algorithm used in the construction. As the old time bound t is introduced in the Algorithm by a parameter i which is in fact an index of a program computing t we conclude that the transformation $t \rightarrow t'$ is effective. It is however not a recursive operator as it is defined on programs and not on functions (cf. the discussion on this subject in [4]).

§1. Basic definitions and results

Let $\mathbb{N}$ denote the set of natural numbers including 0. A function is a partial recursive function in one or more variables. The function is called total if it converges for all possible arguments.

$\mathcal{P}_n$ denotes the set of n-variable functions and
$\mathcal{R}_n$ denotes the subset of $\mathcal{P}_n$ of total functions

The usual inequality $\leq$ for natural numbers is extended so that it is meaningful also if one substitutes at one of both sides an expression f(x) where f does not converge for the argument x. If f(x) is undefined one holds $k \leq f(x)$ true for every finite k; $f(x) \leq k$ is always false; finally $f(x) \leq g(y)$ is true if both f(x) and g(y) are undefined.

A predicate P(n) is assumed to be recursive. It is said to hold infinitely often (i,o) provided P(n) is true for an infinite number of integers n. It is said to hold almost everywhere (a.e) if it is false for at most a finite number of integers: We denote these also by $\overset{\infty}{\exists}_n$:= for (i.o) n and $\overset{\infty}{\forall}_n$:= for (a.e) n.

By applying some fixed pairing and unpairing function the functions in more than one variable can be considered to be one-variable-functions. This fact is used to generalize notions which are defined only properly for one-argument-functions.

In the definition of a complexity measure the notion of an acceptable or effective enumeration of the collection of all algorithms appears. This is an enumeration which is recursively equivalent to the standard Gödel numbering of Turing machines, or more abstractly an enumeration for which the universal machine theorem and the S-n-m theorem holds. (See Rogers Theory of recursive functions and effective computability). The consequence of this acceptability is that a number of natural effective operations on machines (algorithms) are described by applying a recursive function to the indices of the machines.

Definition: A complexity measure [Blum measure] Φ is a pair consisting of

1^e) An admissible enumeration of all algorithms for P_1 say $\{\phi_i\}_{i=1}^{\infty}$

2^e) A sequence of recursive functions $\{\Phi_i\}_{i=1}^{\infty}$ called step-counting-functions

satisfying the following axioms:

AI: $\forall_{i,x}$ $\phi_i(x)$ converges iff $\Phi_i(x)$ converges

AII: The predicate $Q(i,x,y) = [\Phi_i(x)=y]$ is recursive and total.

Remarks:

1: We use essentially an enumeration of algorithms and not of functions. This means that the function f may be computed by distinct algorithms ϕ_i and ϕ_j; we say that i is an index for f.

2: Standard examples are the so called time or tape bounds for the collection of all Turing machines. In these examples the axioms are clear. A I means that the time or tape used by an algorithm is defined if and only if the algorithm stops; A II claims that one can decide whether or not a given algorithm on a given input uses exactly a given number of steps resp. squares on the tapes. (In the case of the tape bounds beware for algorithms which are cycling infinitely on a finite amount of tape: see also [3]).

3: From A II it follows that also the predicates $\Phi_i(x) \leq y$ and $\Phi_i(x) > y$ are decidable.

A consequence of the definitions is:

Bropostition (1.1): There exists a total recursive function t(i) such that Φ_i is computed by $\phi_{t(i)}$.

Proof: By the S-n-m theorem there exists a function t_1 such that

$$\phi_{t_1(i)}(x,y) = \underline{\text{if}}\ Q(i,x,y)\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0;$$

because the enumeration is admissible, application of the "least number operation" μ is given by a recursive function on the indices. Therefore there exists a recursive function t_2 such that

$$\phi_{t_2(i)}(x) = \mu_y[\phi_{t_1(i)}(x,y) = 1] .$$

It is clear that $\phi_{t_2(i)}$ computes Φ_i.

Given one or more complexity measures one can trivially define new measures by means of the following proposition 5:

Proposition (1.2): Let Φ, Φ^* be complexity measures, with the same enumeration $\{\phi_i\}$.
Let J be a recursive set of indices naming algorithms which compute total functions and let h be a monotoneously increasing function (or more generally a total function such that for every x the set $h^{-1}(x)$ is recursive, and included by an interval $[0,k(x)]$ for some total k).
Now the following expressions represent complexity measures

i) $h \circ \Phi = \{\{\phi_i\}, \{h \circ \Phi_i\}\}$.

ii) $\Phi + \Phi^* = \{\{\phi_i\}, \{\Phi_i + \Phi_i^*\}\}$.

iii) $\underline{\text{if}}\ i \in J\ \underline{\text{then}}\ \Phi\ \underline{\text{else}}\ \Phi^* = \{\{\phi_i\}, \{\underline{\text{if}}\ i \in J\ \underline{\text{then}}\ \Phi_i\ \underline{\text{else}}\ \Phi_i^*\}\}$

iv) $\underline{\text{if}}\ i \in J\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ \Phi = \{\{\phi_i\}, \{\underline{\text{if}}\ i \in J\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ \Phi_i\}\}$.

In all cases Axiom I is clearly satisfied. To verify Axiom II one finds new decision procedures by applying old ones either a finite number of times in cases which are a priori computable or by applying one of the old procedures depending on some decidable condition.
iv) shows that one may give away a recursive set of total functions "free of charge". ii) and i) make it possible to have arbitrarily large (expensive) complexity measures.

Although complexity is only defined for one-variable functions we suppose in the sequel that the definition is extended to functions in more than one variable.

One needs in this generalisation an admissible enumeration of all algorithms for functions in $\bigcup_{i=0}^{\infty} P_i$. Furthermore the number of variables has to be recursive in terms of the index. The step counting functions are recursive functions in the same number of variables as the functions they are measuring. i.e. $\phi_i \in P_k$ iff $\Phi_i \in P_k$.
Finally the predicate Q becomes for functions in P_k a predicate Q_k in k+2 variables which is defined only for those indices i which denote algorithms for functions in P_k. i.e.

$$Q_k(i,x_1,\dots,x_k,y) = \underline{\text{if}}\ \phi_i \in P_k\ \underline{\text{then}}\ \Phi_i(x_1,\dots,x_k) = y$$

$$\underline{\text{else}}\ \text{undefined.}$$

The quantifiers $\overset{\infty}{\exists}$ and $\overset{\infty}{\forall}$ have to be interpreted as "there exist infinitely many k-tuples" and "for all k-tuples except a finite number". The reader should note that

$$\overset{\infty}{\forall}_{x,y}\ P(x,y) \underset{\not\Longleftarrow}{\Longrightarrow} \overset{\infty}{\forall}_y \forall_x\ P(x,y) \underset{\not\Longleftarrow}{\Longrightarrow} \forall_x \overset{\infty}{\forall}_y\ P(x,y) \underset{\not\Longleftarrow}{\Longrightarrow} \overset{\infty}{\forall}_x \overset{\infty}{\forall}_y\ P(x,y)\ .$$

An important consequence of the definitions is the theorem which states that all complexity measures bound each other recursively: i.e.

<u>Theorem</u> (1.3): For every two complexity measures Φ and Φ^* with the same enumeration $\{\phi_i\}$ there exists a recursive function R(-,-) such that

$$\forall_i \overset{\infty}{\forall}_x \qquad \Phi_i(x) \le R(x,\Phi_i^*(x)) \quad \text{and}$$

$$\forall_i \overset{\infty}{\forall}_x \qquad \Phi_i^*(x) \le R(x,\Phi_i(x))\ .$$

<u>Proof</u>: Put $R(x,y) := \max\limits_{i\le x} (\underline{\text{if}}\ \Phi_i(x) = y\ \underline{\text{or}}\ \Phi_i^*(x) = y\ \underline{\text{then}}\ \max(\Phi_i(x),\Phi_i^*(x))$

$$\underline{\text{else}}\ 0\ )$$

R is recursive: if one finds that $\Phi_i(x) = y$ for some $i \le x$ then the other $\Phi_i^*(x)$ is defined and the maximum can be computed.

Clearly one has

$$R(x,\Phi_i(x)) \geq \Phi_i^*(x) \quad \text{and}$$

$$R(x,\Phi_i^*(x)) \geq \Phi_i(x)$$

provided $x \geq i$.

One easily generalize Theorem (1.3) for functions in more variables:

Theorem (1.3'): For every two complexity measures Φ and Φ^* there exist functions $S_i \in R_{i+1}$ such that for every i such that $\phi_i \in P_k$ one has

$$\overset{\infty}{\forall}_{x_1..x_k} \; S(x_1,\ldots,x_k,\Phi_i(x_1,\ldots,x_k)) \geq \Phi_i^*(x_1,\ldots,x_k) \qquad \underline{\text{and}}$$
$$S(x_1,\ldots,x_k,\Phi_i^*(x_1,\ldots,x_k)) \geq \Phi_i(x_1,\ldots,x_k) \; .$$

Theorem (1.3) makes it possible to bound the Φ-complexity of algorithms of which it is known that the "number of steps" taken during execution is bounded in some intuitively clear way, for example in terms of the values of the computed functions.

First find a recursive function t which gives for every algorithm ϕ_i a model $\psi_{t(i)}$ in terms of Turing machines or some other class of idealized sequential machines. Then the number of steps taken by $\psi_{t(i)}$ on input x say $T\psi_{t(i)}(x)$ is a complexity measure on the enumeration ϕ_i.

By Th. 1.3 we can bound $\Phi_i(x)$ in terms of x and $T\psi_{t(i)}(x)$. Consequently: if we can bound $T\psi_{t(i)}(x)$ by some function h(x) we can bound also $\Phi_i(x)$ in terms of h.

The next lemma gives an application of this strategy to functions computed by application of the least number operation. This lemma states that for a class of recursive functions which is computed by applying the least number operation on some recursively enumerable series of predicates the complexity is recursively bounded in terms of the arguments and the value of the computed functions. In the lemma one might replace the single variable x by a vector of variables. The parameter j shows that the lemma also gives some uniform bound S.

<u>Lemma</u> (1.4): Let $M(x,y,j)$ be a total recursive predicate in x,y and j (which is considered a parameter). Then there exist a recursive function $R \in \mathcal{R}_2$ and an algorithm ϕ_L computing the function:

$$L(x,j) := \mu_y[M(x,y,j)]$$

for which one has

$$\forall_j \overset{\infty}{\forall}_x \quad \Phi_L(x,j) \leq R(x,L(x,j)) \text{ .}$$

<u>Proof</u>: As indicated above there exists a sequential model for every algorithm for which the number of steps executed gives a time complexity measure. We indicate this time measure by $T\phi_i$. Construct functions S_i as in Th. 1.3' to bound $T\phi_i$ in terms of Φ_i and vice versa.
So

$$\forall_{M' \in \mathcal{P}_3} \overset{\infty}{\forall}_{x,y,j} \quad T\phi_{M'}(x,y,j) \leq S_3(x,y,j,\Phi_{M'}(x,y,j)) \text{ .}$$

We may assume S_3 to be monotonous in the last variable - otherwise replace S_3 by

$$S_3^*(x,y,j,z) = \max_{z' \leq z} S_3(x,y,j,z') \text{ .}$$

Next we choose for L the algorithm performing the computation:

```
integer procedure L(x,j);
      begin m := 0;
            while ¬M(x,m,j) do m := m+1;
            L := m
      end
```

Clearly we have in our time complexity:

$$\forall_{x,j} \quad T\phi_L(x,j) \leq \sum_{m=0}^{L(x,j)} T\phi_M(x,m,j) + H(x,j,L(x,j))$$

where $H(x,j,L(x,j))$ is a fixed recursive function representing the cost of initializing m, increasing m $L(x,j)$ times, and performing the "while-loop".

This gives

$$\overset{\infty}{\forall}_{x,j} \quad T\phi_L(x,j) \leq \sum_{m=0}^{L(x,j)} S_3(x,m,j,\Phi_M(x,m,j)) + H(x,j,L(x,j))$$

$$= S^*(x,j,L(x,j)) \; .$$

Next we use the existence of a recursive S_2, again monotonous in the last variable satisfying

$$\forall_{L'\in\mathcal{C}_2} \overset{\infty}{\forall}_{x,j} \quad \Phi_i(x,j) \leq S_2(x,j,T\phi_{l'}(x,j)).$$

This gives

$$\overset{\infty}{\forall}_{x,j} \quad \Phi_L(x,j) \leq S_2(x,j,S^*(x,j,L(x,j))$$

$$= S^{**}(x,j,L(x,j)) \; .$$

Finally make $R(x,z) = \max_{j\leq x} S^{**}(x,j,z)$.

Then we conclude

$$\forall_j \overset{\infty}{\forall}_x \quad \Phi_L(x,j) \leq R(x,L(x,j)) \qquad \text{q.e.d.}$$

Remark: The first part of the proof (bounding $T\phi_M$ in terms of Φ_M) is superfluous. It gives however an indication how the extra information given by an already uniform bound of the type

$$\forall_j \overset{\infty}{\forall}_{x,y} \quad \Phi_M(x,y,j) \leq K(x,y)$$

can be used to make the final function R independent of Φ_M.
In general it is not true that the complexity of an algorithm can be bounded in terms of the values of the computed functions

Proposition (1.5): For every complexity measure Φ there exists no total function $S(x,i,y)$ such that for almost all x

$$\Phi_i(x) \leq S(x,i,\phi_i(x)) \; .$$

Proof: Suppose this function S exists. Then we have for algorithms ϕ_i which compute 0-1 functions.

$$\forall_i \overset{\infty}{\forall}_x \quad \Phi_i(x) \leq \max\{S(x,i,0),\ S(x,i,1)\} \leq S(x,i,0) + S(x,i,1).$$

Let σ be a recursive function taking each value infinite often; for example take

$$\sigma(x) := x-2 \uparrow (\text{entier } ({}^2\log x)) \qquad \text{for } x > 0,\ \sigma(0) := 0 .$$

Define next

$$f(x) = \begin{cases} 1 - \phi_{\sigma(x)}(x) & \underline{\text{iff}}\ \Phi_{\sigma(x)}(x) \leq S(x,\sigma(x),0) + S(x,\sigma(x),1) \\ & \underline{\text{and}}\ \phi_{\sigma(x)}(x) = 0 \text{ or } 1 \\ 0 & \text{otherwise} \end{cases}$$

f(x) is a total recursive 0-1 function. Suppose j is an index for f. By assumption on S one has for x sufficiently large:

$$\Phi_j(x) \leq S(x,j,f(x)) \leq S(x,j,0) + S(x,j,1)$$

suppose x is also chosen to satisfy $\sigma(x) = j$.
Then one has

$$\Phi_{\sigma(x)}(x) \in \{0,1\}$$

hence $f(x) = 1 - \phi_{\sigma(x)}(x) = 1 - f(x)$

contradiction.
This proof demonstrates the technique of diagonalization in this theory.

Remark: We have already seen in (1.1) that $\Phi_i(x) = \phi_{t(i)}(x) = S(i,x)$ for suitable t and S. This does not contradict (1.5) as this S is not total.

A bounding of the size of the functions in terms of their complexity exists.

Proposition (1.6): For every measure there exists a recursive function $S(x,y)$ such that

$$\forall_i \overset{\infty}{\forall}_x \quad \phi_i(x) \leq S(x,\Phi_i(x)) .$$

Proof: Put $p(x,i,y) = \begin{cases} \phi_i(x) & \text{iff } y = \Phi_i(x) \\ 0 & \text{otherwise.} \end{cases}$

by A II $\quad p(x,i,y)$ is recursive and total

put $$S(x,y) = \max_{i \leq x} \; p(x,i,y)$$

then one has for $x \geq i$ and converging $\phi_i(x)$

$$S(x,\Phi_i(x)) \geq p(x,i,\Phi_i(x)) = \phi_i(x) .$$

If $\phi_i(x)$ does not converge there is nothing to prove.
Another useful technical lemma is the so called combining lemma which states that (under some restrictions) the complexity of a more complex computation can be bounded in terms of the complexity of its "constituent parts". For measures which are in fact counting the number of steps needed for the algorithm interpreted on some model of a machinery it is intuitively clear that such a lemma holds and one could derive it for general measures by a similar argument as in lemma (1.4). The proof given below is straightforward.

