THE OPERATOR GAP

Robert L. Constable

Technical Report

69-35

May 1969

Revised October 1969

Department of Computer Science
Cornell University
Ithaca, New York 14850

THE OPERATOR GAP

Robert L. Constable
Cornell University

<u>Abstract</u>

This paper continues investigations pertaining to recursive bounds on computing resources (such as time or memory) and the amount by which these bounds must be increased if new computations are to occur within the new bound. The paper proves that no recursive operator can increase every recursive bound enough to reach new computations. In other words, given any general recursive operator F[ ], there is an arbitarily large recursive t( ) such that between bound t( ) and bound F[t( )]( ) there is a gap in which no new computation runs. This demonstrates that the gap phenomenon first discovered by Borodin for composition is a deeply intrinsic property of computational complexity measures. Moreover, the Operator Gap Theorem proved here is shown to be the strongest possible gap theorem for general recursive operators. The proof involves a priority argument but is sufficiently self-contained that it can easily be read by a wide audience.

The paper also discusses interesting connections between the Operator Gap Theorem and McCreight & Meyer's important result that every complexity class can be named by a function from a measured set.

<u>Introduction</u>

Ever since the inception of research on computational complexity in terms of bounds on available resources, such as time and memory, one of the important problems has been to determine the amount of additional computing resource needed to allow calculation of new functions. The dynamics of the problem are illustrated succintly by Blum's Compression Theorem:

> For every measure of computational complexity there is a recursive function h( ) such that some function f( ) can not be computed within resource bound Φ_i ( ) but can be computed within bound h(Φ_i( )), for Φ_i( ) a sufficiently large complexity function.

In other words, an h( )-composition applied to certain types of resource bounds will allow the calculation of a new function. (Selection of specific small or minimal h( )'s for specific measures is one problem of continuing interest.)

In 1968, Borodin discovered the existence of complexity gaps with respect to composition. That is, for every recursive h( ) there are recursive functions t( ) such that the programs which run within bound h(t( )) are precisely the programs which run within bound t( ). This fundamental discovery raised the important question of whether complexity gaps existed with respect to every recursive operator or whether gaps indicated only the inadequacy of composition as an extension operator.

It was not entirely unreasonable to believe that a recursive operator existed which would bridge the gaps. For instance, Blum had shown that for sufficiently large t( ) there was always a program which <u>infinitely</u> <u>often</u> ran between

$t(\ )$ and $t(t(\ ))$. Thus self-composition might suffice as an extension operator. In addition, it is easy to show that there is a function $\alpha(\ )$ which operates on programs, i, such that new computations run between $\phi_i(\ )$ and $\phi_{\alpha(i)}(\ )$ for all i (where $\phi_i(\ )$ is the function computed by program i). The question then was, "How deep is the gap phenomenon, is it a property of composition or a basic property of measures?"

The operator gap question proved intractable to elementary methods, but a solution was found using a priority argument. That solution is given below.

The technical details will be smoother for readers familiar with Blum [1], but the presentation is reasonably self-contained. We first postulate an effective list of all partial recursive functions, $\{\phi_i(\ )\}$. Our intention is that the list results from a list of all the algorithms, or programs, in a standard formalism for the class of all computable functions, such as the formulation for Turing machines, Register machines, Markov algorithms, Herbrand-Gödel equations, Church-Kleene λ-calculus or the Kleene μ-operator. This intention is formalized by requiring that the list $\{\phi_i(\ )\}$ is <u>acceptable</u>, which means that there is a recursive isomorphism carrying the index for $\{\phi_i(\ )\}$ to an index for an equivalent program in the standard enumeration of Turing machines.

All the usually encountered indexings of the partial recursive functions are acceptable. The reader is referred to

Rogers [6] for a full treatment of this subject and to Rogers [5] p.41 exercise 2-10 for a brief but precise account.

Given the acceptable indexing $\{\phi_i(\ )\}$ of all partial recursive functions of one variable, the list $\Phi = \{\Phi_i(\ )\}$ is a <u>computational</u> <u>complexity</u> <u>measure</u> iff there is a 0,1-valued recursive function $M(\ )$ such that

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

Axiom 2: $M(i,n,m) = 1$ iff $\Phi_i(n) = m$.

If $\Gamma = \{\gamma_i(\ )\}$ satisfies only Axiom 2, then Γ is called a <u>measured</u> <u>set</u>.

Let $I^{\Phi}_{t(\ )} = \{i \mid \Phi_i(n) \le t(n)$ a.e.n (almost all n) & $\Phi_i(\ )$ total $\}$ where f,g,h,t are used to distinguish total functions from partial functions (denoted by ϕ, ψ, μ, Φ, etc.) Let $R^{\Phi}_{t(\ )} = \{\phi_i(\ ) \mid i \in I^{\Phi}_{t(\ )}\}$ and call $R^{\Phi}_{t(\ )}$ a Φ-complexity class. Let $F[\ ]$ denote a <u>general</u> <u>recursive</u> <u>operator</u> taking total functions to total functions. $F[\ ]: \mathcal{F} \to \mathcal{F}$ where $\mathcal{F}$ is the set of number theoretic functions of one argument (see Rogers [5] for a definition of $F[\ ]$). Let $\mathcal{R}$ be the set of recursive functions of one argument.

