Unbounded Searching Algorithms

Richard Beigel*
Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218

January 5, 1995

Abstract

The unbounded search problem was posed by Bentley and Yao. It is the problem of finding a key in a linearly ordered unbounded table, with the proviso that the number of comparisons is to be minimized. We show that Bentley and Yao's lower bound is essentially optimal, and we prove some new upper bounds for the unbounded search problem. We show how to solve this problem in parallel as well.

1. Introduction

As posed in [BY76], unbounded search is the problem of searching for a key in a sorted table of unbounded size. The following two-player game is equivalent to the unbounded search problem: Player A chooses an arbitrary positive integer, n. Player B is allowed to ask whether an integer x is less than n. In general the number of questions that B has to ask in order to determine n is a function of n. In this paper, we present lower and upper bounds on the size of this function.

The following theorem from [BY76] is instrumental in providing lower bounds on the number of questions needed in order to solve the unbounded search problem.

Theorem 1 (Bentley and Yao) *If $f(n)$ questions suffice to solve the unbounded search problem then $f(n)$ satisfies Kraft's inequality:*

$$\sum_{n\geq 1} 2^{-f(n)} \leq 1.$$

*Current address: Dept. of Computer Science, 51 Prospect Street, P.O. Box 2158, Yale Station, New Haven, CT 06520-2158. Supported in part by the National Science Foundation under grants CCR-8808949 and CCR-8958528. Work done in part while at Stanford University, supported by a fellowship from the Fannie and John Hertz Foundation.

Bentley and Yao proved this by noting that if $\sigma(n)$ is the sequence of answers given by A in the two-player game above then the function σ determines a prefix code for the positive integers.

We write Z^+ to denote the set of positive integers.

Notation 2 If f is a function from Z^+ to Z^+ then

$$\mathrm{Kr}(f) = \sum_{n \geq 1} 2^{-f(n)}.$$

2. Upper Bounds for Unbounded Search

In this section, we prove a partial converse to Theorem 1.

We say that a real number x between 0 and 1 is *recursive* if there is an algorithm to compute the ith bit of the binary expansion[1] of x. Equivalently, x is recursive if there is a fixed algorithm that takes a rational number y as input and determines if $x < y$. Alternatively, x is recursive if the sequence of bits in the mantissa of x is the characteristic sequence of a recursive set.

We use the word monotone to mean non-decreasing.

Theorem 3 *Let f be a monotone, recursive function such that* $\mathrm{Kr}(f)$ *is recursive. There is an algorithm that solves the unbounded searching problem by asking $f(n)$ questions if and only if* $\mathrm{Kr}(f) \leq 1$.

Proof: Suppose that there is an unbounded searching algorithm that asks $f(n)$ questions. Bentley and Yao's result implies that $\mathrm{Kr}(f) \leq 1$. Conversely, let

$$s(j) = \sum_{1 \leq i \leq j} 2^{-f(i)}.$$

Let

$$g(x) = \min \{j : s(j) > x\}.$$

We determine n by determining $s(n)$. A single question tells us whether $n < g(x)$, and hence whether $s(n) \leq x$ (if $g(x)$ is undefined, then we know without asking any questions that $s(n) \leq x$, so we ask an arbitrary (rhetorical) question). Thus we can determine $s(n)$ by performing a binary search on the half-open interval $(0, 1]$: First we ask if $n < g(1/2)$; based on the answer to that question, we either ask if $n < g(1/4)$ or if $n < g(3/4)$, etc.

After asking $f(n)$ questions, the binary search determines that $s(n)$ belongs to an interval $(a, b]$ where $b - a = 2^{-f(n)}$. Thus

$$0 = 2^{f(n)}b - 2^{f(n)}b \leq 2^{f(n)}b - 2^{f(n)}s(n) < 2^{f(n)}(b - a) = 1.$$

[1] If the binary expansion of x is not unique, then x is rational, and there are algorithms to compute both expansions of x, so we say that x is recursive.