Lemma (1.7): Let $c(i,j)$ be a recursive total function such that $\phi_{c(i,j)}$ denotes an algorithm which converges on some input n provided the algorithms ϕ_i and ϕ_j converge on input n. Then there exists a recursive function $H(-,-,-)$ such that

$$\forall_{i,j}\overset{\infty}{\forall}_n \quad \Phi_{c(i,j)}(n) \leq H(n,\Phi_i(n),\Phi_j(n)).$$

Proof: put $p(n,i,j,x,y) = \begin{cases} \Phi_{c(i,j)}(n) \ \underline{\text{iff}}\ \Phi_i(n) = x \ \underline{\text{and}} \\ \qquad\qquad\qquad \Phi_j(n) = y \ . \\ 0 \qquad\quad \text{otherwise} \ . \end{cases}$

$p(n,i,j,x,y)$ is recursive and total; if the conditions $\Phi_i(n) = x$ and $\Phi_j(n) = y$ are both true then $\phi_i(n)$ and $\phi_j(n)$ both converge; hence $\phi_{c(i,j)}(n)$ and $\Phi_{c(i,j)}(n)$ are defined.

The function H is obtained by maximalization:

$$H(n,x,y) = \max_{i,j\leq n} \{p(n,i,j,x,y)\} \quad .$$

The relation $\Phi_{c(i,j)}(n) \leq H(n,\Phi_i(n),\Phi_j(n))$ follows provided $n \geq i,j$.

Applications: The combining lemma contains the condition that convergence of the combined algorithm is dependent of the convergence of the constituent parts at the same input. Thus the lemma fails in a number of interesting cases.

The lemma works in the following situations.

i) Arithmetic expressions of functions:

$\phi_{c(i,j)} = \phi_i + \phi_j,\ \phi_i - \phi_j,\ \phi_i * \phi_j,\ \phi_i \div \phi_j,\ \phi_i \uparrow \phi_j$ etc.

For fixed recursive and total $K(-,-)$, $K(\phi_i(x),\phi_j(x))$.

ii) Simulation by a universal machine:

Let $\phi_M(-,-)$ be a universal machine i.e.

$\phi_M(i,-)$ simulates $\phi_i(-)$ then one has

$$\exists_H \ \forall_i \overset{\infty}{\forall}_n \quad \Phi_M(i,n) \leq H(x,\Phi_i(x)) \ .$$

iii) Shutting off of algorithms:

$$\phi_{c(i,j)}(n) = \begin{cases} \phi_j(n) & \underline{\text{iff}}\ \Phi_j(n) \leq \phi_i(n) \\ 0 & \text{otherwise} \end{cases}$$

$$\exists_H \quad \forall_{i,j} \overset{\infty}{\forall}_n \quad \Phi_{c(i,j)}(n) \leq H(n,\Phi_i(n),\Phi_j(n)) .$$

iv) Iteration of a fixed total algorithm K:

(put $K^{(0)}(x) = x$; $K^{(n+1)}(x) = K(K^{(n)}(x))$.

$$\phi_{c(i,j)}(x) = K^{(\phi_i(x))}(\phi_j(x)) .$$

The result $\Phi_{c(i,j)}(x) \leq H(x,\Phi_i(x),\Phi_j(x))$ is not the bound one would like to have; one should want an expression bounding the complexity in the number of iteration steps.

v) In ii) we conclude that by simulation of algorithms one finds a new price $\Phi_M(i,-)$. It is easy to see that this is again a complexity measure: $\Phi_i^* = \Phi_M(i,-)$. Th. (1.3) gives another way to prove ii) without using the combining lemma. Furthermore Th. (1.3) gives a reverse bound:

$$\forall_i \overset{\infty}{\forall}_x \quad \Phi_i(x) \leq R(x,\Phi_M(i,x)) .$$

The combining lemma is not adequate to treat the following combinations of algorithms.

a) bounded sums:
$$\phi_{c(i,j)}(x) = \sum_{y=0}^{\phi_i(x)} \phi_j(y) \quad .$$

b) substitution and composition: $\phi_i(\phi_j(x))$.

c) least number operation: $\phi_{c(i)}(x) = \mu y\{\phi_i(y,x) = 0\}$.

d) primitive recursion: $\phi_{c(i,j)}(x) = \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ \phi_i(x)\ \underline{\text{else}}$
$\phi_j(x,\phi_{c(i,j)}(x-1))$.

§2. Measured sets and honest sets

We have seen that in general complexity is not bounded in terms of the size of a function. However, by restricting oneself to subclasses of the set of all algorithms one may have such bounds.

Def.: Let R be a total function in $\mathcal{R}_2$. A set $\mathcal{F}$ of functions in $\mathcal{P}_1$ is called R-honest if one has the inequality

$$\forall_{f \in \mathcal{F}} \exists_i [\phi_i = f \text{ and } \overset{\infty}{\forall}_x \ \Phi_i(x) \leq R(x, f(x))] \ .$$

If one has this inequality for all x and not just for almost all x one the functions of $\mathcal{F}$ are called really R-honest.

The union of two R-honest (really R-honest) sets is again R-honest (really R-honest). It is clear that there exists a maximal R-honest set (the set of all R-honest functions) (similar for really R-honest).

A set of functions is called honest (resp. really-honest) if it is R-honest (really R-honest) for some R $\in \mathcal{R}_2$.

Def.: A sequence of functions $\{f_i\}_{i \in \mathbb{N}}$ is called measured provided that the predicate

$$Q(j,x,y) \equiv f_j(x) = y$$

is recursive.

Example: The set of timebounds $\{\Phi_i\}_{i \in \mathbb{N}}$ is measured. (This is in fact Axiom II).
Each recursive enumerable set of total functions is measured (Just wait for the computation to stop and look at the result.)

In [1] the following equivalence is claimed:

(6.7) fact: The set of functions which are G-honest (with respect to Φ) form a measured set for any G $\in \mathcal{R}_2$ and any Blum measure Φ.

Conversely, given any measured set and Blum measure Φ there exists a $G \in R_2$ such that the set of G-honest functions with respect to Φ contains this measured set.

We show that the first assertion is false. It can be proved if one replaces "G-honest" by "really G-honest". The second assertion is true and we shall prove it also.

<u>Example</u> (2.1): Let $k(i,y)$ be a total function in R_2 such that $\phi_{k(i,y)}$ denotes the algorithm:

$$\phi_{k(i,y)}(x) := \underline{\text{if}}\ x > y\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ \phi_i(x).$$

It is clear that the set of indices $k(i,y)$, $i,y \in \mathbb{N}$ is a recursive set as the enumeration is effective. Also given an index z in this set one can compute the i and y for which $z = k(i,y)$.
Let Φ be an arbitrary measure. Define a new measure Φ^* by:

$$\Phi^*_j(x) = \begin{cases} \Phi_j(x) + x & \underline{\text{iff}}\ j \neq k(i,y) \text{ for all } i \text{ and } y \\ \Phi_i(x) & \underline{\text{iff}}\ j = k(i,y) \text{ and } x \leq y \\ 0 & \underline{\text{iff}}\ j = k(i,y) \text{ and } x > y\ . \end{cases}$$

Axiom I is satisfied: if $j \neq k(i,y)$ then $\Phi^*_j(x)$ converges <u>iff</u> $\Phi_j(x)$ converges; if $j = k(i,y)$ then $\Phi^*_j(x)$ converges if $\Phi_i(x)$ converges or if $x > y$ but in both cases $\phi_j(x)$ also converges.

Also Axiom II is satisfied. To test whether $\Phi^*_j(x) = z$ first see whether $j = k(i,y)$ for some i and y (which is possible as the set $\{k(i,y) \mid i,y \in \mathbb{N}\}$ is recursive).
If not so perform the decision procedure for $\Phi_j(x) = z-x$.
If so compute i and y. Test whether $x \leq y$.
If so perform the old decision procedure for $\Phi_i(x) = z$.
If not test whether $z = 0$.

In this case one has

$$\overset{\infty}{\forall}_x \quad \Phi^*_j(x) \leq 0 \quad \underline{\text{iff}} \quad j = k(i,y)$$

so only functions which are zero almost everywhere are 0-honest. Conversely, each such function is computed by some algorithm $\phi_{k(i,y)}$ for some i and some y. So the 0-honese functions are exactly those functions which are almost everywhere zero. (functions of finite support).

Now suppose the set of functions of finite support is measured. This gives that the collection of all functions is measured as we can decide

$$\phi_i(x) = y \quad \text{by testing}$$
$$\phi_{k(i,x)}(x) = y \quad \text{which is decidable.}$$

This would give a solution to the halting problem for 0-1 functions which is impossible.
This example proves that the first assertion in [1] fact (6.7) is false.
If we replace R-honest by really-R-honest the assertion becomes:

Proposition (2.2): For every measure Φ and every $G \in \mathcal{R}_2$ the set of really R-honest functions is measured.

Proof: f is really R-honest iff there exist an index i for f such that $\forall_x$: $\Phi_i(x) \leq R(x,f(x))$. (This index is known at the moment one concludes that f is really R-honest).

In order to test whether $f(x) = y$ first test whether $\Phi_i(x) \leq R(x,y)$. If this is not the case, the outcome of the test is negative; otherwise f(x) converges and the test reduces to straightforward computation and comparison.

The second assertion becomes:

<u>Proposition</u> (2.3): Let $\{\psi_i\}_{i \in \mathbb{N}}$ be a measured set. Then the set $\{\psi_i\}$ is R-honest for some $R \in \mathcal{R}_2$.

<u>Proof</u>: Let ϕ_Q be an algorithm which computes the function:

$$\phi_Q(i,x,y) = \begin{matrix} 0 & \text{iff } \psi_i(x) = y \\ 1 & \text{otherwise .} \end{matrix}$$

Then one has in the measure Φ

$$\phi_Q(i,x,y) \text{ is recursive and total}$$

and

$$\psi_i(x) = \mu y\{\phi_Q(i,x,y)=0\} = \phi_L(i,x).$$

By application of Lemma (1.4) after replacing x by the pair i,x (and forgetting about the parameter j) one has

$$\overset{\infty}{\forall}_{i,x} \quad \Phi_L(i,x) \leq H(i,x,\phi_L(i,x)) = H(i,x,\psi_i(x))$$

putting $R(x,z) = \max\limits_{i \leq x} H(i,x,z)$ one obtaines the result

$$\forall_i \overset{\infty}{\forall}_x \quad \Phi_L(i,x) \leq R(x,\psi_i(x)) \text{ i.e. } \psi_i(x) \text{ is computed at most } R(x,\psi_i(x)) \text{ steps.}$$

The proof gives at the same time also the algorithm for computing $\psi_i(x)$ within this time bound. By the S-n-m theorem there exist a recursive function σ such that

$$\phi_{\sigma(i)}(x) = \phi_L(i,x) = \psi_i(x).$$

Further we did remark before in section I, Lemma 1.7, application v) that

$$\forall_i \overset{\infty}{\forall}_x \quad \Phi_{\sigma(i)}(x) \leq R'(x,\Phi_L(i,x)) \leq R'(x,R(x,\psi_i(x))) = R''(x,\psi_i(x)) \ .$$

The notion of honest y is not completely measureindependent. It is in general not true that the collection of all R-honest functions with respect to Φ is again the complete set of all R'-honest functions with respect to Φ' (for some R'). However one has the following weaker assertion

Proposition (2.4): If F is an R-honest set with respect to a measure Φ then there exists for every Φ' an R' such that F is R'-honest with respect to Φ'.

Proof: Suppose $f \in F$ with $f = \phi_i$ and $\overset{\infty}{\forall}_x \ \Phi_i(x) \leq R(x,f(x))$.
Choose $S \in R_2$ such that

$$\forall_i \overset{\infty}{\forall}_x \quad \Phi'_i(x) \leq S(x,\Phi_i(x)) \ .$$

One may assume that S is monotonous in the second variable (otherwise put $S'(x,y) = \max\limits_{y' \leq y} S(x,y)$).
Then one has

$$\overset{\infty}{\forall}_x \quad \Phi'_i(x) \leq S(x,\Phi_i(x)) \leq S(x,R(x,f(x))) := R'(x,f(x)).$$

It is clear that R' is independent of i and f.

§3 a. The naming theorem

In this section we need the notion of a complexity class which is defined below. We assume that Φ is some fixed complexity measure.

Definition: Let $t \in P_1$ be a recursive function. Then the complexity class C_t in the measure Φ (where necessary denoted by C_t^Φ) is the set of all functions possessing an algorithm which runs for almost all inputs x within t(x) steps.

In formulas:

$$F_t = \{i \mid \Phi_i(x) \leq t(x) \text{ for almost every } x\}$$

$$C_t = \{f \mid \exists_i \quad f = \phi_i \text{ and } i \in F_t\} \ .$$

To understand the significance of the naming theorem consider the following two theorems: (see also [1] and [2]).

Th. 3.1 Weak compression theorem [Blum]: Let $\{\gamma_i\}$ be a measured set of functions. Then there exists a recursive function $R \in R_2$ such that

$$C_{\gamma_i} \underset{\neq}{\subset} C_{R(x,\gamma_i(x))}$$

provided γ_i is total.

Th. 3.2 Gap theorem [Borodin]: In every measure there exists for every total function $R \in R_2$ $R(x,y) \geq y$ a total function $t \in R_1$ such that

$$C_t = C_{R(x,t(x))} \ .$$

Theorem 3.1 shows that it is possible for every measured set of runtimes to enlarge the complexity classes uniformly by increasing the running times in an effective way (as long as the runtimes are total). There exist however for any measure and any effective increasing function R always runtimes t which have the peculiar property that there are no algorithms running almost everywhere within time R(x,t(x)) which are not running already almost everywhere within time t.

The naming theorem shows that the Gap result is induced by the possibility of choosing the wrong names t for complexity classes.

Th. 3.3 <u>Naming theorem</u> [McCreight-Meyer]: There exists a measured set $\{\gamma_i\}$ such that every complexity class C_t is equal to a class C_{γ_i}.

First we describe proofs for Th. 3.1 and Th. 3.2. The proof of Th. 3.3 uses a complicated priority-list argument. This proof is sketched in [1] and in [2]. The proof given here differs from these other proofs only at one essential point. After the proof we indicate and discuss this point and show by an example that the correction is necessary (see section 4).

<u>Proof of th. 3.1</u>: Let $\{\gamma_i\}$ be a measured set. Let $\sigma(x) = x - 2 \uparrow [{}^2\log x]$ if $x \geq 0$: $\sigma(0) = 0$; we define a function h_i by

$$h_i(x) = \begin{cases} \phi_{\sigma(x)}(x) + 1 & \underline{\text{iff}} \quad \Phi_{\sigma(x)}(x) \leq \gamma_i(x) \\ \\ 0 & \text{otherwise.} \end{cases}$$

The function h_i should be computed as follows: First try to compute $\gamma_i(x)$. If this fails $h_i(x)$ is undefined otherwise compute $\sigma(x)$ and test whether $\Phi_{\sigma(x)}(x) \leq \gamma_i(x)$. If so then $\phi_{\sigma(x)}(x)$ converges and we have $h_i(x) = \phi_{\sigma(x)}(x) + 1$ otherwise $h_i(x) = 0$.

From this description it follows that $h_i(x)$ converges iff $\gamma_i(x)$ converges. Now $\{\gamma_i\}$ is a measured set so by prop. 2.4 one has a recursive function s such that

i) $\phi_{s(i)} = \gamma_i$ and

ii) $\Phi_{s(i)}(x) \leq R^*(x,\gamma_i(x)) \qquad \forall_i \overset{\infty}{\forall}_x$.

By the combining lemma we have

$$\forall_i \overset{\infty}{\forall}_x \quad \Phi_{h_i}(x) \le R^{**}(x, \Phi_{s(i)}(x)) \le R(x, \gamma_i(x))$$

(where we can choose R such that $R(x,y) \ge y$)

hence $h_i \in C_{R(x,\gamma_i(x))}$.