In [2] Borodin proved the important theorem

<u>Theorem 1</u> (Gap Theorem)

For all Φ and for all recursive $h(\ )$ there exist arbitrarily large increasing $t(\ )$ such that no $\Phi_i(\ )$ satisfies

$$t(n) < \Phi_i(n) < h(t(n)) \text{ i.o.}$$

Proof

Let a( ) be the arbitrary function which t( ) must exceed and let

$$P(y,n) \equiv y > a(n) \ \& \ \forall i \leq n\, [\Phi_i(n) < y \text{ or } h(y) < \Phi_i(n)].$$

Define

$$t(n) = \mu y P(y,n).$$

Since $\Phi_i(n) = m$, $\Phi_i(n) < m$ and $\Phi_i(n) > m$ are all recursive predicates, P( ) is a recursive predicate. Moreover, for all n there is a y such that P(y,n) holds. Thus by Kleene's basic theorem on μ-recursion (see [3] Theorem III p. 279), t( ) is recursive. Clearly t( ) satisfies the conditions of the theorem except that it need not be increasing. To guarantee an increasing t( ) take $P^1(\ )$ for P( ) where

$$P^1(y,n+1) \equiv y > t(n) \ \& \ P(y,n+1).$$

Q.E.D.

Corollary

For all Φ and all h( ) as above, there exist arbitrarily large t( ) such that

$$I^{\Phi}_{t(\ )} = I^{\Phi}_{h}(t(\ )) \quad (\text{thus } R^{\Phi}_{t(\ )} = R^{\Phi}_{h(t(\ ))}).$$

The Gap Theorem shows that there exist arbitrarily large gaps between functions, t( ), h(t( )), where no computation is running. It also shows that it is possible to name complexity classes so that there is no uniform way of extending them by composition on their names.

In [4] McCreight & Meyer prove the deep result that it is possible to rename complexity classes with names from a measured set, Γ, such that for all $t(\ ) \in \mathcal{R}\ \exists g_i(\ ) \in \Gamma$ such that $R_{t(\ )} = R_{g_i(\ )}$. Then by Blum's compression theorem, [1], there is an $h(\ )$ for Γ such that for all i

$$R^{\Phi}_{g_i}(\ ) \subset R^{\Phi}_{h(g_i(\ ))}.$$

So complexity classes can be named so that there is a uniform way to extend them by composition on their names. Let $H[\ ]$ be an operator which assigns a Meyer-McCreight $g_i(\ )$ to $t(\ )$. Then $R^{\Phi}_{t(\ )} \subset R^{\Phi}_{H[t(\ )]}$. Can this $H[\ ]$ be made recursive? That is, is there a recursive operator which will extend every complexity class no matter how named?

Although an index for $g_i(\ )$ can be effectively found from an index for $t(\ )$, we show that no recursive $H(\ )$ exists. This means, among other things, that the Meyer-McCreight procedure must work on indicies for $t(\ )$ rather than on $t(\ )$ itself.

The existence of an "operator gap" shows the extent of the gap phenomenon which Borodin discovered. The proof of the operator gap is interesting in its own right because it is another instance of an intricate priority argument being used in computational complexity theory. The first two examples are by Meyer & McCreight [4] and by Young [7]. In [2] Borodin also uses a simpler priority argument to establish minimum growth rate properties of complexity measures.

<u>Tha Operator Gap</u>

We proceed now to prove

<u>Theorem 2</u> (Operator Gap).

For all Φ and all general recursive operators $F[\ ]$ there are arbitrarily large recursive increasing t( ) such that $t(n) < \Phi_i(n) < F[t(\ )](n)$ i.o. implies $F[t(\ )](m) < \Phi_i(m)$ i.o.

Corollary

For all Φ there is no general recursive operator $F[\ ]$ such that for all $t(\ )$

$$R^{\Phi}_{t(\ )} \subset R^{\Phi}_{F[t(\ )]}.$$

Proof

(1) $F[t(\ )]$ is a total function, $F[t(\ )](n)$ is that function's value at n. If $F[\ ]$ is a recursive operator, then in computing $F[t(\ )](n)$, only finitely many values of $t(\ )$ may be used, say $t(x_{n_1}), t(x_{n_2}), \ldots, t(x_{n_p})$. If in computing $F[t(\ )](n)$ only $x_{n_i} \leq n$ are used, then Borodin's procedure would produce a gap $t(\ )$ for $F[\ ]$. However, x_{n_i} may be larger than n. For example, consider

$F[t(\ )](n) = \sum_{i=0}^{t(t(n))} t(i)$. This situation causes the trouble. What Borodin's procedure does in defining $t(n+1)$ is try $t(n+1) = t(n)+1$, calculate $F[t(\ )](n+1)$, look at $\Phi_i(\ )$ for $i \leq n$ and ask whether $t(n+1) < \Phi_i(n+1) < F[t(\ )](n+1)$. If $\Phi_i(n+1)$ lies in the interval, then "lift $t(n+1)$" above $\Phi_i(n+1)$, i.e., define $t(n+1) = \Phi_i(n+1)+1$ and test the new interval $[t(n+1), F[t(\ )](n+1)]$ for all $\Phi_i(\ )$, $i \leq n$. If another $\Phi_i(\ )$ falls in the interval, then lift $t(\ )$ above it, etc. Since only finitely many $\Phi_i(\ )$ are being tested, eventually the process will halt with a value of $t(n+1)$

satisfying the following condition:

* for all $i \leq n$ either

$\Phi_i(n+1) < t(n+1)$ or

$F[t(\ )](n+1) < \Phi_i(n+1)$.

But now perhaps the value $t(n+2)$ had to be specified in defining $t(n+1)$ and perhaps for all extensions of $t(\ )$.

$t(n+2) < \Phi_i(n+2) < F[t(\ )](n+2)$.

We are no longer free to "lift $t(n+2)$ over $\Phi_i(\ )$" because doing that may lift $F[t(\ )](n+1)$ over a $\Phi_i(\ )$ for $i \leq n$ which was previously above $F[t(\ )]$. That is, "lifting $t(n+2)$ may spoil the gap at $t(n+1)$."

(2) Our strategy for circumventing this problem will be to allow certain $\Phi_i(\ )$'s to spoil the gaps as long as we can eventually arrange $F[t(\ )](n) < \Phi_i(n)$ after Φ_i has spoiled a gap. (Our attempts to arrange getting $F[t(\ )]$ below $\Phi_i(\ )$ will be referred to as "attacks on i".) We shall then have

$t(n) < \Phi_i(n) < F[t(\ )](n)$ i.o. implies $F[t(\ )](m) < \Phi_i(m)$i.o.

The question which arises in implementing the above strategy is when do we let a $\Phi_i(\ )$ spoil a gap. In answering this question, we use a priority argument. We will first describe the $t(\ )$ construction process informally, then in the appendix we give a formal definition.

(3) Informal Construction, preliminaries and initialization

The main construction is an algorithm for extending finite functions defined on initial segments of $\mathbb{N}$. Let $t(\ )|_n$ denote such a function defined on $\{0,1,\dots,n\}$. We consider various ways of temporarily extending $t(\ )|_n$ to a function $\hat{t}(\ )$ so that it can be used to compute $F[\hat{t}(\ )](n)$. Recall that for $F[\ ]$ a general recursive operator, the computation of $F[t(\ )](n)$ requires only finitely many values of $t(\ )$, say $t(x_1),\dots,t(x_{p_n})$. The maximum argument required, x_{p_n} , will be denoted b_n. Say that $[n,\dots,b_n]$ is the <u>foreward region of support</u> for $F[\ ]$ at $t(\ )$ and n. It should thus make sense to write $F[t(\ )|_{b_n}](n)$ if we allowed $F[\ ]$ to act on finite functions. We shall however consider only extensions of $t(\ )|_n$ which are total functions. Three types of extension are used, $\hat{t}(\ )=t(\ )|_n$ min, $\hat{t}(\ ) = t(\ )|_n$ min & y_i and $\hat{t}(\ ) = t(\ )|_n$ F. That is,

(a) minimal increasing extension with respect to the function $a(\ )$: we abbreviate this to <u>minimal extension</u> and denote it by $t(\ )|_n$ min. The definition is, if $\hat{t}(\ ) = t(\ )|_n$ min, then $\hat{t}(m+1) = \max\{a(m+1), \hat{t}(m)+1\}$ for all $m \geq n$.

(b) minimal increasing extension wrt $a(\ )$ and consistent with $t(x_i) = y_i$ for $i=1,\dots,p$; we abbreviate this to <u>minimal consistent extension</u> and denote it by $t(\ )|_n$ min & y_i .

by definition, $\hat{t}(\ )=t(\ )|_n$ min & y_i iff

$$\hat{t}(m+1) = \max\{a(m+1),\ \hat{t}(m)+1,\ y_i \text{ if } m+1=x_i\}$$

$m \geq n$.

(c) increasing F-extension wrt $t(\ )|_n$ min & y_i. Abbreviate this to F-level extension and denote it by $t(\ )|_nF$. Then $t(\ )=\hat{t}(\ )|_nF$ iff $\hat{t}(m+1) = \max\{F[t(\ )|_n \text{ min } \& \ y_i] \ (m+1), \ \hat{t}(m)+1\}$ for all $m \geq n$.

(d) increasing level p+1 F-extension wrt $t(\ )|_n$ min & y_i, denoted $t(\ )|_nF, p+1$. The definition is, $\hat{t}(m+1) = \max \{F[t(\ )|_nF,p] \ (m+1), \ \hat{t}(m)+1\}$ for all $m \geq n$ and where a level 1 extension is an F-extension.

We often write $\hat{t}_p(\ )$ to denote a level p F-extension.

Notice, these extensions are all computable. We use them extensively below, ususally denoting the extension by $\hat{t}(\ )$ and specifying the type by context.

We now describe an initialization procedure to get the finite function, $t(\ )|_n$, which the main procedure extends.

Given the arbitrarily large function $a(\ )$ above which $t(\ )$ is to be constructed, set $t(0) = a(0)+1$. Construct a gap at $t(0)$, i.e., $\phi_o(0) < t(0)$ or $F[\hat{t}(\ )](0) < \phi_o(0)$. Whenever $F[\ ]$ needs a value $t(\ )$, assume the minimal extension, $\hat{t}(\ )$. Let the maximum argument of $\hat{t}(\ )$ called for by $F[\ ]$ at 0 be b_o. The points z such that $0 \leq z \leq b_o$ are called the first foreward region, E_o. $b_o + 1$ is the next free integer.

Having produced a gap at $t(0)$, go to the next free integer and produce a gap there for $\phi_o(\ ), \phi_1(\ )$. Let the largest argument of $t(\ )|b_o$ min needed be b_1. Put $E_1 = \{z | b_o < z \leq b_1\}$, the 2nd foreward region.