Now suppose $h_i = \phi_j$ with $j \in F_{\gamma_i}$ and γ_i total.
Then for x large enough we have

$$\Phi_j(x) \le \gamma_i(x)$$

so if $\sigma(x) = j$ we have

$$\Phi_{\sigma(x)} \le \gamma_i(x) \qquad \text{which implies}$$

$$h_i(x) = \phi_j(x) + 1 = h_i(x) + 1 \quad \underline{\text{contradiction}}$$

this completes the proof.

<u>Remark</u>: The diagonalisation argument fails when γ_i is not total.

The next trivial example shows that the relation $C_{\gamma_i} \underset{\neq}{\subset} C_{R(x,\gamma_i(x))}$ may fail for non total γ_i:
Let the measured set contain an algorithm which converges never. So $\gamma(x)$ = undefined and $\gamma(x) = y$ is always false. Now $C_{\gamma_i} = \mathcal{P}_1$ and $C_{R(x,\gamma_i(x))} = \mathcal{P}_1$ as well.

<u>Proof of th. (3.2)</u>: Let $R \in \mathcal{R}_2$ be total, $R(x,y) \ge y$.
We define a function t by

$$t(k) = \mu z \, \{\forall_{i \le k} \ \underline{\text{either}} \ \Phi_i(k) \le z \ \underline{\text{or}} \ \Phi_i(k) \ge R(k,z)\}.$$

The predicate used to define t is recursive hence t is recursive also. To show that t is total we must show that for each k a z with this property exists.

Define $z_0 = 0$ and

$$z_{n+1} = \max_{i \leq k} \{\Phi_i(k) \mid \Phi_i(k) \leq \max_{z \leq z_n} \{R(k,z)\}\} .$$

From this definition we conclude $z_{n+1} \geq z_n$. Next put

$$I_n = \{i \mid i \leq k \wedge \Phi_i(k) \leq z_n\} .$$

We conclude $I_{n+1} \supset I_n$. As $I_n \subset [0,k]$ there exists an $m(\leq k)$ such that $I_{m+1} = I_m$.

Now $i \in I_m \Longleftrightarrow i \in I_{m+1}$ $\forall_{i \leq k}$ hence

$$\forall_{i \leq k} \quad \Phi_i(k) \leq z_m \Longleftrightarrow \Phi_i(k) \leq \max_{j \leq k}\{\Phi_j(k) \mid \Phi_j(k) \leq \max_{z \leq z_m} \{R(k,z)\}\}$$

Therefore we have

$$\forall_{i \leq k} \quad \Phi_i(k) \leq z_m \ \underline{\text{or}} \ \Phi_i(k) > \max_{j \leq k} \Phi_j(k) \mid \Phi_j(k) \leq \max_{z \leq z_m} \{R(k,z)\}\}$$

consequently

$$\forall_{i \leq k} \quad \Phi_i(k) \leq z_m \ \underline{\text{or}} \ \Phi_i(k) > R(k,z_m) .$$

So z_m is a z with the property we need. This proves t to be total.

The inclusion $C_{t(x)} \subset C_{R(x,t(x))}$ is trivial as $R(x,y) > y$. To see the reverse inclusion suppose that for some i we have $i \notin F_t$, i.e.:

$$\overset{\infty}{\exists}_x \quad \Phi_i(x) > t(x)$$

for $x \geq i$ one has by definition

<u>either</u> $\Phi_i(x) \leq t(x)$ <u>or</u> $\Phi_i(x) > R(x,t(x))$

<u>so</u> $\overset{\infty}{\exists}_x \; \Phi_i(x) > t(x)$ implies $\overset{\infty}{\exists}_x \; \Phi_i(x) > R(x,t(x))$

and therefore $i \notin F_{R(x,t(x))}$

<u>Remark</u>: The timebound t constructed in the proof can be chosen larger than any total recursive function h.
Just define t by

$$t(k) = h(k) + 1 + \mu z\{\forall_{i \leq k} \; \underline{\text{either}} \; \Phi_i(k) \leq h(k) + 1 + z \; \underline{\text{or}} \; \Phi_i(k) > R(k,h(k)+1+z)\}$$

the proof continues as before.

By a similar trick one can construct monotonous functions t having this property.

§3 b. Discussion of the algorithm and proof of the naming theorem

The naming theorem is proved by constructing an algorithm computing for each timebound t a new timebound t' such that $C_t = C_{t'}$. We must have the relation:

$$\overset{\infty}{\exists}_x \quad \Phi_i(x) > t(x) \Longleftrightarrow \overset{\infty}{\exists}_x \quad \Phi_i(x) > t'(x) \ .$$

The set of timebounds t' is to be a measured set. We construct the algorithm in such a way that during the execution of the algorithm the least possible result is an increasing function of the time. So we can have the computation going on for so much time that we can prove it will never more produce the result t'(x) = y. This alone would not suffice as we do not have an indication when this time has come, this indication however is present in the outcome of a failing computational try for t'(x).

Except for a number of subroutines which are left undeclared the algorithm can be described by an ALGOL 60 program.

In this section we first discuss the essential parts; after this we give the complete program.
The program uses the ALGOL 60 feature of an own array. This way it is possible to extend arrays by leaving and reentering a block without losing the information stored in the array. (This feature is forbidden by almost all ALGOL 60 systems *).

The algorithm is devided in stages (stage 1, stage 2,..., stage x,...) each of which is divided in two parts. Each stage is completed after finitely many steps. One can prove that the number of steps of stage x is bounded by a function F(x) (and hence independent of t!).

*) There exist recursive techniques by which this problem is solved. We do not treat these as they are irrelevant to the precent problem.

We describe the essential tasks of the algorithm at stage x, Together with a number of specifications of the subroutines used this description is sufficient to prove that the algorithm does the job it is designed for.

Stage x: PART 1:
Algorithm x is introduced. It is supposed to satisfy $\Phi_x \leq t$. It is assigned a priority number which is (as is each newly assigned priority number) the highest priority (lowest priority number) not assigned before. (So no two priorities will be equal).

By means of an universal machine (procedure UNIVERSAL) a "dovetailed" computation of the function t is performed on the arguments 1,...,x.

If no new value of t is computed we proceed at once to PART 2; otherwise we test for all the new values of t which have been computed say $t(z_1),\dots,t(z_r)$ whether the algorithms $\phi_1 \dots \phi_x$ violate the timebound t at the arguments $z_1,\dots,z_r$; i.e. one tests whether $\Phi_i(z_j) \leq t(z_j)$ $1 \leq i \leq x$, $1 \leq j \leq r$.

If a function ϕ_i is caught violating the timebound it is put on "the black list" if it was not already there. In the algorithm this is done by assigning a new priority number to which a minus sign is attached.

When this test is completed we proceed to PART II.

PART 2:
First make y equal to $\sigma(x)$; σ being a total function taking each positive value infinitely often and never becoming zero, $\sigma(x) < x$ for $x \geq 2$. y is the argument for which we try to compute the new timebound t'. Hence we proceed straightforwardly to the next stage if t'(y) is found to have been computed already.

Next we reconstruct (or retrieve!) the priorities and the black list set up as computed after having completed stage y. This is done as we want to consider each time we try a computation of t'(y) the same problem.

To compute t'(y) we search for a value z for t'(y) which makes it possible to delete a function, say ϕ_j from the black list by having

$$\Phi_j(y) > z = t'(y) .$$

However we have at the same time to respect the requests of functions not on the black list which have a higher priority; i.e.:
for ϕ_i not on the black list such that ϕ_i has a higher priority we must have

$$\Phi_i(y) \leq z = t'(y).$$

If we succeed in finding such a value z and a corresponding index j we give ϕ_j a new priority (at <u>stage</u> x!) removing ϕ_j from the black list by removing the minus sign and giving it a new priority at <u>stage</u> x! unless ϕ_j is found not to be on the black list at <u>stage</u> x.
As it is not at all clear that we can find a solution to this problem the algorithm searching for z and j has a built-in clock which breaks off the algorithm up on the exceeding of a certain timebound. Again the notion of a timebound is introduced (and hence we need some complexity measure).

The timebound we use depends on whether t(y) has already been computed at stage x or not. If t(y) has not yet been computed we take the timebound equal to x; if failure occurs we proceed to the next stage. If however t(y) has been computed we give the algorithm an amount of "time" equal to:

$$\max\{x, G(y,t(y))\} \quad ^{(*)}$$

where G is some recursive function large enough to permit the algorithm for t'(y) to test all possible values z upto t(y) + 1. If failure occurs now we give for t'(y) a definition ad hoc which is in fact the last value z tried in the computation (hence $t'(y) \geq t(y)$ in this situation). Without altering on the priorities and the black list we proceed to the next stage.

(*) In the proof of [1] and [2] the bound $\max\{t(y),\Phi_t(y)\}$ is used. In our proof we shall have G(y,t) = t+1! However this trivial G is made possible by choosing a special clock to measure the searching algorithm. If we should use the abstract complexity measure Φ G must be in general a complicated function. This interpretation of the timebound is the essential difference between our proofs and those in [1] and [2].

From the construction it follows that the domain of t' contains the domain of t, the reverse inclusion not necessarily being true.

The reader should note that in PART 2 the central search for a value of t'(y) is more or less independent of t as long as t(y) is not yet computed; only the set up of the priorities and the black list are influenced by t. In our construction there are only $\leq ((y^2!) * (2\uparrow y)$ possible priority - and black list - set ups at the end of <u>stage</u> y (and of these only (y!) * 2↑y are essentially different).

Also the set up of the searching algorithm is irrelevant for the construction of PART 1.

It has to be remarked that the timebound t is introduced in the algorithm by introducing a parameter i in UNIVERSAL which is an index for t. So t' is dependent on the program we use to compute t. This is the best we can hope for. An effective transformation $t \longmapsto t'$ which depends only on t as a function is forbidden by Constables Operator Gap theorem [4].

Complexity measures are introduced at three places in the algorithm

i) In the "dovetailed" computation of t(1)...t(x).
(Performed by a procedure UNIVERSAL (i,x,BOUND) where i is an index for t and BOUND is a limit on the number of steps alotted to the computation).

ii) In the testing of the predicate $\Phi_i(x) \leq t(x)$. This is performed by a procedure MEASURE(i,x,y). The measure considered here is the abstract measure presupposed in the theorem.

iii) In the timebound put upon the calculation of t'(y), in PART 2. This measure could be defined to be the number of "elementary" ALGOL statements executed (see APPENDIX). This part of the algorithm is performed by the procedure SEARCH TIME(list,y,index, result,bound).

The reader should note that the three measures playing a role in i), ii) and iii) <u>all may be different</u>. Also the exact type of machinery which is simulated in UNIVERSAL or MEASURE is irrelevant for the Algorithm.

We even do not need a complete measure in SEARCH TIME. Any clock preventing infinite but allowing arbitrarily large computations for this specific algorithm will do.
Given this global description we indicate why this algorithm does the job. We also indicate a number of specifications of parts of the program which we need in order to prove that our algorithm works.

The naming theorem can be proved if we can prove the following three claims:

<u>CLAIM 1</u>: If for a given i and t there exists an infinite sequence of arguments $x_j (x_{j+1} > x_j)$ for which $\Phi_i(x_j) > t(x_j)$ then also for the new timebound t' there exists a sequence $y_j (y_{j+1} > y_j)$ for which $\Phi_i(y_j) > t'(y_j)$.

<u>CLAIM 2</u>: If for a certain function ϕ_i and a certain t the number of violations of ϕ_i against t is finite (so $\overset{\infty}{\forall}_x \; \Phi_i(x) \leq t(x)$), then the number of violations of ϕ_i against t' is finite also.

<u>CLAIM 3</u>: For every timebound t and for every x and y we can test whether the new timebound t' satisfies $t'(x) = y$.

CLAIM 1, 2 and 3 together prove the naming theorem.

From CLAIM 1 we derive $C_{t'} \subset C_t$ for

$$i \notin F_t \Longrightarrow \overset{\infty}{\exists}_x \quad \Phi_i(x) > t(x) \underset{\text{CLAIM 1}}{\overset{}{\Longrightarrow}} \overset{\infty}{\exists}_x \quad \Phi_i(x) > t'(x) \Longrightarrow i \notin F_{t'}$$

CLAIM 2 shows the reverse inclusion $C_t \subset C_{t'}$ as

$$i \in F_t \Longrightarrow \overset{\infty}{\forall}_x \quad \Phi_i(x) \leq t(x) \Longrightarrow \overset{\infty}{\forall}_x \quad \Phi_i(x) \leq t'(x) \Longrightarrow i \in F_{t'}$$

CLAIM 3 shows the sequence {t'} to be a measured set.

To prove these claims we need more details of the algorithm. Ultimately the correctness of our claims must be derived from the program text given in Section III c and d. We can decompose this task in smaller parts; in doing this we formulate during the proof a number of specifications which the program has to satisfy.

Claim 1: can be divided into two parts.

Let ϕ_i be an algorithm violating infinitely often the timebound t. i.e. $\overset{\infty}{\exists}_x \quad \Phi_i(x) > t(x)$. Then the following assertions hold:

Claim 1a: There exists for every $z \in N$ a stage number $z' > z$ such that ϕ_i is on the black list at the end of the execution of stage z'; PART 1.

Claim 1b: If the function ϕ_i is on the black list at the end of execution of PART 1 of stage z', then it is still on the black list at the end of stage z' unless it has been removed from the black list.

If the function is on the black list at the end of stage z' it will be removed during execution of PART II at some future stage z''

We need a few specifications.

Specification A: If some function ϕ_j is removed from the black list this is performed during execution of PART 2 of some stage z. This removal has the following side effects:

a): The value of $t'(\sigma(z))$ is computed; the computed value satisfies $\Phi_j(\sigma(z)) > t'(\sigma(z))$.

b): At any future stage $z' > z$ with $\sigma(z') = \sigma(z)$ the execution of PART II is suppressed. (This happens also if $t'(\sigma(z))$ is computed without removing some function from the black list).

The consequence of b) is that given an infinite sequence $\{z_i\}$ of stage numbers where a function is removed from the black list (or more generally a value of t' is defined), the values of $\sigma(z_i)$ are all distinct; consequently $\lim_{i\to\infty} \sigma(z_i) = \infty$.

Specification B: The priorities of the functions are registered within an array prior. The priority number of function ϕ_i is given by the absolute value of prior[i] and the sign of prior[i] denotes whether the function is on the black list or not. Each time the value of prior[i] is changed the new value will have the opposite sign of the old value. The absolute value of prior[i] will be a number larger then all the absolute values assigned to an element of the array prior before.

After the completion of the execution of PART 2 of stage x the contents of the array prior are copied into a new section of the array old prior for future use (in PART 2 of stage $z > x$ for which $\sigma(z) = x$).

Claim 1 is easily derived from claim 1a, b and specifications A and B. The specifications show that changes in the priority set up are made only when a function not on the black list is moved to the black list or when a function on the black list is removed from it.

Now let ϕ_i be a function which violates timebound t infinitely often. Let z be a stage number. Claim 1a shows that there exists a stagenumber z' such that ϕ_i is on the black list after execution of stage z', PART 1. $z' > z$. Claim 1b shows that the function ϕ_i will be of the black list again at the end of execution of PART 2 during some stage $z'' > z'$.

By specification A this results in the creation of a violation of ϕ_i against the timebound t' which will be distinct from all violations created earlier in the algorithm.

This argument can be repeated starting with stage z''. By complete induction one can find a sequence of distinct violations of ϕ_i against t'. Hence $\overset{\infty}{\exists}_x \quad \Phi_i(x) > t'(x)$.

Claim 2 can also be divided in two parts.

Claim 2a. A function ϕ_i which violates the timebound t only finitely many times has its priority and presence on or off the black list changed only a finite number of times.

Claim 2b. If the priority number and presence on or off the black list of the function ϕ_i is not changed after stage z then the number of violations of ϕ_i against t' created on arguments $> z$ is finite.

To prove claim 2 from claim 2a and claim 2b is not difficult. Suppose $\overset{\infty}{\forall}_x \quad \Phi_i(x) \leq t(x)$.

By claim 2a there exists only a finite number of stage numbers $z_1,\dots,z_R$ such that the value of prior[i] is changed during stage z_j. Let $z_0 := \max\{z_1,\dots,z_R\}$. At stage z_0 only a finite number of values of t' is computed. By claim 2b the values of t' computed at arguments exceding z_0 are violated by ϕ_i only a finite number of times. Hence $\overset{\infty}{\forall}_x \quad \Phi_i(x) \leq t'(x)$.