What has been done so far is just an <u>initialization</u>. The main procedure begins below with the priority list L empty.

(4) Main Construction

Two algorithms will be given for extending t( ). The first is a direct result of imposing a priority scheme on the process described in part 1 of the theorem, the "gap process." This algorithm requires a nesting scheme and is in some ways more complex than the second. Moreover, proving that the first algorithm terminates involves an interesting excursion into non-constructive proof techniques (part (4) below).

The second algorithm is a modification of the first which unwinds the nesting and explains the non-constructive halting argument.

<u>First Algorithm</u>

Assume that t( ) has gone through n stages of its definition (each stage results in fixing some finite number of values of t( )). So $t(\ )|_{x-1}$ is defined and x is the next free integer.

Step 1: Find all i < x such that

(a) $t(y) < \Phi_i(y) < F[\hat{t}(\ )](y)$ for y < x and $\hat{t}(\ ) = t(\ )|_{x-1}$ min, provided $F[\hat{t}(\ )](y)$ can be computed using t( ) only up to x-1. That is, F[ ] needs only $t(\ )|_{x-1}$, so by an abuse of notation, $F[\hat{t}(\ )](y) = F[t(\ )|_{x-1}](y)$.
Say that <u>y locates i</u>.

(b) y has not previously located i.

(c) i is not already on L.

Order these i's (in their natural order) and put them at the end of the list L (assign them the lowest priorities). For example, if the indicies found are $i_1 < i_2 < i_3$ and L is $\{j_0, j_1, \ldots, j_p\}$ then put $j_{p+1} = i_1, j_{p+2} = i_2, j_{p+3} = i_3$. The indices of highest priority are those with the lowest subscript, i. e. j_0 is highest, j_1 is next, etc.

It is clear that given any $y < b_x + 1$ for any free x that eventually $F[\hat{t}(\ )](y)$ can be computed using $t(\ )|_{z-1}$ for some free z. This is because defining t( ) using minimum consistent extensions and the gap process at free integers defines some total recursive function t( ). Since F[ ] is general recursive, F[t( )](y) must be defined and requires arguments only up to b_y. Take z to be the first free integer beyond b_y.

At the point z if an index $i < z$ has spoiled a gap at y, then z will locate i and place it on a priority list L. Once on the list L, ϕ_i will spoil gaps only for the values less than z and for a certain controlled finite set of values beyond z.

The rest of the construction is concerned with creating gaps and removing indicies from L. An index j leaves L only when $F[t(\ )](z) < \phi_j(z)$. While an index is on L, it will spoil gaps only finitely often. Precisely, <u>if x locates the index i, then i will be allowed to spoil gaps beyond x only when their doing so allows us to remove an index of higher priority from L.</u> (So j_0, the index on L of highest priority, can never spoil a gap beyond its locator after it goes on L.)

Step 2:

Calculate $F[\hat{t}(\ )]\ (x)$ for $\hat{t}(\ ) = t(\ )|_{x-1}$ min. As before let b_x be the maximum argument needed.

When this step is re-entered (from case 2 or case 3 below), $\hat{t}(\ )$ must be taken as $\hat{t}(\ )$ min & y_i. Also let $b_{x,m}$ be the maximum argument needed on the m-th time that this step is entered for the same value x(so $b_x = b_{x,1}$).

Step 3. Examine the interval $I_x = [t(x), F[\hat{t}(\ )]\ (x)]$. Say that an index $j \in L$ is <u>safe</u> iff $\phi_j(x) < t(x)$. Say that an index $j \in L$ is <u>attackable</u> iff

$F[\hat{t}(\ )](x) < \phi_j(x)$. (note, $\phi_j(x)$ might be undefined).

The following situations may arise

(1) all $j \in L$ are safe.

(2) the highest priority unsafe index lies in the interval, I_x.

(3) the highest priority unsafe index is attackable.

In case (1), permanently fix the value of $t(x)$. This ends stage n. The next free integer is x+1. Go to Step 2.

In case (2) lift $t(x)$ over $\phi_i(x)$ for those $j \in L$ which fall in the interval. Go to Step 2.

In case (3) we "pursue an attack on the index." To describe this attack, suppose that L is $\{j_1, j_2, \ldots, j_p\}$. Suppose we are attacking j_e $e > 0$ (if $e = 0$ then freeze values of $\hat{t}(\ )$ up to those needed to define $F[\hat{t}(\ )](x)$, i.e., values of $\hat{t}(\ )$ up to b_x, remove j_o from L and go to Step 1).

Step 4:[+] Set up a procedure, (A), which depends on the parameters X, the point of attack, J, the index being attacked, and B the boundary of the foreward region of support. Start the procedure with x in X, j_e in J and b_x in B. Then

A1: extend t( ) in a minimal manner by enough values that $F[\hat{t}(\ )](z)$ can be calculated for all $z \in E_x$. (This will require values up to some $b^1_{x,v}$, which is b_z for some $z \in E_x$. In general, if procedure (A) is nested to a level s, then arguments up to $b^s_{x,w}$ are required.)

A2: examine $I_z = [\hat{t}(z), F[\hat{t}(\ )](z)]$ for all $z \in E_x$. The following situation may arise.

(i) all higher priority indicies (j_q for $q < e$) are safe for all $z \in E_x$.