To show the validity of our sub claims made above and claim 3 we need more details of subroutines in the algorithm. The subroutines we explain are called UNIVERSAL, MEASURE and SEARCH TIME.

Specification C: The "dovetailed" computation of the timebound t in PART 1 is performed by the integer procedure UNIVERSAL(i,x,bound). i is an index for the timebound t (in the effective enumeration of the algorithms used in the set up of UNIVERSAL which may be distinct from the one we are considering in relation to the complexity measure Φ). x is the argument for which we want to compute t(x) and bound is the number of steps allotted to the computation with respect to some complexity measure Φ (which again may be distinct from Φ). The calculation of UNIVERSAL terminates on each input with the result

UNIVERSAL := if $\Psi_i(x) \leq$ bound then $\psi_i(x)$ else -1.

Specification D: The test for the occurrence of violations in PART 1 is performed by the boolean procedure MEASURE(i,x,y). This procedure computes the value of the predicate $\Phi_i(x) \leq y$. The computation of MEASURE is also used in SEARCH TIME.

Specification E: In stage x PART 1 we test for each argument $y \leq x$ whether t(y) is already computed or not. If not, we compute UNIVERSAL(i,y,x), i being an index for t. If the result is positive (UNIVERSAL $\neq -1$) then we compute for $j = 1,\dots,x$ MEASURE(j,y,t(y)). Violations of t which are found (MEASURE(j,y,t(y)) is false) result in the moving of ϕ_j to the black list, unless ϕ_j is found to be on the black list already. No other process will place a function on the black list.

Global description of PART 1:

```
for y := 1 step 1 until x do
            if t(y) not yet computed then
      begin t(y) := UNIVERSAL(i,y,x);
            if t(y) $\neq$ -1 then
               for j := 1 step 1 until x do
            if ¬MEASURE(j,y,t(y)) and prior[j] > 0 then
               put $\phi_j$ on the black list
      end;
```

Specification F: The search for a value for t'(y) is performed by the procedure SEARCHTIME(y,oldprior,index,result,bound).
The parameters have the following meaning.
y is the argument for which t' has to be computed. oldprior contains the contents of prior[1:y] after the completion of stage y.
index is a variable in which the index of a function to be removed from the black list is stored if the calculation succeeds; otherwise index is made -1 result is a variable in which the value of t'(y) is stored if the calculation succeeds. Otherwise result is made equal to the value tried for t'(y) in the computation at the moment failure occurred.
bound is the number of steps allotted to the computation of SEARCHTIME.

The procedure has a local array functions[1:y] and a pair of local variables candidate and value.

SEARCHTIME first orders the functions $\phi_1,\ldots,\phi_y$ in order of their priorities as registered in old prior. abs(functions[j]) is the index of a function ϕ_i i = 1,...,y and abs(prior[abs(functions[j])]) < abs(prior[abs(functions[j+1])]).

Next, candidate is made 1 and value is made 0. During execution of SEARCHTIME, these variables will be increased by one each time their value is changed. When SEARCHTIME halts by succes or failure the last content of value is stored in result.

Failure occurs if value tries to exceed bound; also if candidate exceeds y value is made to increase up to bound and failure occurs. Then the value of result at exit will be bound.

If the computation succeeds the following assertions hold.

a) if the value of index is j then ϕ_j was on the black list at the end of <u>stage</u> y. Further if w is the value of result we have $\Phi_j(y) > w$.

b) for each function ϕ_k, $1 \leq k \leq y$, which was not on the black list at the end of <u>stage</u> y and which had a higher priority than ϕ_j we have $\Phi_k(y) \leq w$ equality being true at least once.

c) j has the highest priority of those functions on the black list for which the conditions laid down in a) and b) are satisfiable, for some value of w.

In general we can not say much about the outcome of SEARCHTIME. We know however the following:

<u>Specification G</u>: Let ϕ_j be a function on the black list at the end of <u>stage</u> y. Let z be a stage number $z > y$ such that $\sigma(z) = y$ and such that t(y) is computed at stage z. Suppose $\Phi_j(y) > t(y)$.
Let $T = \max\{t(y)+1,z\}$

(This timebound T is the time allotted to SEARCHTIME by our algorithm in the situation t(y) computed, t'(y) still undefined.)

Now there are two possibilities:

I) the computation of SEARCHTIME(y,oldprior,index,result,T) succeeds to deliver the index i of a function which can be removed from the black list, having a priority higher or equal (in which case i = j) than the priority of j.

II) The computation of SEARCHTIME is frustrated because one of the functions ϕ_i having a higher priority then ϕ_j and not on the black list violates the timebound t at y: $\Phi_i(y) > t(y)$.

If the latter occurs we know however that this function is moved to the black list during the stage $z' \leq z$ at which t(y) was computed.

Specification H: A value of $t'(y)$ is defined either by a positive result of SEARCHTIME at a moment $t'(y)$ was not yet defined or at the moment failure in SEARCHTIME occurs during PART 2 of stage z' where z' is the least stage number such that $t(y)$ is defined at stage z' and $\sigma(z') = y$. The definition of $t'(y)$ is in this latter case not accompanied by a change in the priority and black list set up. The chosen value of $t'(y)$ is not smaller than $t(y)$, and also greater or equal than any number stored in result during a previous run of SEARCHTIME at the argument y.

Specification I: If a call SEARCHTIME(y,oldprior,index,result,bound) succeeds and if bound' > bound then the call SEARCHTIME(y,oldprior, index,result,bound') would succeed as well delivering identical values in index and result. If the former call does not deliver a positive result and the latter call does give a positive result the value stored in result by the latter call exceeds bound.

Proof of claim 1a: Let ϕ_j be a function such that

$$\overset{\infty}{\exists}_x \quad \Phi_j(x) > t(x). \text{ Let } t = \psi_k.$$

There exists a sequence (not necessarily recursive but surely recursively enumerable) of distinct arguments z_i for which $t(z_i)$ converges (otherwise $\Phi_j(z_i) \leq t(z_i)$) and for which $\Phi_j(z_i) > t(z_i)$. The convergence of $t(z_i)$ means that at some stage $v_i \geq z_i$ $t(z_i)$ is newly computed. ($v_i = \max\{z_i, \Psi_k(z_i)\}$.)

Now let z be an arbitrary stage number. As all z_i are distinct there exist an i for which $z_i > z$. Now also $v_i > z$. At stage v_i the call UNIVERSAL(k,z_i,v_i) computes $t(v_i)$. This value was not computed before. Next we compute MEASURE($j,z_i,t(z_i)$) and the violation is discovered. Hence at the end of execution of stage v_i, PART 1 ϕ_j is on the black list.

Proof of claim 1b: The first assertion being completely trivial we turn to the second one. By assumption there exists an infinite sequence of violations of ϕ_j against t at arguments z_i.

Now let ϕ_j be on the black list after execution of stage z. Let $z_i > z$ be an argument for which $\Phi_j(z_i) > t(z_i)$. Let v_i be the stage number at which $t(z_i)$ is first computed, and let w_i be the least stage number $\geq v_i$ for which $\sigma(w_i) = z_i$.

We claim that during the execution of the algorithm between stage z and stage w_i at least one function having a priority higher or equal than the priority of ϕ_j at stage z is moved.

By this move this higher priority is deleted - the newly assigned priority will be lower than all priorities assigned before.

This claim proves claim 1b. If ϕ_j is not removed before or at stage w_i a higher priority will have vanished the priority of ϕ_j being unchanged. Repeating our argument with stage w_i instead of stage z we find a new stage w_{ii} before or at which ϕ_j or a function with higher priority moves. This argument can be repeated by complete induction. As there exists only a finite number of higher priorities ϕ_j must move at some future stage and this means that ϕ_j leaves the black list (Specification B).

To prove our claim we discuss the different possible developments of the algorithm.

i) If ϕ_j is not on the black list at the end of stage z_i then ϕ_j is removed from the black list before or at stage z_i hence before or at stage w_i and there remains nothing to prove.

ii) If one of the functions ϕ_i which have at stage z_i a higher priority than ϕ_j and which are not on the black list violates t it is put on the black list at stage v_i which is before or at stage w_i and again we are done.

iii) Hence we may assume that ϕ_j is on the black list at stage z_i and that functions ϕ_i having a higher priority and not on the black list do not violate t. As furthermore $\Phi_j(z_i) > t(z_i)$ the function ϕ_j satisfies the two conditions a) and b) in Specification F.

iv) Next take $T = \max\{w_i, t(z_i)+1\}$. By Specification G we know that the call SEARCHTIME(z_i,oldprior,index,result,T) delivers a positive result, the frustrating case being prohibited by our assumption iii) above.

v) Consider the computation of stage w_i, PART 2. If $t'(z_i)$ is already computed this is the result of a positive result of SEARCHTIME(z_i,oldprior,index,result,bound) at stage w with $z_i < w < w_i$ and $\sigma(w) = z_i$.
At this stage w we use bound = w as $t(z_i)$ is not yet computed (see the global description). Now the results of SEARCHTIME(z_i, old prior,index,result,w) and SEARCHTIME(z_i,oldprior,index,result,T) are equal by Specification I. This last result is computed in the case $t'(z_i)$ was not yet computed at stage w_i.

vi) In both cases ($t'(z_i)$ computed before stage w_i or at stage w_i) Specification G applies. Some algorithm ϕ_i with higher priority then ϕ_j at stage z_i can be removed from the black list. As the priority of ϕ_i is higher than the priority of ϕ_j it must have been assigned before or at stage z. It will be removed at stage w(resp w_i) unless it is already removed in between stage z_i and stage w(resp w_i). So the move occurs after stage z and before or at stage w_i.

This proves our claim and hence claim 1b is proved.

Proof of Claim 2a: The only places where moves of functions on or off the black list and changes of priority occur are in the test part of PART 1 (see Specification E) or in PART 2 after a positive result of SEARCHTIME(Specification A and H).
By Specification A each change of priority number is accompanied by a move to or from the black list. A necessary condition for a move of ϕ_j to the black list is the discovery of a violation of ϕ_j against t. (Specification E).

Hence a function which violates t only a finite number of times can perform only a finite number of moves to the black list. After the last move to the black list it may be removed or not. But no more than one move will occur afterwards. Hence the number of moves is finite.

Proof of Claim 2b: Suppose the function ϕ_j has reached its ultimate place and priority at stage z_0. We show that for a violation of ϕ_j against t' created at an argumenty $\geq z_0$ a priority higher than the ultimate priority of ϕ_j is deleted somewhere between stage z_0 and w. Suppose $y > z_0$ and $\Phi_j(y) > t'(y)$. This means that t'(y) converges . If a value is assigned to t'(y) this may have occurred by the ad hoc definition at a stage w where t(y) was computed. Now $t'(y) \geq t(y)$. However $\Phi_j(y) \leq t(y)$; otherwise a violation of ϕ_j against t would have been discovered before or at stage w but after stage z_0 quod non. Then we have a contradiction as we assumed $\Phi_j(y) > t'(y)$.

Hence we may assume that t'(y) is defined by a positive result in SEARCHTIME. Let the value of index be i. Again there is a number of possible cases we must treat.

Case i) The algorithm ϕ_i had at the end of stage y a priority higher than the ultimate priority of ϕ_j. In this case it is removed after stage y but before or at stage w, and we are done.

Case ii) The algorithm ϕ_i had at the end of stage y a lower priority than the ultimate priority of ϕ_j.
These are two subcases:
Case ii)a) ϕ_j is not on the black list at the end of stage y .
By Specification F b) we know that the value r of result satisfies $\Phi_j(y) \leq r$ as ϕ_j is not on the black list and has a higher priority than the selected function ϕ_i. Now t'(y) is made equal to r. We have again a contradiction as we assumed $t'(y) < \Phi_j(y)$.

Case ii)b) ϕ_j is on the black list. By Specification F c) we know that ϕ_j does not satisfy conditions a) and b) in Specification F for any value of w. This means that there exists no w such that both

a) $w \geq \Phi_k(y)$ for those k having higher priority than ϕ_j and not being on the black list

b) $w < \Phi_j(y)$.

Now take $w = t'(y)$. We know by assumption that $t'(y) < \Phi_j(y)$. Furthermore $t'(y) \geq \Phi_k(y)$ for all ϕ_k which are not on the black list at stage y and have a higher priority than ϕ_i which has a lower priority than ϕ_j.

Again a contradiction arises. We assumed in specification F that the selected program ϕ_i should have a highest possible priority but also ϕ_j is a solution.

Proof of Claim 3: The procedure to test $t'(x) = y$ is the following. First search for a stage number z such that $z \geq y+1$ and $\sigma(z) = x$. Let the algorithm to compute t' run up to stage z. If $t'(x)$ is found to be computed before or at stage z compare the result with y. If $t'(x)$ is not yet computed the answer of the test is negative.

To see the correctness of this procedure we must show that a failure to compute $t'(y)$ before or at stage z indicates that $t'(x)$ is either undefined or gets a value $\geq y$. Now suppose $t'(x)$ is computed at stage $z' > z$. The value of $t'(x)$ is either defined by the ad hoc definition or by positive result in SEARCHTIME. In both cases $t'(x) \geq w$ where w is the value stored in result after the completing of the call of SEARCHTIME during stage z'.

Now we know that failure occurred during execution of SEARCHTIME at stage z. Our try at stage z' may have succeeded or not. If failure occurred again the value w of result upon exit is equal to the value of bound at stage z' which is $\geq z'$ (Specification F). If no failure occurred the value w stored in result exceeds the number used for bound at stage z. (Specification I) which is $\geq z$.

Hence we have:

failure at stage z'	$t'(x) \geq w \geq z'+1 > z > y$
succes at stage z'	$t'(x) \geq w \geq z > y$

hence $t'(x) > y$.

If t'(y) does not converge there is nothing to prove.

This completes the proof of the naming theorem.

Remark By our choice of σ, σ never becomes zero. Consequently t'(0) is never computed. One might define t'(0) = 0 before starting the algorithm. In both cases a number of assertions made on convergence and meaning of t'(x) may be false for x = 0. As this gives no difficulties in the proof of the naming theorem (which allows us to fool around with the first 10^{100} values of t') the finding of these "errors" is left to the reader.

§3c. The ALGOL text of the algorithm - the procedures

We assume the following procedures to be declared outside our program, or in the outermost block.

```
     integer procedure sigma(n); value n; integer n;
```

sigma computes a total function which is never zero but assumes each positive value infinitely often; and satisfying $\sigma(n) < n$ for $n \geq 2$.
Example: the function

$$\text{sigma}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ 2\uparrow k & \text{if } n = 2\uparrow(k+1) \\ n-2\uparrow \text{ entier}({}^2\log(n)) & \text{otherwise} \end{cases}$$

is computed by the program (procedure body)

```
     if n ≤ 3 then sigma := 1
else begin integer k, l;  k := n; l := 1;
          for k := k ÷ 2 while k > 0 do l := l*2;
          sigma := if l = n then n ÷ 2 else n - l
     end;
```

Proof: Trivial for $n \leq 3$. For larger n the following assertions remain unchanged during each successfull execution of the while loop:

i) l is a power of 2.

ii) $n \geq k * l$

iii) $n-k*l < l$.

ii) follows as we have $k\div 2 \leq \frac{1}{2}*k$ and hence
$2*l*k\div 2 \leq 2*l*k/2 = l*k \leq n$.

iii) is equivalent to $n < (k+1) * l$. This follows as
$(k\div 2+1) * l*2 \geq (k+1)/2 * l*2 = (k+1)*l > n$.

Execution of the while loop is interrupted when k becomes 1 but not before : our relations now give

$$l \leq n < 2*l.$$

Hence l is the largest power of 2 smaller than n. If l = n we make sigma = n÷2 proving the second line of the definition; otherwise we make sigma = n-l proving the third of the definition.