[+]

If when checking the interval I_z it is found that certain j_i are attackable at z and others of higher priority are safe at z, it might seem that we should attack these indicies and remove them from L "while we have the chance, that is, dispense with the priorities. But being too eager to remove j_i causes trouble because higher priority indices, say j_0, j_1, may spoil gaps in the foreward region used to remove j_i. We might later add new indices to L below j_0, j_1. Continuing to remove lower indicies might cause j_0, j_1 to spoil gaps infinitely often.

We cannot however ignore the attackable index j_i because all higher priority indices may be "small functions", hence always safe in the future. Thus we must check ahead to see whether the higher priority indices <u>support our attack</u> on j_i (case (i)), or <u>interfere</u> with it (case (ii)) or <u>pre-empt</u> it (case (iii)).

(ii) highest priority unsafe index (of <u>higher</u> priority than j_e) occurs in some interval I_z;

(iii) highest priority unsafe index (of higher priority than j_e) is attackable at some I_z.

A3: In case (i), permanently set all those $\hat{t}(z)$ for $z \in E_x$, remove J from L. The next free integer is B+1. Go next to Step 1.

A4: In case (ii), lift $\hat{t}(z)$ above the $\phi_i(z)$ in question. Recompute $F[\hat{t}(\ )](x)$ using a new minimal extension consistent with the new value at z. Decide whether J is still attackable at x, if it is, then return to X and to A1 (or to Step 2 if X=x). If it is not, then see if any values are being saved from case (ii) below.

If values are being saved, then put them into X,J and B and return to A1. If no values are being saved, than go to Step 2 (it must be the case that X=x and all indicies on L are safe for the current value of $\hat{t}(x)$).

A5: In case (iii), save the old values X,B,J (say on a push down stack), suppose j_q is the highest priority attackable index in E_x and z_o is the least z in E_x at which it is attackable. Then put z_o in X, j_q in J and b_{z_o} in B. Return to A1.

<u>The attack procedure is now nested</u>, so that an exit (ii) from A will return the process to the highest priority index still under attack. An exit from (i) will remove from L the highest priority index successfully attacked and will produce some B+1 as the next free integer.

(4) The above process terminates for all x. Two points are argued. First, there are only finitely many j_i's of higher priority than j_e, so the nesting of attacks will stop if each separate attack stops. But each attack will stop as long as there is a bound on $b^e_{n,w}$. To see that there is such a bound notice that there is a maximum value at each $z \geq x$ beyond which $t(z)$ will never be raised. Because $t(z)$ at some point will be above all $\phi_j(\ )$'s which are defined at z for $j \in L$. Thus for each $z \geq x$, there is a least value at which the "t( ) lifting

process" will stablize. Let $\bar{t}(\ )$ be the function with those stable values. Then $\bar{t}(\ )$ is total. So $F[\bar{t}(\ )](x)$ must be defined since F[ ] is general recursive. There is thus an M such that F[ ] uses only values $\bar{t}(z)$ for $z \leq M$. Therefore, once $\hat{t}(\ )$ has stablized beyond M, $\hat{t}(\ )$ too will need only values $z \leq M$ so that only $b^{*}_{n,w} \leq M$ are required. M is the bound we needed.

<u>Second Algorithm</u>

If instead of using a minimal extension at each stage in the above algorithm we use a level p F-extension where p-1 is the number of indicies on L at the time of extension, then it turns out that the "lifting process" can be eliminated because every index is either safe or can be attacked by using the extension of one lower level. Once the lifting process is elminated, it becomes easy to see that the algorithm terminates. In terms of the above process this means there are no nested attacks and no loops which increase the value b_x to $b_{x,m}$.

We outline below the construction of $t(\ )$ using level p F-extensions. Suppose that the initial segment, $t(\ )|_{x-1}$ of $t(\ )$ has been defined by the initialization desceribed above and suppose that L contains p+1 indicies, $j_0, j_1, \ldots, j_p$. Then $t(\ )$ is being defined at x with L given.

Compute $F[\hat{t}(\ )](x)$ using an ℓ-level F-extension $t_\ell(\ )$ of $t(\ )|_{x-1}$ for $\ell=p+2$. Let the foreward region of support be $[x,d]$.

Check L to see if all indicies are safe at x.

If all are safe, then fix the value of t( ), $t(x)=\hat{t}(x)$.

If all are not safe, then let J be the highest priority unsafe index of L. This index is attackable using the extension $\hat{t}_V(\ )$, for $V=\ell-1$. Let the foreward region of support [X,B], start as [x,d]. (*) Check the region to see whether all higher priority indicies support the attack, i.e., are safe.

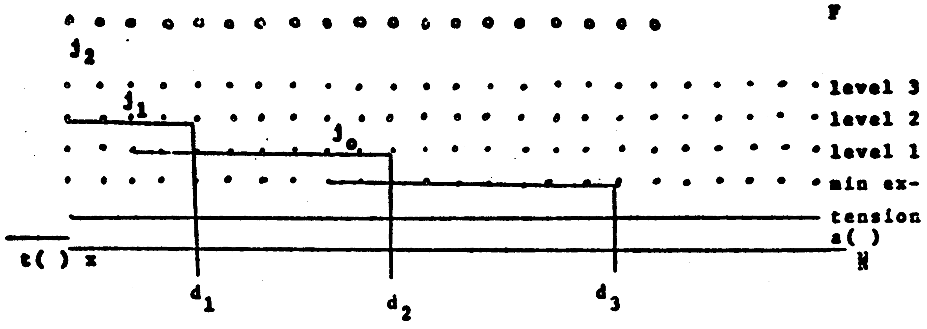
If all higher indicies, support the attack, then fix t( ) to have the value $\hat{t}_V(\ )$ in the region [X,B] and remove J from L.