<u>procedure</u> ORDER BY PRIORITY (y,list,functions);<u>value</u> y;
<u>integer</u> y; <u>integer</u> <u>array</u> list, functions;

list is an array containing the priorities of the algorithms $\phi_1,\dots,\phi_y$ (at the end of <u>stage</u> y), a minus sign indicating the algorithm to be on the black list. The procedure stores the indices of the algorithms in order of decreating priority in the array "functions". Again a minus sign indicates that the algorithm is on the black list.

```
integer procedure Nextprior;
        Nextprior := P := P+1   ;
```

P is a globally declared integer variable which is initialized at zero and which is never reverred to outside Nextprior. Thus each call of Nextprior results in the increasing of P by one the resulting value being returned as result of Nextprior. This guarantees that the priority number assigned somewhere in the program by using Nextprior is always a higher number than all priority numbers given before. We shall see that no priority numbers are given without a previous call of Nextprior to compute this number.

<u>procedure</u> HALT;

HALT stops the execution of the program

<u>procedure</u> OUTPUTT(n);<u>value</u> n; <u>integer</u> n;
<u>procedure</u> OUTPUTT(b);<u>value</u> b; <u>boolean</u> b;

OUTPUT(OUTPUTT)results in the outputting of the value of the actual parameter.

<u>integer</u> <u>procedure</u> IN;
<u>boolean</u> <u>procedure</u> INN;

IN(INN) results in reading a value from some external medium.

<u>Integer</u> <u>procedure</u> UNIVERSAL(i,x,bound); <u>value</u> i,x,bound;
<u>integer</u> i,x,bound;

UNIVERSAL computes the function ϕ_i at argument x for at most bound steps in some complexity measure. If the computation fails UNIVERSAL is given the value -1;

<u>Boolean</u> <u>procedure</u> MEASURE(i,x,y); <u>value</u> i,x,y; <u>integer</u> i,x,y;

MEASURE computes the value of the predicate $\Phi_i(x) \leq y$.

<u>Integer</u> <u>procedure</u> MAXTIME(y,t); <u>value</u> y,t; <u>integer</u> y,t;

MAXTIME computes the function G(y,t) used to fix the maximal runtime for SEARCHTIME at a stage where t(y) is computed.

The independence of the subroutines makes it possible to use for UNIVERSAL any program simulating some universal machinery. The structure of the procedure MEASURE is unknown but its existence follows from Ax II. The procedure MAXTIME depends on the structure of SEARCHTIME. We shall discuss this procedure again after having defined SEARCH (=SEARCHTIME without clock).

<u>The procedures SEARCH and SEARCHTIME</u>

The central routine in PART 2 is the algorithm which tries to compute t'(y). It has to find a solution to the following problem:

Given a sequence of programs $\phi_1 \ldots \phi_y$ in order of descending priority. If the function ϕ_i is on the black list it requests to have $\Phi_i(y) > t'(y)$; if ϕ_i is not on the black list it requests to have $\Phi_i(y) \leq t'(y)$. Find the function ϕ_j on the black list with the highest priority for which a value t'(y) can be found such that $\Phi_j(y)$ exceeds t'(y) but that $\Phi_i(y) \leq t'(y)$ for all functions ϕ_i not on the black list with higher priority.

To understand and to discuss our program we translate the problem into the following equivalent version:

The Treasurer General of HARAD is asked to fix a price for the high quality gadgets produced in his country. There are a number of inhabitants of HARAD which either are producing gadgets or are consuming them.

As is common in corrupt societies (like HARAD) the Treasurer has a strict preference for some of the inhabitants and a dislike for others; the inhabitants are ordered sequentially by the amount of friendship to the Treasurer.

It is against the sense of ethics of the HARADiens to inquire after the price for which they are selling or buying gadgets. However if you ask them whether they are willing to sell or buy gadgets for a certain price they must say yes or no.

The Treasurer is allowed to keep the difference between the price paid by the consumer and the price asked by the producer. He will not fix a price unless he is earning at least one credit for each sale made.

How has the Treasurer to compute in order to find a price accepted by the most beloved consumer willing to pay a price higher than the prices asked by all the more beloved producers before him.

Solution: The Treasurer visits his nearest friend. If this man happens to be a consumer willing to pay at least one credit he will get the gadgets for the lowlow price of one credit. If this man however is a producer (a consumer not willing to pay anything is skipped) the price must be at least the price asked by this man. As the Treasurer can not ask him to name his price he has to start offering him time and agian a new price until the man accepts (say for K credits). After this the Treasurer goes to the next man. If this is also a producer the Treasurer asks him whether he is also willing to produce for K credits. If not the price is raised again step by step until this man accepts also; after this the Treasurer goes to the next man.

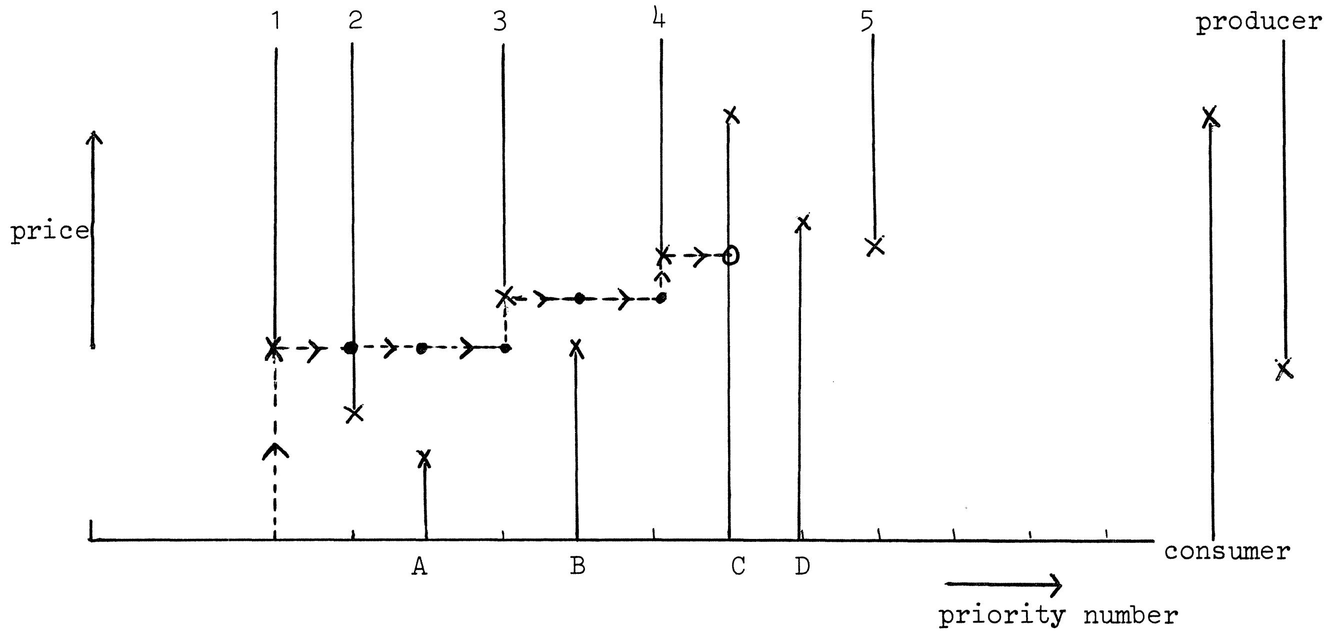
Suppose however that the second man is a consumer. The Treasurer asks him whether he wants to pay the price of K+1 credits. If so the solution is found; otherwise the Treasurer goes to the next man.

The computation for the k-th man is equal to the computation given above for the second man ($k \geq 2$).

In translating the problem of our algorithms and runtimes into the economical problem above one should think the functions on the black list to be the consumers; the nice functions not on the black list are the producers. The unknown prices p_i are the runtimes $\Phi_j(y)$. Only the final price x should be decreased by one to give a solution to our original problem. (which is always possible as $x \geq 1$). Compare:

x at least one more than the price p_j of producer j : $x > p_j$	$\Phi_j(y) < x \equiv \Phi_j(y) \leq x-1$
x less or equal than the price p_j offered by consumer j: $x \leq p_j$	$\Phi_j(y) \geq x \equiv \Phi_j(y) > x-1$.

For an example see the next diagram;



The path of the computation is indicated in the diagram. Consumer C is willing to pay more than the price of producers 1,...,4. The requests of consumers A (resp. B) are refuted by the prices asked by producer 1 (resp. 3). The requests of D and 5 are not taken into consideration.

The given solution is formalized by the following ALGOL program:

```
procedure SEARCH(y,list,index,result); value y; integer y, index, result;
                                       integer array list;
begin integer array functions[1 : y];
      integer k, value, candidate;
            value := 0;
            ORDER BY PRIORITY(y,list,functions);
      for candidate := 1 step 1 until y do
NEXT MAN: if functions[candidate] > 0 then begin
PRODUCER: for k := value while  ¬ MEASURE(functions[candidate],y,value) do
            value := value + 1            end
                                       else
CONSUMER: if ¬ MEASURE(-functions[candidate],y,value) then
          begin index := - functions[candidate]; result := value; goto EXIT
                                                                       end;
GIVE UP:  value := value + 1; goto GIVE UP;
EXIT:
end SEARCH
```

The program tests the candidates in order of their priority (for candidate := 1 step 1 until y do). First their presense on or of the black list is tested (NEXT MAN). If the function is not on the black list value is increased until $\Phi_j(y) \leq$ value (value := value + 1 as long as MEASURE(j,y,value) is false).
If the function j is on the black list one tests whether $\Phi_j(y) >$ value is true (¬ MEASURE(j,y,value)); if so the function is accepted; otherwise we do nothing (i.e. we proceed to the next candidate).

If we run out of candidates we arive in GIVE UP.
At this place the execution gets in a loop and value is increased beyond all bounds. The only other reason why the algorithm may fail to stop (by a jump to EXIT) is that we may fail to leave the while loop in PRODUCER. This case arives if we are asking a producer for a price while this producer is not willing to sell for any price at all!
(So we are asking for the running time of an algorithm which does not converge). Also in this case the contents of value increases to infinity.

This makes it possible to use value as clock to shut off the computation of SEARCH. Using this clock has some other advantages: The program for SEARCHTIME(y,list,index,result,bound) is derived from the above program for SEARCH by
1e) replacing at two places the statement

```
                    value := value + 1
```

by

```
   begin value := value + 1; if value > bound then goto TIMEOUT end
```

or an equivalent procedure statement (see complete text of the program).
2e) further one must introduce between GIVE UP and EXIT the labeled statements:

```
   TIMEOUT: index := -1; result := value;
```

MAXTIME becomes now very simple as we may take $G(y,t) = t+1$.

The complete text of SEARCHTIME is given in section 3d.

Remarks: 1) The use of $G(y,t) = t+1$ is made possible by the choice of our ad hoc clock for the algorithm SEARCH.
The use of the contents of value as clock for SEARCH is made possible by the fact that value is increasing during execution of search. The same monotony was used in the proof of CLAIM 3. This argument however does still not justify the use of timebound Φ_t in the proofs in [1] and [2]. We shall discuss this in section 4.

2) One might ask whether it is possible to extend our choice of a clock for SEARCH to a complete complexity measure. In order to do this one must first be able to recognize SEARCH as being a function-computing algorithm. In doing this the array "list" has to be taken as one single argument. Sure it is possible to encode arrays by integers but the computation of SEARCH is defined only if the number of entries in the array is equal to y.(Otherwise we may get nonsense out of ORDER BY PRIORITY). Hence we will have to extend SEARCH by a routine which tests whether the arguments are compatible, and which if so starts SEARCH as before, and otherwise jumps to GIVE UP.

For the value computed we take the value stored in result upon halting.

Let $\Xi = \{\{\xi_i\},\{\Xi_i\}\}$ be an arbitrarily complexity measure. Let s be an index for the extended SEARCH. The incompitability of arguments results in the increasing of value above all bounds; if SEARCH fails to halt for some other reason this is also the case. This shows again that the function SEARCH as a singleton forms a measured set. Let j be an index for SEARCH* (extended SEARCH). Replace in Ξ Ξ_j by ξ_j. The modified measure Ξ' is again a complexity measure which extends our choice for a clock for SEARCH.

3) To see that a choice for G (see global description of part 2) is possible in every complexity measure we estimate the number of steps in some "time measure" (see discussion after Th. 1.3). By Th.1.3 the general case will follow, although the bounds given by Th.1.3 may fail in finitely many cases - see also section 4.

Assuming that SEARCH(y,list,index,result) stops the number of steps taken by SEARCH can be estimated by:

$$\begin{aligned} &\#\text{ steps for initialisation} \quad + \\ + \;&\#\text{ steps in ORDER BY PRIORITY} \quad + \\ + \;&(\text{Max}\{\#\text{ steps in MEASURE}\} + \text{constant}) * (\#\text{ calls for MEASURE}) \leq \\ &C_1(y) + C_2 * y^2 + (y+\text{result}) * \max_{\substack{i\leq y\\ v\leq \text{result}}} (\#\text{ steps for MEASURE}(i,y,v)+C_3) \leq \\ &\qquad G(y,\text{result}) \\ &\qquad\qquad \text{for some } G \in R_2. \end{aligned}$$

This proves that a program for MAXTIME exists whatever time measure is used as clock in SEARCHTIME.

<u>The procedures PART 1 and PART 2</u>

These two procedures together define the computation at <u>stage</u> x.
Assumed to be declared and defined globally are:

An integer variable i representing an index for the timebound t.

An integer variable x indicating the stage number;

An integer array t[1:x] containing the values of the old timebound if computed. t[i] = -1 denotes that t(i) is not yet computed. So in general t[x] = -1 at the beginning of PART 1, stage x;

An integer array t1[1:x] containing the values of the new timebound t'. If not yet computed t1[i] = -1.

An integer array prior[1:x] representing the requests of the programs $\phi_1, \ldots, \phi_x$ at the beginning of stage x. |prior[i]| is the priority number while a minus-sign indicates the function to be on the black list.

An integer variable y containing the value of sigma(x);

An integer array oldprior[1,y] which is equal to the contents of the array prior[1,y] after the completing of PART 2 at stage y.

An integer variable testvalue containing the z in $t'(y) \leq z$? if one is measuring t' at y.

A Boolean variable test used to store the answer to the above question.

The program follows strictly the description presented before in section 3b.

```
procedure PART 1(t,t1,prior); integer array t,t1,prior;
begin integer j,jj,q;
      t[x] := t1[x] := -1; prior[x] := Nextprior;
      for j := 1 step 1 until x do
          if t[j] = -1 then
          begin q := t[j] := UNIVERSAL(i,j,x);
                if q ≠ -1 then
                for jj := 1 step 1 until x do
        COMPARE: if prior[jj] > 0 ∧ ¬MEASURE(jj,j,q) then
                     prior[jj] := -Next prior;
          end
end PART 1  ;
```

```
procedure PART 2(t,t1,prior,oldprior); integer array t,t1,prior,oldprior;
          if t1[y] < 0 then
begin     integer index, result, bound;
          boolean last time;
          last time := t[y] ≠ -1;
          if last time
          then begin bound := MAXTIME(y,t[y]);
                     if x ≥ bound then bound := x
               end.
          else bound := x ;

          SEARCHTIME(y,oldprior,index,result,bound);
 OPTION:  test := result ≤ testvalue;
          if index > 0
     then begin if prior[index] < 0 then prior[index] := Nextprior;
                     t1[y] := result
          end
     else if last time then t1[y] := result
end PART 2  ;
```

Remarks: A) In the global description we claimed that t'(y) should be greater than t(y) if we define t'(y) by failing for the last try. This is quaranteed by the choice of the bound in SEARCHTIME which is sufficient to have t'(y) = t(y)+1 tested.

B) The variable result containing the integer "how far" SEARCHTIME has come gets lost upon leaving PART 2. Therefore the eventual inequality result > testvalue used in measuring the new time bound t' is stored in a global Boolean test.

C) The integer arrays are introduced as formal parameters in order to have the procedures declared in the outer most block of the program.

D) During stage 1 there is no usefull information present in ooldprior, hence also not in oldprior (see STAGE). Hence we suppress the execution of PART 2 at stage 1.