If some index J', say j_ℓ's for $\ell'<\ell$ pre-empts the attack, then replace J by the index J' replace $\hat{t}_V(\ )$ by $t_{V-1}(\ )$ compute a new forward region of support. Return to * with these new X,V,J and B.

The above process must terminate because at most p indicies can cause an entry to * and so one exit from the * routine must result in the specification of t( ) in some foreward region. The process will terminate with a successful attack because each level k F-extension is defined so that it attacks indicies above the level k+1 F-extension.

We will describe a precise implementation of both algorithms in the appendix.

diagram for level $\ell+1$ extensions in the simple case that F[ ] is increasing



Fixing t( ) to be t( )$|_x$ min on the range $[x, d_3]$ will attack j_o and remove it from L.

(5) Finally we prove that the t( ) defined by the above construction satisfies the conditions of the theorem.

Suppose

* $t(n) < \phi_i(n) < F[t(\ )](n)$ i.o. then at some point x, i goes on the priority list L and * holds for only finitely many n. Suppose moreover that there are p indicies on L of higher priority than i.

$\phi_i(\ )$ will satisfy * for n > x only when higher priority indicies are being removed from L. So at most $c_1+c_2+\ldots+c_p$ times before i itself is of highest priority. Then either i is removed so that

** $\qquad \phi_i(m) < F[t(\ )](m)$

or i never again satisfies *. But * holds i.o. so ** must hold. This means i goes back on L and the above argument is repeated to show that ** must be satisfied every time i goes on L.

Summarizing

$t(m) < \phi_i(m) < F[t(\ )](m)$ i.o.

iff i gets put on L i.o.

iff i gets removed from L i.o.

iff $F[t(\ )](m) < \phi_i(m)$ i.o.

This concludes the proof.

Q.E.D.

The corollary is immediate from the theorem. We can prove another corollary. First define

$F[\ ]$ is an <u>increasing general recursive operator</u> iff $F[\ ]$ is general recursive and for all increasing $t(\ )$, $t(x) < F[t(\ )](x)$.

<u>Corollary</u>

For all ϕ and for all increasing general recursive operators $F[\ ]$ there are arbitrarily large strictly increasing recursive $t(\ )$ such that

$$I^{\phi}_{t(\ )} = I^{\phi}_{F[t(\ )]} \ .$$

<u>Proof</u>

This is immediate from Theorem 1 and the above definition.

Q.E.D.

Conclusion

A strong operator gap theorem would read,

'For all measures Φ and for all general recursive operators $F[\ ]$ there are arbitrarily large recursive $t(\ )$ such that no $\Phi_i(\ )$ satisfies

$t(n) \leq \Phi_i(n) \leq F[t(\ )](n)$ i.o.'

Such a result is impossible as we now show. Let $L^1 = \{ L_i(\ ) \}$ be the tape measure of computational complexity on one-tape Turing machines.

Theorem 3

Let $H[\ ]$ be a general recursive operator such that for all sufficiently large $t(\ )$

(i) if $t(x) \geq x$ for a.e.x, then

$H[t(\ )](x) > t(x)$ a.e.x

(ii) $H[t(\ )](x) \geq t(x+1)$ a.e.x .

Then for all sufficiently large $t(\ )$ there is an i such that

$t(n) \leq L_i(n) \leq H[t(\ )](n)$ i.o.

Proof

Our method of proof is to construct $L_i(\ )$ for a given $t(\ )$ and program t, $\Phi_t(\ ) = t(\ )$.

(1) Define $\Phi_i(0)$ so that $L_i(0) > t(0)$. (Compute $t(0)$ using t, mark off the amount of tape used, $L_t(0)$, if $t(0) < L_t(0)$, then stop, otherwise fill up to $t(0)+1$ squares.)

(2) For input $n+1$ consider two cases:

(a) if $n+1 < L_i(n)$, then define $L_i(n+1) = L_i(n)$. (Mark off $L_i(n)$ amount of tape, test whether $n+1 < L_i(n)$, if yes, then halt. Otherwise, go to case b.)

(b) if $n+1 = L_i(n)$, then as in step (1) define $L_i(n+1)$ to be greater than $t(n+1)$ and greater than $L_i(n)$.

(3) The above steps define $L_i(\ )$. This step will show that $L_i(\ )$ satisfies the theorem, i.e., that

$t(n) \leq L_i(n) \leq H[t(\ )](n)$ i.o.

We consider two parts depending on the properties (i) and (ii) of $H[\ ]$.

(i) To say that $t(\ )$ is sufficiently large means in this case that $t(x) > x$ a.e.x . Thus whenever $n+1 = L_i(n)$, $t(n) \geq n+1 = L_i(n)$. Let j_n = largest $x < L_i(n)$ for which $L_i(x+1) > L_i(x)$. Let z_n = least y such that $j_n \leq y \leq L_i(n)$ & $t(y) \geq L_i(n)$. Then $t(z_n-1) < L_i(z_n-1)$.

(ii) Since $H[t(\ )]\ (z_n-1) \geq t(z_n)$ and $L_i(\ )$ is non-decreasing, it follows that $t(z_n-1) < L_i(z_n-1) \leq L_i(z_n) \leq t(z_n) < H[t(\ )](z_n-1)$. This happens for all n, thus infinitely often.

Q.E.D.

Notice that the operation of self-composition, $H[t(\ )] = t(t(\ ))$ when applied to non-decreasing functions satisfies (i) and (ii) for $t(x) > x$ a.e.x. Thus the operator $F[\ ]$ which first forms $\hat{t}(x) = \sum_{i=0}^{x} t(i)$ and then performs $H[\ ]$ is an example of an operator for which there are no strong gaps.

Blum has observed in [1] p. 335 that for the operation $H[\ ]$ of self-composition there is no strong gap. A proof of this observation is similar to our proof of Theorem 3.

APPENDIX

Here we implement the algorithms of Theorem 2, (4). Think of a "machine" having:

<u>registers</u> capable of holding any $n \in N$,

B: holds upper boundary of foreward region of support

J: holds index under attack

F: holds value of F[ ]

Z,X: hold arguments

<u>Lists</u> which hold lists of integers,

L: list of indices (priority list)

t( ): list of values of t( )

$\hat{t}$( ): list of temporary values of t( ).

a <u>push down stack</u> (first on-last off type of memory list) which holds the intermediate results of recursion. "Poping" the stack means removing the top element. This causes the next element on the list to be the new top element.

Our program will use certain basic subprograms described below.

GET B(X): This program will cause $\hat{t}$( ) to be extended in the minimum manner until F[$\hat{t}$( )](x) can be computed. GET B(X) will return the value of F into register F and the maximum argument of $\hat{t}$( ) needed into register B.

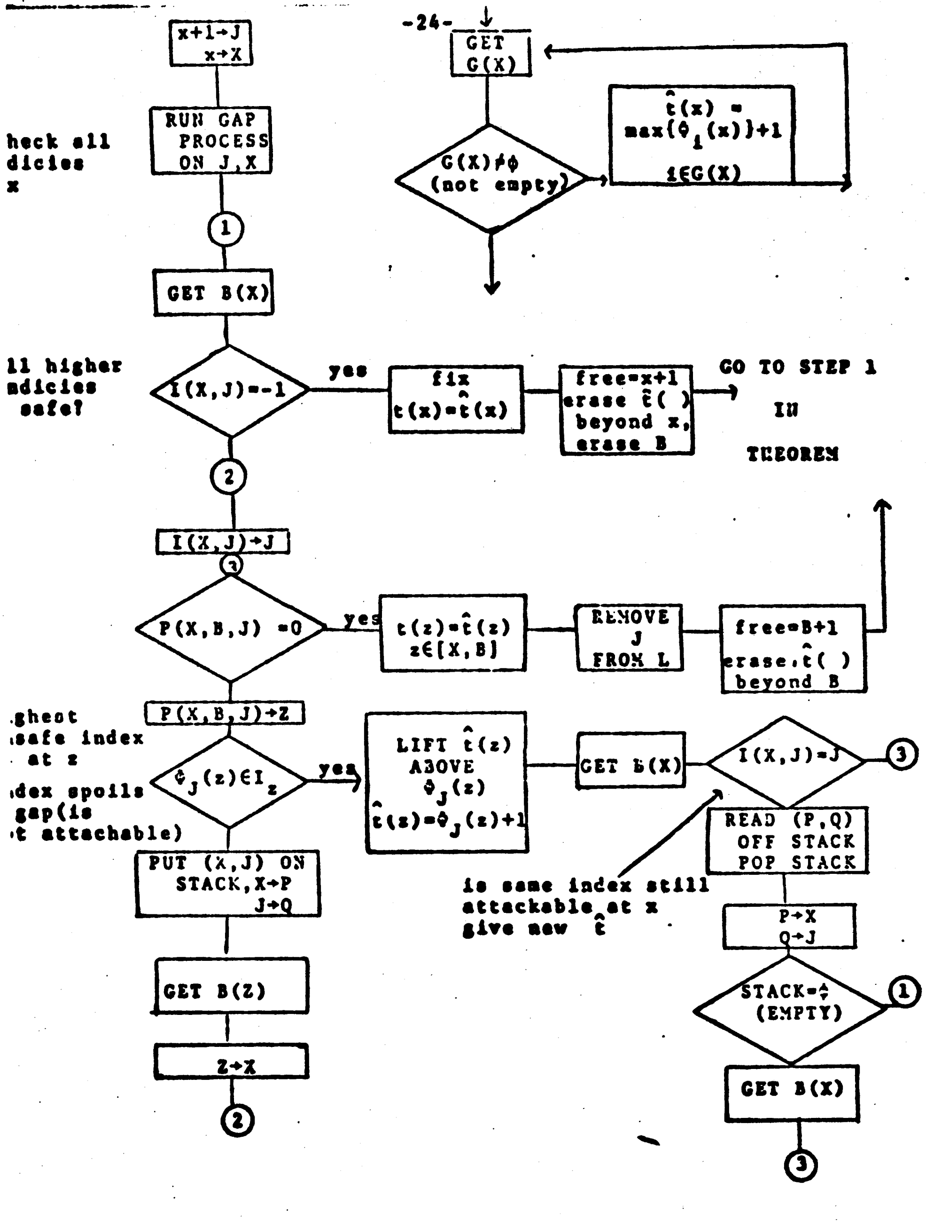
I(X,J) = the highest index on L greater than J which is attackable at X if there is one, otherwise the value-1.