The procedure STAGE

The only task remaining for stage is the creation of the necessary amount of storage and the interpretation of the results of PART 1 and PART 2;

Assumed to be declared globally are the integer variables x, arg, testvalue and i and the boolean variables test, answer and asking.

```
procedure STAGE  ;

begin       own integer array t, t1, prior[1:x], ooldprior[1:x*(x+1)÷2];
         integer k, kk ;
            y := sigma(x); k := y*(y-1)÷2 ;
         begin integer array oldprior[1:y];
               if x=1 then oldprior [1] := 1 else
               for kk := 1 step 1 until y do oldprior[kk] := ooldprior[k+kk];

               PART 1(t,t1,prior);
   if x>1 then PART 2(t,t1,prior,oldprior)
         end ;
         if t1[y] ≥ 0  ∧ arg = y then
         begin OUTPUT(t1[y]); answer := t1[y] ≤ testvalue;
               OUTPUTT(answer); HALT
         end
   else if arg = y   ∧ asking then
        begin answer := test; if ¬ answer then
              begin OUTPUTT(answer); HALT end
        end ;
        k := x*(x-1)÷2 ;
        for kk := 1 step 1 until x do ooldprior[k+kk] := prior[k] ;
end STAGE ;
```

Remarks: A) It is assumed that asking is true iff one is interested to know whether t'(arg) ≤ testvalue is true. If asking is false the program is in the first place trying to compute the value for t'(arg).

B) The reason for a call for HALT is

i) t'[arg] is found to be computed

ii) we are measuring t' at arg and we have concluded that
t'[arg] $\leq$ testvalue is false.

As we see in the complete program the program is only stopped by a call for HALT.

C) At stage 1 the array ooldprior is declared up to ooldprior[1] but no useful value is stored in this place. To prevent the undefined result of the fetching of an undefined value we take oldprior[1] = 1 at this stage. This has no further influence as the execution of stage 1 PART 2 (the only place where oldprior is used) is suppressed.

The complete program.

```
begin  integer x, arg, testvalue, P, i, y                 ;
       boolean test, answer, asking                       ;
       <declarations of procedures given before>          ;
       P := 0 ; x := 0;                                   ;
       arg := IN; testvalue := IN; i := IN; asking := INN;

       for x := x+1 while true do STAGE
end
```

It remains still to indicate how the program can be shown to be correct. The program as it is given is nothing but a concrete realisation of the algorithm as given by Meyer & McCreight. We have seen in section 3.b that their correctness proof cannot be given without assuming a number of concrete specifications about the subroutines which are taken as primitive in their description. This is especially necessary for the subroutine which we have called SEARCHTIME; this routine is not specified in the descriptions in [1] and [2] but the "number of steps" allotted to it is precisely given.

The concreteness of our program is incomplete insofar that the three subroutines MEASURE, UNIVERSAL and ORDER BY PRIORITY remain undeclared. However it is possible by inserting three declarations of these procedures to compose a working program. (In doing this one may take for MEASURE and UNIVERSAL a pair of functions which have no relation to any complexity measure or universal computing machine at all. The only function of UNIVERSAL is to compute a function or not - depending on the value of bound; while MEASURE has to tell for which i, x and y a "violation" occurs).

This way the algorithm can be tested on a real computer. (Appendix II contains such a framed-up-version of our program).
The overall structure of the program is clear. x and P are initialized at 0; the values of the variables arg, testvalue, i and asking are initialized from outside. Then the procedure STAGE is called within an infinite loop. The controlled variable in this loop is x. x is increased by 1 before a new call of STAGE is issued. The value of x at the first call of STAGE equals 1.

Inspection of the program* learns that nowhere within STAGE or within a subroutine of STAGE the value of x is changed. This shows that x performs the function of the stage number indicated in the global description. The value of y is made equal to sigma(x) at the beginning of STAGE and remains unchanged until the next call of STAGE. (Procedures having y as actual parameter call y by value).

The variable P does not occur in the program outside the procedure. Nextprior. Within this procedure the value of P is increased by one and the result is given as the value computed by Nextprior. Hence we conclude that Nextprior delivers at each call a positive value which is larger than all values delivered befor.

The crucial data structure is the own array prior. The name prior occurs also as the formal parameter in PART 1 and PART 2. These procedures are called within the procedure STAGE and at this call the actual parameters called by name have the same name as the formal parameters.

Assignments to elements of prior occur at three places:

*) the program given in section 3d.

1^e) In PART 1 at the beginning

```
        prior[x] := Nextprior
```

2^e) In PART 1 in COMPARE:

```
    if prior[pp] > 0 ∧  MEASURE(pp,p,q) then prior[pp] :=-Nextprior
```

3^e) In PART 2, the line after OPTION:

```
    if prior[index] < 0 then prior[index] := Nextprior
```

For a fixed variable prior[t] an assignment of type 1^e) occurs only once (at the beginning of _stage_ t) and assignments of type 2^e) and 3^e) occur only afterwards as prior[t] does not exist before _stage_ t !
Note that we have pp $\leq$ x (pp is controlled variable in _for_ pp := 1 _step_ 1 _until_ x _do_ ...) .
We have also index $\leq$ x for index =- functions[candidate] where 1 $\leq$ candidate $\leq$ y , functions[candidate] < 0 and 1 $\leq$ abs(functions[candidate]) $\leq$ y (assuming correctness of ORDER BY PRIORITY!).

One concludes that each value assigned to a member of prior has an absolute value computed by Nextprior (hence larger than all values computed before) and that except for the first assignment (of type 1^e)) the sign of the value assigned to prior[t] is the opposite of the sign of the old value.
The storing of the values of prior in ooldprior occurs at the end of STAGE.

This proves Specification B.

We see that only an assignment to prior[t] of type 3^e) makes a negative value positive. This action is followed by the assignment t1[y] := result In the discussion of SEARCHTIME we shall indicate that the value of result is $\geq$ 0 and that MEASURE(index,y,result) is _false_. Furthermore the computation of PART 2 is suppressed if t1[y] $\geq$ 0 as the body of PART 2 has the structure

```
        if t1[y] < 0 then <ST>  .
```

This proves Specification A.

SPECIFICATIONS C and D are taken for granted. We are allowed to assume that the undeclared subroutines are correct.

SPECIFICATION E is nothing but a prediction of the structure of PART 1. This structure is easily compared to the structure of the text of PART 1. As we have seen before the only assignment of a negative value to prior[t] is the assignment of type 2^e) which occurs in PART 1.

Assignment to t1 occurs at three places:

1^e) at the beginning of PART 1   t1[x] := -1,
2^e) in PART 2 the line after OPTION:

```
        if index > 0 then begin .....; t1[y] := result  end
```

3^e) in PART 2 two lines after OPTION:

```
    else if last time then t1[y] := result.
```

Assignment of type 1^e) does not result in a definition of t1(y) as -1 represents the "value" undefined.
For assignment of type 2^e) or 3^e) a necessary condition is index > 0 (which means a positive result of SEARCHTIME) or lasttime ≡ <u>true</u> (which means that t(y) is defined (the only assignment to lasttime which precedes the assignment of type 3^e) is lasttime := t[y] $\neq$ -1).

These assertions prove Specification H.

The remaining specifications (F,G and I) deal with the procedure SEARCHTIME.

Assuming ORDER BY PRIORITY to be correct we know that functions[1:y] is filled with the numbers $\pm 1 \ldots \pm y$ permuted in such a way that abs(prior[abs(functions[j])]) is monotonously increasing and that sign(functions[j]) = sign(prior[abs(functions[j])]).

The work of SEARCHTIME is performed by the <u>for</u> loop <u>for</u> candidate := 1 <u>step</u> 1 <u>until</u> y <u>do</u> <ST>. Consequently candidate is monotonously increasing from 1 upto y. If for candidate = y execution of <ST> is completed we arive in GIVE UP.

GIVE UP: labels a line which ends with a jump to GIVE UP; the netto result of execution of this line is the execution of CLOCK(value, TIMEOUT,bound) i.e. the execution of

```
    if value ≥ bound then goto TIMEOUT else value := value+1.
```

i.e. "increase value by one but do not exceed bound"
Consequently we jump to TIMEOUT before executing GIVE UP bound +1 times.

The <ST> in the for loop is in fact a conditional statement

```
    if functions[candidate] > 0 then <STProd> else <STCons>
```

Hence for a single execution of <ST> either <STProd> or <STCons> is executed but not both of them.
During execution of CONSUMER no assignment to value is performed. The statement <STProd> is in fact a compund statment consisting of one for statement: its function can be represented by

```
    while value < $\Phi_{\text{functions[candidate]}}(y)$     do
        "increase value by one but do not exceed bound"
```

if value exceeds bound we jump to TIMEOUT.

The following correctness relations can be shown to hold.

1^{e}) During execution of SEARCHTIME the inequalities $0 \leq$ value $\leq$ bound remain true.
If value = bound and the procedure CLOCK is called execution of SEARCHTIME is completed via TIMEOUT.

2^{e}) If j < candidate and functions[j] > 0 then value $\geq \Phi_{\text{functions[j]}}(y)$ with equality holding for at least one such j or value = bound.

The assertion 2^{e}) is void for candidate = 0. Assume 2^{e}) to be true for candidate = l. If functions[l] < 0 the assertion is trivially true for l+1. However if functions[l] > 0 for candidate = l we execute <STprod> and there value is stepwise increased until value $\geq \Phi_{\text{functions[l]}}(y)$ or value = bound becomes true. If value is not increased the equality still holds for the same j as before. Otherwise we have equality now for j = l. (Unless value =, bound).

There are two ways to complete the execution of SEARCHTIME

1^e) via TIMEOUT. This represents failure. The only way to arrive in TIMEOUT is by a call of CLOCK with value = bound. Hence the result will be index = -1, result = bound.

2^e) via jump to EXIT. The instruction <u>goto</u> EXIT is executed only in CONSUMER if a certain condition holds. Because we are in CONSUMER we know functions[candidate] < 0. Now the condition for a jump to EXIT is MEASURE(-functions[candidate],y,value) $\equiv$ <u>false</u>
i.e. $\Phi_{\text{abs(functions[candidate])}}(y) > \text{value}$

or considering the assignments index = -functions[candidate]; result := value; which are executed at the same time

$\Phi_{\text{index}}(y) > \text{result}.$

We have seen already that the value of result is assigned to t1[y] if index > 0 hence we have

$\Phi_{\text{index}}(y) > \text{t1[y]}$.

These assertions prove the essential statements made in Specification F Fa) is shown above and Fb) follows from our second correctness relation. The fact that in this correctness relation equality holds for at least one j shows that value is kept as low as possible. If there should exist a solution ' with index' < index this means that for candidate = index' the conditions for a jump to EXIT are fulfilled, so this jump is executed and consequently candidate does not increase upto index anymore. This proves Fc).

The only assertion made in F which remains to be checked is the assertion about the contents of oldprior. Now oldprior is filled with the contents of ooldprior and more precisely with the elements
ooldprior[k+kk] for kk = 1,...,y and k = y*(y-1)÷z.
Assignment to these elements of ooldprior has been executed at stage x' = y; after this the values stored in ooldprior remain unchanged. Hence oldprior[1:y] contains the values of prior[1:y] at the end of execution of <u>stage</u> y (The case x = y = 1 is excluded in this argument).
q.e.d.

Specification G claims that SEARCHTIME succeeds when a number of assumptions are made. Suppose functions[j] < 0 and MEASURE(-functions[j], y,bound) ≡ false.

There are two possibilities:

a) During execution of SEARCHTIME candidate := j is executed.
   We conclude that for candidate = j <STcons> is executed as also MEASURE(-functions[j],y,value) ≡ false assignments index := -functions[j], result := value and the jump goto EXIT is executed.
   This represents a possible realisation of I).

b) Candidate := j is not executed. This means that either a jump goto EXIT or a jump goto TIMEOUT is executed. In the first case there was a positive result and we know that
   abs(prior[-functions[j]]) > abs(prior[index]), i.e., a function with higher priority has been moved to the white list. Again this represents a possible realisation of I.
   In the second case we have executed for some l < j within PRODUCER
   CLOCK(value,Timeout,bound) with
   value = bound.
   This means that we have
   functions[l] > 0 and
   MEASURE(functions[l],y,bound) ≡ false

This represents a realisation of II.

This proves Specification G.

The only place where bound is used in SEARCHTIME is within the procedure CLOCK. Now if bound' > bound and CLOCK(i,L,bound) results in i := i+1 then also CLOCK(i,L,bound') will result in i := i+1. So the increasing of bound delays the jump to TIMEOUT. If a call of SEARCHTIME does not lead to a jump to TIMEOUT and if bound is increased the new call will execute the same actions as the old call. If the first call does lead to a jump to TIMEOUT then the second call executes the same actions as the first call until value = bound. At this place the first call executes goto TIMEOUT and the second call executes value := value+1, However value will not decrease afterwards so we know that any number stored in result is at least bound + 1.

This proves Specification I.

We now have indicated that the specifications and predictions of our program are correct. We conclude with some remarks on the execution of the program as a whole.
Inspection of the program learns that there exist no procedures which (indirectly) may call themselves. The only place at which a jump backwards occurs is in SEARCHTIME at the label GIVEUP. We have seen above that this creates a loop from which the program escapes by a jump to TIMEOUT.

There are a number of for statements in the program. With three exceptions these are of the type

```
   for k := 1 step 1 until Q do <S.T.>
```

where Q is x or y, both variables which have a value which is unchanged within the statement <ST>.

At three places a for while loop occurs. The first place is within the procedure sigma; this while loop creates no difficulties (we did prove the correctness of sigma before).
A second while loop occurs in SEARCHTIME within <STprod>. Again the program escapes from this while loop by a jump to TIMEOUT.

The third while loop is the overall loop in the program:

```
   for x := x+1 while true do STAGE.
```

The conclusion which we are allowed to draw from this inspection is that every call of STAGE leads to a finite computation. On the other hand the normal completion of a call of STAGE will result in a new call. Hence the only way the program can terminate is by call of the procedure HALT. Such a call can occur at two places in STAGE.
The necessary and sufficient conditions are:

either i) $y = arg \quad \wedge \quad t1[y] > 0$

or ii) $\quad y = arg \wedge asking \wedge \neg answer.$

If i) occurs we have computed the value of t1 for the argument for which we were trying to compute t1.
If ii) occurs we have not yet computed t1(arg); however the fact that answer = false indicates that test ≡ false. In the call of PART 2 we did complete at this stage we have executed test := result ≤ testvalue; Hence test ≡ false means that result > testvalue. This means as we indicated in the proof of claim 3 that we can conclude that t1(arg) > testvalue. An asking is true we have the answer to the question we were interested in.
Our conclusion is that the program terminates if and only if it delivers the answer we want from it.

This completes the discussion of the program and the proof of the naming theorem.

§3d. The complete ALGOL text

```
begin           integer x,arg,testvalue,p,i,y        ;
                boolean test,answer,asking           ;

   integer procedure sigma(n); value n; integer n    ;
           if n ≤ 3 then sigma := 1
   else begin integer k,l ;
                k := n; l := 1;
             for k := k÷2 while k > 0 do l := l*2 ;
                sigma := if l = n then n ÷ 2 else n-1
        end ;

   integer procedure Nextprior ;
           Nextprior := P := P+1  ;

   integer procedure MAXTIME(y,t); value y,t; integer y,t;
           MAXTIME := t+1  ;

   procedure CLOCK(i,L,time); value time, L; integer i, time;
                                  label L  ;
           if i ≥ time then goto L else i := i+1;

   procedure ORDER BY PRIORITY(y,list,functions); value y; integer y;
                                       integer array list, functions  ;
           <body to be filled in by the reader>

  integer procedure UNIVERSAL(i,x,bound); value i,x,bound ;
                    integer i,x,bound  ;
           <body to be composed by the user>                     ;

   Boolean procedure  MEASURE(i,x,y); value i,x,y; integer i,x,y;
           <body to be presented by the person proposing the condidered
            abstract complexity measure>                         ;

   procedure SEARCHTIME(y,list,index,result,bound); value y, bound;
             integer y,index,result,bound; integer array list;
```

```
begin integer array functions[1:y];
      integer k,value,candidate ;
      value := 0 ;
      ORDER BY PRIORITY(y,list,functions);
      for candidate := 1 step 1 until y do
NEXT MAN:  if functions[candidate] > 0 then     begin
PRODUCER:  for k := value while  ¬ MEASURE(functions[candidate],y,value)
                                                                     do
                 CLOCK(value,TIMEOUT,bound)        end

  else
CONSUMER:  if ¬MEASURE(-functions[candidate],y,value) then
           begin index := -functions[candidate]; result := value;
                    goto EXIT end;

GIVEUP:    CLOCK(value,TIMEOUT,bound); goto GIVEUP;
TIMEOUT:   index := -1; result := value;
EXIT:
end SEARCHTIME                                                  ;

procedure PART 1(t,t1,prior); integer array t,t1,prior ;

begin  integer j,jj,q   ;
       t[x] := t1[x] := -1; prior[x] := Nextprior;
       for j  := 1 step 1 until x do
           if t[j] = -1 then
           begin q := t[j] := UNIVERSAL(i,j,x) ;
                 if q ≠ -1 then
                 for jj := 1 step 1 until x do
       COMPARE:  if prior[jj] > 0  ∧ ¬ MEASURE(jj,j,q) then
                    prior[jj] := -Nextprior
       end
end PART 1
                                                                 ;
```

```
procedure PART 2(t,t1,prior,oldprior); integer array t, t1,prior,oldprior;
      if t1[y] < 0 then
begin   integer index,result,bound;
        boolean last time ;
        last time := t[y] ≠ -1 ;
        if lasttime
      then begin bound := MAXTIME(y,t[y]);
                 if x ≥ bound then bound := x
           end
      else bound := x     ;
SEARCHTIME(y,oldprior,index,result,bound).
OPTION: test := result ≤ testvalue;
        if index > 0
      then begin if prior[index] < 0 then prior[index] := Nextprior;
                 t1[y] := result
           end
      else if lasttime then t1[y] := result
end PART 2                                                          ;

procedure STAGE ;
begin   own integer array t,t1,prior[1:x], ooldprior[1:x*(x+1)+2];
        integer k,kk            ;
        y := sigma(x); k := y*(y-1) + 2 ;
        begin  integer array oldprior [1:y];
               if x = 1 then oldprior[1] := 1 else
               for kk := 1 step 1 until y do oldprior[kk] := ooldprior[k+kk];
               PART 1 (t,t1,prior);
 if x > 1 then PART 2(t,t1,prior,oldprior)
        end;
        if t1[y] ≥ 0  ∧ arg = y then
        begin OUTPUT(t1[y]); answer := t1[y] ≤ testvalue ;
              OUTPUTT(answer); HALT
        end
```

```
   else if arg = y  ∧  asking then
        begin answer := test; if ¬ answer then
                  begin OUTPUTT(answer); HALT end
        end;
   k := x*(x-1)÷2 ;
   for kk := 1 step 1 until x do ooldprior[k+kk] := prior[k];
end STAGE                                                          ;

BEGIN OF PROGRAM: x := P := 0;
                  arg :=IN; testvalue :=IN; i :=IN; asking :=INN;
   for x := x+1 while true do STAGE
end complete program
```

§4. Discussion of earlier proofs

As we indicated in section 3 the essential difference between our proof and the proofs described in [1] and [2] lies in the clock used to shut off SEARCHTIME. We have seen that one can use the number stored in value as clock with a very simple timebound. Any other measure is as correct provided the bound chosen in searchtime at the moment $t(y)$ is already computed is large enough to test for all numbers up to $t(y) + 1$ whether they can be used as the value for $t'(y)$.

The proofs in [1] and [2] give no definition of the measure in which the number of steps taken in searching for the value of $t'(y)$ is counted. Hence we may assume that this is the abstract measure Φ which is considered in the theorem as a whole. If this assumption is correct the timebound for MAXTIME $= \Phi_t(y)$ is not correct as can be concluded from the following example.
The example is constructed by choosing a set of timebounds $\{t_{p(i)}\}_{i=1}^{\infty}$ which are easily computed and by making the cost of SEARCHTIME so expensive that it never will deliver any useful result.
The example uses also the definition of $t'(y)$ in the give up case used in [1] and [2] which is distinct from the one in our proof. In these proofs the value of $t'(y)$ is made equal to $\max\{t(y),\Phi_t(y)\}$ if failure occurs when $t'(y)$ is computed for the last time in SEARCHTIME. We shall see that the example can be modified that it works also for our definition $\{t'(y)$ becomes the last value tried$\}$.

Another difference between our proof and the proofs in [1] and [2] lies in the stage x for which $t(y)$ is computed.
In our proof this happens at stage $\max\{x,\Phi_t(y)\}$. In the proofs in [1] and [2] this stage can be delayed as only one new value for t is accepted at a certain stage number. The delay is however at most x stages. The difference is not essential.

Example: [An example showing that timebound MAXTIME $= \Phi_t$ does not give $C_t = C_{t'}$].

Let h be an arbitrary total recursive function; $h(x) \geq x$. Let X_i be the recursive set $\{(2i-1) * 2^k | k \in \mathbb{N}\}$. We have $X_i \cap X_j = \emptyset$ for $i \neq j$ and $\bigcup X_i = \mathbb{N}\backslash\{0\}$. Let χ_i be the characteristic function of X_i:

$$\chi_i(x) := \underline{\text{if}}\ x \in X_i\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0 \quad .$$

There exist a recursive function r such that $\phi_{r(i)}$ computes $\chi_i * h$. This is proved by applying the S-n-m theorem on the total recursive function $F \in R_2$ defined by

$$F(x,k) = \chi_k(x) * h(x).$$

Next take a complexity measure Φ on this enumeration such that

(i) for $j = r(i)$ we have $\Phi_j(x) = h(x) * (1-\chi_j(x))$

(ii) for $j \neq r(i)$ for all i we have $\Phi_j(x) \geq 3 * h(y) + 1$

This measure exists by Th. 1.2 and the fact that the family $\{\phi_{r(i)}\}_i$ is measured.
The consequence of this choice is that any call of SEARCHTIME(y,list, index,result,bound) is doomed to fail whenever bound $\leq 3 * h(y)$. SEARCHTIME is in fact a "part of an algorithm" and has therefore not a well defined running time; however we can consider SEARCHTIME to be a function (cf the discussion in section 3 before) and then it certainly does not belong to our privileged set of the $\phi_{r(i)}$. Hence we may assume that it costs at least $3 * h(y) + 1$ steps.

It is easy to see that failure in SEARCHTIME occurs always if the old timebound is one of the $\phi_{r(i)}$.
The value of $\phi_{r(i)}(y)$ becomes known at <u>stage</u> y' which y' = y iff $\phi_{r(i)}(y) = h(y)$ and $y' = h(y)$ iff $\phi_{r(i)}(y) = 0$. By our choice for sigma there exists a stagenumber z satisfying $y' \leq z < 3*y'$ such that $\sigma(y') = y$.
At this stage lasttime is made <u>true</u>; however bound is made equal to $\max\{z,\Phi_{r(i)}(y)\} = \max\{z,y'\} \leq 3*y' \leq 3*h(y)$. As we remarked before the call of SEARCHTIME(y,list,index,result,bound) fails whenever bound $\leq 3*h(y)$.

The give up definition is taken to be

$$t'(y) := \max\{t(y), \Phi_t(y)\}.$$

for $t = \phi_{r(i)}$ this results into

$$t'(y) = \max\{\phi_{r(i)}(y), h(y)-\phi_{r(i)}(y)\} = h(y)$$

for $\phi_{r(i)}(y) = 0$ or $\phi_{r(i)}(y) = h(y)$.

Hence $t'(y)$ is made equal to $h(y)$ for every y.
Next we have by the choice of our measure Φ:

a) $C_{h(x)} = \{\phi_{r(i)} | i \in \mathbb{N}\}$

b) $C_t = \emptyset$ if $t = \phi_{r(i)}$ for the computation of $\phi_{r(j)}$ is "expensive" everywhere outside X_j and the support of $\phi_{r(i)}$ consists only of one set X_i.

In a) and b) we can forget about all non privileged algorithms which need at least $3 * h(x) + 1$ steps.

Now for $t = \phi_{r(i)}$ we conclude $C_t = \emptyset$ and $C_{t'} = \{\phi_{r(i)} | i \in \mathbb{N}\} \neq \emptyset$.
This shows that the algorithm does not work.

<u>Remark</u>. This example gives in our own construction still $C_{t'} = C_t$ as we define in our construction $t'(y)$ to be the number up to which SEARCHTIME has arrived. As SEARCHTIME begins with a call MEASURE(1,y,0) which is not going to be finished within $3 * h(y) + 1$ steps we leave SEARCHTIME with 0 stored in result.
So $t = \phi_{r(i)} \Longrightarrow t' \equiv 0$ and $C_{\phi_{r(i)}} = C_0 = \emptyset$.

Hence we modify our example by putting $\phi_{r(i)} = \Phi_{r(i)} = h(x) * \chi_{X_i}$

and now $C_{\phi_{r(i)}} = \{\phi_i\}$ whereas $C_{t'} = \emptyset$ for $t = \phi_{r(i)}$.

<u>The use of a clock making errors</u>:

If one is timing SEARCHTIME by means of some arbitrary complexity measure there is always a function MAXTIME(y,t) such that the assertion of specification F holds, for all situations "with a finite number of exceptions". This restriction ia automatically introduced when using the general argument of two measures bounding each other recursively (Th.1.3). Inspection of the proof learns that exceptions at finitely many values of y do not harm the proof (simply forget about arguments y which are not safe). More exceptions may cause troubles: for example MAXTIME(y,t) is "large enough" $\forall_y \overset{\infty}{\forall}_t$ does not prevent all calamities as one may have a function t such that $\{y,t(y)\}$ is infinitely often an exceptional pair.
In our construction these difficulties need no attention.

<u>The total amount of computing time for t'.</u>

Both proofs in [1] and [2] claim the set of timebounds t' to be measured. They indicate however only that the new timebounds are members of a honest set which is as we have seen in section 2 not sufficient.
A proof of honestness of t' could be given in three phases:

i) Proof that the complexity of <u>stage</u> x is independently of the old timebound t bound by some function G(x). For part I this is clear as the only arguments are either x itself or values t(y) which are computed in x steps in UNIVERSAL and hence may considered to be bounded in terms of x. PART 2 is programmed along fixed lines with a finite amount of freedom (i.e. the contents of old-prior). As the finitely many possibilities of filling in oldprior are recursive in x the maximal computing time of PART 2 is in itself recursive in x. By application of some form of the combining lemma one finds that the complexity of <u>stage</u> x is bounded in terms of x.

ii) Proof that the stage number at which t'(y) is defined is recursive in y and t'(y). This stage number is in fact $\leq$ the first x with sigma(x) = y and x "large enough" to try all values up to t'(y). This is clearly recursive in y and t'(y).

iii) i) and ii) together show that the algorithm is in fact a sequential process consisting of steps of a bounded complexity for which the number of steps is recursively bounded by the argument and the final result. Using a similar trick as in the proof of Lemma (1.4) one may prove that the total complexity of the algorithm is recursively bounded in terms of the argument and the final result.

We shall not work out the proof sketched above as we do not need it. We have shown already in general that a measured set is honest, and our proof gives the measuredness of the t' straightforwards. Furthermore, it is sufficient to prove honesty only for a suitable chosen timebound, for which the assertion is easy to prove by showing i) and ii).

Appendix I Counting steps in ALGOL 60.

The possibility of having an ad hoc clock in SEARCHTIME made it unnecessary to give a more precise definition of "the number of elementary ALGOL statements" executed during a computation.

One might define this notion in different ways. An approach which would give a result anyhow is the following:

Approach A: It is known that the programming language G_3 consisting of instructions of the following types is universal:
1^e) add one to the contents of a specific register
2^e) subtract one of the contents of a specific register
3^e) jump to a specific point of the program.
4^e) simple one branch conditional: if <relation> then <goto label>

This means that every ALGOL program can be translated effectively in a program using only these instructions. Next one might execute the translated program and count the number of instructions executed.

This approach has the disadvantage that one can construct the step counting program from the original program only after compiling it for a very primitive machine. It also has nothing to do with the properties of the language ALGOL 60 we started with.

Another approach is the one which results from using the recursive syntaxis of the program to define the cost of the program in terms of the cost of its more primitive parts.
We give a few indications of things which are going to happen if one follows this approach.

Approach B: The cost of a program derived by syntactical analysis.

Consider first the case of an Arithmetic Expression.

One has the syntaxis:

```
<AE>   ::= <SAE>
<SAE>  ::= <T>
<T>    ::= <F>
<F>    ::= <P>
<P>    ::= <unsigned number>|<variable>|<function designator>|(<AE>)
```

<AE> = <arithmetic expression> . <SAE> = <simple arithmetic expression>
<AO> = <adding operator> . <T> = <term> . <MO><multiplying operator>,
<F> = <factor>, <P> = <primary> etc.

One can define a system of recursive functions $T_{<\alpha>}$ calculating the

cost of an element of a certain syntactical category $<\alpha>$ the pattern would be:

```
T_{<AE>}(<AE>) = if <AE> → <SAE> then T_{<SAE>}(<SAE>)
                 else if <AE> → if <BE> then <SAE> else <AE> then
                 1 + T_{<BE>}(<BE>) + ?
                 else 0
```

one should want to fill in the <SAE> or the <AE> which is executed. Here syntaxis alone fails; we need also the semantics of the program to find out which side is going to be executed:

So put for ?

if <BE> then $T_{<SAE>}$(<SAE>) else $T_{<AE>}$(<AE>)

The next categories are more simple:

```
$T_{<SAE>}(<SAE>)$ = if <SAE> → <T> then $T_{<T>}(<T>)$ else
                     if <SAE> → <AO><T> then $T_{<T>}(<T>)$ + 1 else
                     if <SAE> → <SAE><AO><T> then.
                           $T_{<SAE>}(<SAE>)$ + 1 + $T_{<T>}(<T>)$
     else 0.
```

A similar rule expresses $T_{<T>}$ in $T_{<T>}$ and $T_{<F>}$ and $T_{<F>}$ in $T_{<F>}$ and $T_{<P>}$.

Remains the cost of determining the cost of a primary. The beginning is easy

```
$T_{<P>}(<P>)$ = if <P> → <unsigned number>      then 1
       else if <P> → <variable>             then $T_{<v>}(<variable>)$
       else if <P> → <function designator>  then $T_{<FD>}(<FD>)$
       else if <P> → (<AE>)                 then $T_{<AE>}(<AE>)$
       else 0 .
```

This time there is one direct way out (unsigned number). The variable presents more troubles. We can make

```
$T_{<v>}(<v>)$ = if <v> → <simple variable> then 1 else
           if <v> → <subscripted variable> then
             1 + "cost of evaluating all subscript expressions".
```

The function designator gives even more problems. One should like to put the cost of the execution of the body of the function designator at this phase in the computation. Again the syntax fails to give a straight away definition of the cost. Syntactical analysis of the body of the procedure called by the function designator gives no information whatsoever on how complex the actual parameters are. Further complications arise whenever the actual parameters are formal parameters passed on by other procedures.

Again one has the impression that the calculation of the exact "charges" has to wait until compile and execution time.

Conclusion: Syntactical structure can be used in calculating the cost of a program but syntaxis alone is not sufficient.

Approach C: The minimal charge needed to prevent looping.

In order to have a complexity measure the important demand is that the number of steps goes up to infinity if the program is looping without a result. It is therefore sufficient to tax only those statements which might be used in constructing undefined programs.

Statements which are essential in creating loops are the following:

```
Jumps:                  X : goto X;
for statements:      for x := x while true do .....
                  for x := 1 step 1 until x do .....


procedure statements:        P ;        where
     P is declared:     procedure P;  begin  x := x+1; P end
```

There are also more hidden loop-creators; for example the switch mechanism:

```
          begin  switch S := S[1] ;
               goto S[1]
          end
```

is a program which loops within a single statement. In order to tax it we have to translate the switch declaration into one which gives us space to tax the infinite loop.