P(X,B,J) = the least integer z>X at which an index higher than J is not safe.

$G(X)$ = list of indicies $\leq X$ which spoil the gap in the interval $I_X = [\hat{t}(x), F[\hat{t}(\ )](x)]$.

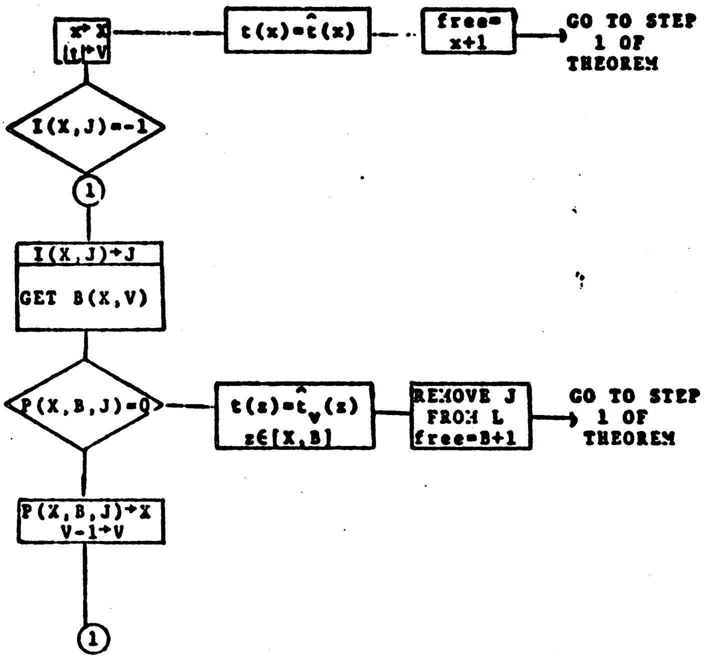
The following flow chart along with step 1 of the informal algorithm of Theorem <u>2</u> give a precise definition of the algorithm.

Suppose that $t(\ )$ is defined up to x and that L has $p+1$ elements, $j_0,\ldots,j_p$.


x+1→J x→X
RUN GAP PROCESS ON J,X
heck all dicies x
1
GET B(X)
ll higher ndicies safe?
I(X,J)=-1
yes
fix $t(x)=\hat{t}(x)$
free=x+1 erase $\hat{t}$( ) beyond x, erase B
GO TO STEP 1 IN THEOREM
2
I(X,J)→J
3
P(X,B,J) =0
yes
$t(z)=\hat{t}(z)$ $z\in[X,B]$
REMOVE J FROM L
free=B+1 erase $\hat{t}$( ) beyond B
P(X,B,J)→Z
ghest safe index at z
dex spoils gap(is t attachable)
$\Phi_J(z)\in I_z$
yes
LIFT $\hat{t}(z)$ ABOVE $\Phi_J(z)$ $\hat{t}(z)=\Phi_J(z)+1$
GET B(X)
I(X,J)=J
3
is same index still attackable at x give new $\hat{t}$
READ (P,Q) OFF STACK POP STACK
P→X Q→J
STACK=ϕ (EMPTY)
1
GET B(X)
3
PUT (X,J) ON STACK, X→P J→Q
GET B(Z)
Z→X
2
GET G(X)
G(X)≠ϕ (not empty)
$\hat{t}(x)=\max\{\Phi_i(x)\}+1$ $i\in G(X)$

By removing the "gap process" block in the first algorithm, we get a simpler algorithm which still defines an F[ ] gap t( ). We use this modified process as the basis of the second algorithm.

Let register V determine the level of the F-extension used. Let the machine have lists $\hat{t}_p(\ )$ p=0,1,2,... to hold the level p F-extensions. Let GET B(X,V) be a routine which calculates the boundary of the foreward region of support for $F[\hat{t}_V(\ )](x)$ and calculates the value of F[ ] (putting the values in registers B and F). Let $|L|$ be the cardinality of the priority list L. Again assume that t( ) is defined up to x. The following is a flow chart for the main part of the second algorithm.


x→X
|t|→V
t(x)=t̂(x)
free=
x+1
GO TO STEP
1 OF
THEOREM
I(X,J)=-1
1
I(X,J)→J
GET B(X,V)
P(X,B,J)=0
t(z)=t̂_V(z)
z∈[X,B]
REMOVE J
FROM L
free=B+1
GO TO STEP
1 OF
THEOREM
P(X,B,J)→X
V-1→V
1

ACKNOWLEDGEMENTS

The author would like to thank Allan Borodin, Patrick Fischer and Michael Fischer for their helpful discussions about the operator gap. P. Fischer first obtained a gap for the self-composition operator using a priority scheme. A.Borodin checked and simplified my original algorithms for general recursive operator gaps and prompted Theorem 3. M.Fischer suggested a smooth termination argument for the first algorithm given here.

REFERENCES

[1] Blum,M. A machine independent theory of computational complexity, JACM, 14 (1967), 322-336.

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

[3] Kleene, S. C. Introduction to Metamathematics Princeton, 1952

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

[5] Rogers, H. Theory of Recursive Functions and Effective Computability, New York, 1967.

[6] ______________ Gödel numberings of partial recursive functions, J. Symb. Logic, 22, #3, 1958, 331-341.

[7] Young, P. Speed-up by changing the order in which sets are enumerated, ACM Symposium on the Theory of Computing, 1969, 89-92.