Even worse is the example:

```
begin  integer procedure KK(l); value l; label l;
                    begin KK := 1; goto l end;
   switch s := d;
   switch loop := if KK(loop[1]) = 1 then d else d;
                    goto loop[1]; d:
end
```

This example shows the occurrence of infinite recursion without entering a single procedure body:

We next describe a transformation translating a program P into an "equivalent" program T{P} with a built in clock. The clock is introduced by locating at suitable places the procedure statement TAX; (a name which is supposed not to occur in the original program, as is "W" introduced below).
TAX is declared and initialised in a block which is constructed around the whole translated program:

So P is transformed into:

```
   begin integer i, bound;
         integer procedure W(n); integer n;
             begin TAX; W := n end;
         procedure TAX;
                 begin i := i+1; if i > bound then HALT;
         procedure HALT; <body of Halt as you like it>;

         i := 0; bound := IN

         T{P}
   end
```

We use the parentheses { and } as meta-symbols as they do not occur as terminal symbol in ALGOL 60.

The transformation P → T{P} commutes with nearly all direct productions in the syntactical structure of the program; The only places where things are changed are:

i) switch designators contained within a switch declaration

   T{<id>[<AE>]} ::= <id>[W(T{<AE>})].

ii) jumps

   T{goto <DE>} ::= begin TAX; goto T{<DE>} end

iii) procedure declarations:

```
    T{{<type>} procedure <procedure heading> <ST>}} ::=
      {<type>} procedure <procedure heading> begin
               TAX; T{<ST>} end
```

This does not cover the case of a procedure body which is a code program. The best we can do is in this case is illustrated by the example:

```
    T(procedure P(x); <code>;) ::=

    procedure P(x);  begin TAX; PP(x) end;
    procedure PP(x); <code>;
```

It is impossible to find loops which are the result of executing the code body. This is considered illegal tax evasion.

iv) for statements:

```
   T{<for clause><ST>} ::= T{<for clause>} begin TAX; T{<ST>} end;
```

Finally one has T{<basic symbol>} := <basic symbol>.

If one likes one can have also:

```
   T{;}     ::=  ; TAX; if; precedes a statement
   T{begin} ::= begin TAX; if begin is followed by a compound stail
   T{end}   ::=  ; TAX; end
```

The first two transformations are context sensitive.

It is intuitively clear that a syntax analyser can be constructed which performs this operation effectively. The clock is not able to discover loops created by an operating system (for example if division by zero results in a loop).
Another possible extension is the taxing of the pre assignment on variables in the value list of the values of the actual parameters by a method similar to the one used for the switch designators. Although formal parameters in the value list have a known type it is not possible to replace every call Procedure(arg) by a call procedure(value(arg)) if one has applications of the procedure resulting from substitution for formal parameters. See the example.

```
begin integer A; procedure APPLY(P,Q); procedure P;
                           P(Q);
                 procedure x(n); value n; integer n;
                                       A := n;
                 procedure Y(b); value b; label b;
                            goto b;
                 XX :      Apply(x,A); Apply(y,XX);
end
```

This difficulty disappears by making the call by value mechanism explicitely visible.

Appendix II A framed-up test example

The following program was derived from the program in section 3 by replacing the own arrays t, t1, prior and ooldprior by static arrays which are sufficiently large to simulate the algorithm for a number of stages. For UNIVERSAL and MEASURE we constructed ad hoc programs which simulate the computing of timebounds and the measuring of functions in some complexity measure. As can be seen from the programs the simulated complexity measures in UNIVERSAL and MEASURE are distinct. In both measures the result depends on the remainder mod 4 of the index i and the argument x.

The distinct cases are described in the following two diagrams. We always have $xx \equiv x \bmod 4$ and $ii \equiv i \bmod 4$ $0 \leq xx, ii \leq 3$.

UNIVERSAL	xx = 0	xx = 1	xx = 2	xx = 3
ii = 0	x + i	if bound>x+i then x+i else -1	if bound>x^2 then x+i else -1	-1
ii = 1	if bound>x+i then x+i+1 else -1	if bound>x+i then x+i+1 else -1	if bound>x+i then x+i+1 else -1	if bound>x+i then x+i+1 else -1
ii = 2	if bound>x^2 then x+i+2 else -1	if bound>x^2 then x+i+2 else -1	if bound>x^2 then x+i+2 else -1	if bound>x^2 then x+i+2 else -1
ii = 3	-1	i+x+3	if bound>x then x+i+4 else -1	if bound>x then x+i+4 else -1.

MEASURE	xx = 0	xx = 1	xx = 2	xx = 3
ii = 0	0	0	0	0
ii = 1	4*i + x÷2	4*i + x÷2	4*i + x÷2	4*i + x÷2
ii = 2	x^2	x	x	x
ii = 3	∞	0	x	x^2

In the diagram of MEASURE we give the value of $\Phi_i(x)$ for the distinct cases.

From these diagrams one sees that the functions ϕ_i with index $i \equiv 0$ or 1 mod 4 are in the complexity class C_t for all the timebounds computed by Universal. The functions ϕ_i with $i \equiv 2$ or 3 mod 4 violate all timebounds infinitely often and belong therefore not to the complexity class C_t.

The program was tested with i = 8, arg = 200, testvalue = 500, and asking ≡ <u>true</u>. The first 256 stages were executed. The new timebound t1 was computed for all arguments $1 \leq x \leq 128$ except x = 3. We have $t1(1) = 17$, $t1(2) = 11$, $t1(3) = \infty$ and $t1(x) = \Phi_1(x) + 1$ for $x = 4,\ldots,128$.

This result is consistent with the choices of UNIVERSAL and MEASURE. The functions ϕ_i with $i \equiv 0$ mod 4 do never violate the timebound and are therefore always of the black list with the priority that get the moment they are introduced. In SEARCHTIME they have no influence on the value computed for t1(y) as they have runtime 0 for all arguments. The function ϕ_1 does also not violate the timebound. ϕ_5 however violates the timebound for all arguments x which satisfy $x \leq 23$, $x \not\equiv 3$ mod 4. The last violation (x=22) will be discovered at <u>stage</u> 485. However it happens that after <u>stage</u> 26 ϕ_5 is on the black list with priority -55 while there are 15 functions on the black list with higher priority. Each of these functions "prevents" the function ϕ_5 to be removed as long as they are not removed themselves before. Upto <u>stage</u> 256 only 5 of these higher priorities are removed. Higher priority can only be removed at stage x when the obstruction by a still higher priority on the black list has been deleted at the reconstructed <u>stage</u> y where y = sigma(x). This way one can roughly estimate that the stage number where the next "bad priority" gets deleted is two up to three times larger than the most recent removal of a bad priority. This estimate gives that ϕ_5 is not removed from the black list before <u>stage</u> 250.000! This illustrates how long it takes before the moves predicted by the theory actually take place.

The procedures EXIT, read, print, fixt, and printtext are standard procedures in the MILLI-ALGOL 60 system for the EL X8 at the Mathematical Centre in Amsterdam. (See Report LR1.1 Handleiding Milli-systeem voor de EL X8, ed. D. GRUNE).

```
begin    comment meyer mccreight algorithme in framed up test version
            with tracers.;

    integer     x,arg,testvalue,p,i,y    ;
    boolean     test,answer,asking     ;
    integer array     t,t1,prior[1:256] ,ooldprior[1:32896]     ;

    procedure HALT ; EXIT ;

    procedure output(k); value k ; integer k ;
    begin    nlcr; printtext(‹ value t1 computed ›);
         print(y) ;print(k)
    end ;

    procedure outputt(b) ; value b ;boolean b ;
    begin    nlcr ;printtext(‹ answer whether t1[y] ≤ testvalue ›);
        print(y) ;print(testvalue) ;
        if b    then printtext(‹ no violation ›)
            else printtext(‹ violation ›)
    end    ;

    integer procedure in ; in:= read  ;

    boolean procedure inn; inn:= read> 0   ;

    integer procedure maxtime(y,t) ; value y,t ;integer y,t ;
    maxtime:=t+1    ;

    procedure clock(i,l,time) ; value time,l ; integer i,time ;
                            label l ;
    if i ≥ time then goto l else i:=i+1      ;

    integer procedure  UNIVERSAL( i,x,bound ) ;value i,x,bound ;
                            integer i,x,bound ;
    begin    integer ii,xx ;
        ii:= i - i : 4 × 4   ; xx:= x - x : 4 × 4 ;
        UNIVERSAL :=
            if ii= 0 then
            (       if xx= 0 then  x+i
              else  if xx= 1 then
                    (if bound > x+i then x+i else -1 )
              else  if xx= 2 then
                    (if bound > x × x then x+i else -1 )
              else  -1   )
        else if ii=1 then
            (if bound > x then i+x+1 else -1 )
        else if ii=2 then
            (if bound > x × x then i+x+2 else -1 )
        else    if xx= 0 then  -1
          else  if xx= 1 then  i+x+3
          else  if  bound > x then  i+x+4 else -1
    end  UNIVERSAL  ;
```

```
boolean procedure MEASURE ( i,x,y ) ; value i,x,y ;
                           integer i,x,y ;
begin    integer ii,xx ;
    ii:= i - i : 4 × 4 ;xx:= x - x : 4 × 4 ;
    MEASURE := if ii = 0 then true
           else if ii = 1 then  y ≥ 4 × i + x : 2
           else if ii = 2 then
             ( if xx = 0 then y ≥ x × x else y ≥ x )
           else    if  xx = 0 then false
               else if  xx = 1 then true
               else if  xx = 2 then  y ≥ x
               else     y ≥ x × x
end  MEASURE  ;

procedure ORDER BY PRIORITY ( y,list,functions ) ;
    value y ; integer y ;
          integer array list,functions ;
    if y = 1 then
    functions[1]:= sign( list[1])
  else  if y = 2 then
    begin   if abs( list[1]) < abs( list[2])
      then    begin    functions[1]:=sign( list[1]) ;
            functions[2]:=2 × sign( list[2])
        end
      else  begin    functions[2]:=sign( list[1]) ;
            functions[1]:=2 × sign( list[2])
        end
    end
  else  begin    integer z,zz ;
        z:=y : 2 ; zz:= y - z ;
        begin    integer array u,v[1:z] ,uu,vv[1:zz] ;
            integer k,w,ww ;
            for k := 1 step 1 until  z do
                u[k] := list[k]      ;
            for k := 1 step 1 until zz do
                uu[k]:= list[k + z] ;
            ORDER BY PRIORITY (  z, u, v );
            ORDER BY PRIORITY ( zz,uu,vv );
            w:=ww:= 1 ;

            for k:= 1 step 1 until y do
            begin
                if   z< w then goto LL
              else  if zz<ww then goto L
              else  if
          abs( u[abs( v[ w])]) > abs(uu[abs(vv[ww])])
                    then goto LL
              else             goto L      ;

        L  :    functions[k]:= v[ w] ;
            w :=  w + 1 ;
            goto SKIP ;
        LL :    functions[k]:=vv[ww] + sign(vv[ww])×z;
            ww:=ww + 1 ;
        SKIP :
            end
        end
    end  ORDER BY PRIORITY  ;
```

```
procedure SEARCHTIME( y,list,index,result,bound ) ;
        value  y,bound ;integer y,index,result,bound ;
                integer array  list ;
begin    integer array functions[ 1 : y ] ;
    integer  k,value,candidate  ;

    value:= 0 ;
    ORDER BY PRIORITY ( y,list,functions ) ;
    for candidate := 1 step 1 until y do
NEXTMAN:if functions[ candidate ] > 0 then
    begin
    PRODUCER :    for k:= value while
          ¬ MEASURE( functions[candidate],y,value )
                do
           clock( value,TIMEOUT,bound )
    end
        else
    CONSUMER :    if ¬ MEASURE( -functions[candidate],y,value )
        then
    begin   index:= - functions[candidate ] ; result:=value ;
        goto EXIT
    end ;
GIVE UP:clock( value,TIMEOUT,bound ); goto GIVE UP ;
TIMEOUT:index:= -1 ; result:= value ;
EXIT :
end SEARCHTIME    ;

integer procedure sigma(n) ;value n; integer n ;
if n< 3 then sigma:= 1
    else    begin    integer k,l ;
            k:= n; l:= 1 ;
            for k:= k : 2 while k > 0 do l:=l × 2 ;
            sigma:= if l = n then n : 2 else n - 1
        end sigma ;

integer procedure nextprior ; nextprior:= p:= p + 1   ;

procedure PART 1( t,t1,prior ); integer array t,t1,prior ;
begin    integer j,jj,q ;
    nlcr ; printtext(⪡ part 1 ⪢); print( x ) ;
    t[x]:=t1[x]:= -1 ; prior[x]:=nextprior ;
    for j:= 1 step 1 until x do
        if t[j] = -1 then
    begin    q:= t[j]:= UNIVERSAL( i,j,x ) ;
        if q ≠ -1 then
        begin    nlcr ; printtext(⪡value t1 computed⪢) ;
            print( j ) ; print( q ) ;
            for jj:= 1 step 1 until x do
          COMPARE:  if prior[jj] > 0 ∧ ¬ MEASURE( jj,j,q )
              then
            begin    prior[jj]:= -nextprior ;
                nlcr ; print( jj ) ;
                print( prior[jj] ) ;
                printtext(⪡becomes black⪢) ;
            end
        end
    end
end PART 1 ;
```

```
procedure PART 2( t,t1,prior,oldprior ) ;
        integer array t,t1,prior,oldprior ;
begin    nlcr ; printtext(≮part 2≯) ; print(x) ; print(y) ;
    if x = 1 then
  else  if t1[y] < 0 then
    begin    integer index,result,bound ;
        boolean lasttime ;
        lasttime := t[y] ≠ -1 ;
        if lasttime then
        begin    bound := maxtime( y,t[y] ) ;
            if x ≥ bound then bound:=x
        end
                 else
        bound:=x ;

        SEARCHTIME( y,oldprior,index,result,bound ) ;
    OPTION: test:= result ≤ testvalue ;
        if index > 0 then
        begin    if prior[index] < 0 then
            begin    prior[index] :=nextprior ;
                nlcr ; print(index) ;
                print( prior[index]);
                printtext(≮becomes white≯) ;
            end ;
            t1[y] :=result
        end
                else
        if lasttime then  t1[y]:= result ;
        if t1[y] > -1 then
        begin    nlcr ; printtext(≮value t1 computed≯) ;
            print(y) ; print( t1[y] )
        end
    end
end PART 2  ;
```

```
    procedure STAGE ;
    begin    integer k,kk ;
        y:=sigma(x) ; k:=y × (y-1) : 2 ;
        begin    integer array oldprior[1:y] ;
            if x= 1 then oldprior[1]:= 1 else
            for kk:= 1 step 1 until y do
                oldprior[kk]:=ooldprior[kk + k]   ;
            PART 1( t,t1,prior ) ;
            if x > 1 then PART 2( t,t1,prior,oldprior ) ;
        end ;
        if t1[y] > 0 ∧ arg = y then
        begin    output( t1[y] ) ; answer:= t1[y] ≤ testvalue ;
            outputt( answer ) ; HALT
        end
                      else
        if arg = y ∧ asking then
        begin    answer:=test ;
            if ¬ answer then
            begin    outputt( answer ) ; HALT end
        end ;
        k:= x × (x-1) : 2 ; nlcr ;
        for kk:= 1 step 1 until x do
        begin    ooldprior[k + kk]:= prior[kk] ;
            fixt( 4,0,prior[kk] )
        end
    end STAGE  ;


BEGIN OF PROGRAM :    x:=p:= 0 ;
            arg:=in; testvalue:=in; i:=in; asking:=inn ;
            for x:= x+1 while true do STAGE

end COMPLETE PROGRAM

200 , 500 , 8 , 4
```

References

[1] E.M. McCreight, A.R. Meyer : Classes of computable functions defined by bounds on computation: preliminary report. ACM Symposium on the theory of Computing 1969. 78-88.

[2] J. Hartmanis & J.E. Hopcroft : An Overvew of the theory of Computational Complexity. Computer Sci. Tech. Report 70-59. 1970 Cornell University (also J.A.C.M. 18(1971),444-475).

[3] G. Ausiello : Abstract Computational Complexity and Cycling Computations. Journ. Comp. and System Sciences 5 118-128 (1971).

[4] R.L. Constable : The Operator Gap. Preprint Cornell University. (Presented in the conference Record of the IEEE 10-th anval Symposium on Switching and Authomata Theory (SAT) 1969 pp. 20-26.)