Because

$$2^{f(n)}s(n) = \sum_{1 \leq i \leq n} 2^{f(n)-f(i)}$$

and f is monotone, $2^{f(n)}s(n)$ is an integer. Because b is a division point from the binary-search algorithm, $2^{f(n)}b$ is an integer. Since their difference belongs to $[0, 1)$, the integers $2^{f(n)}s(n)$ and $2^{f(n)}b$ must be equal. Therefore $s(n) = b$. Therefore $s(n+1)$ cannot belong to $(a, b]$. Thus $n = g(s(n)) - 1 = g(b) - 1$, so n is determined by asking exactly $f(n)$ questions. ∎

The existence of an unbounded searching algorithm that asks $f(n)$ questions is equivalent to the existence of an infinite recursive left-finite binary tree such that the nth leaf in the tree is at depth $f(n)$ for all n (see Definition 6 and Proposition 7). For finite trees, the preceding theorem is equivalent to Gilbert's theorem that the optimal alphabetic tree with weights in sorted order is an optimal Huffman tree.

In [Knu81], Knuth proved a result about unbounded searching schemes that ask no irredundant questions, *i.e.*, no questions to which the answer could be deduced from previous answers.

Theorem 4 (Knuth) *Let f be a monotone, recursive function. There is an algorithm that solves the unbounded searching problem by asking $f(n)$ irredundant questions if and only if* $\mathrm{Kr}(f) = 1$.

Since the number 1.0 is recursive, the existence of an unbounded searching algorithm follows from our Theorem 3. However, Knuth's proof is especially beautiful because it produces a closed form for the sequence of questions asked by the unbounded searching algorithm.

Knuth proved the existence of extremely good unbounded searching algorithms by constructing series whose sum is exactly 1 (see [RS88] for some improvements by Reingold and Shen). We, on the other hand, only need to construct series whose sum is less than or equal to 1, provided that we can show that the sum is recursive. Lemma 5 below shows that if we have an effective bound (approaching zero) on the tail of a series, then the sum of that series is recursive. Because proofs of convergence usually provide such an effective bound, Lemma 5 implies that the effective-computability requirement can usually be verified without extra effort.

We write Q^+ to denote the set of positive rational numbers.

Lemma 5 *Let g and h be total recursive functions from Z^+ to* Q^+ *such that*

$$\lim_{n \to \infty} h(n) = 0,$$

and

$$\sum_{i > n} g(i) < h(n).$$

Then $\sum_{i \geq 1} g(i)$ is recursive.

Proof:

$$\begin{aligned}\lim_{n\to\infty}\sum_{i>n} g(i) &\leq \lim_{n\to\infty} h(n)\\ &= 0.\end{aligned}$$

Therefore $\sum_{i\geq 1} g(i)$ converges, so let

$$\sum_{i\geq 1} g(i) = w.$$

We will show how to determine for any rational number y whether $w < y$. If w is rational then we can simply compare the two rational numbers w and y. Otherwise, assume that w is irrational. Therefore $w \neq y$. Since $h(n)$ approaches 0, we can find the least n such that

$$\textstyle\sum_{1\leq i\leq n} g(i) \geq y \quad \text{or} \quad h(n) + \sum_{1\leq i\leq n} g(i) < y.$$

In the first case $w > y$. In the second case $w < y$. ∎

We end this section with a straightforward characterization of unbounded searching algorithms. This characterization will be useful in Section 5..

Definition 6 Let T be a recursive subset of $\{0,1\}^*$. T is a *recursive binary tree* if, for all strings σ,

$$((\sigma 0 \in T) \text{ or } (\sigma 1 \in T)) \Rightarrow (\sigma \in T).$$

T is a *recursive, left-finite binary tree* if, in addition, T has no infinite path except for its rightmost path.

Proposition 7 *There is an unbounded searching algorithm that asks exactly $f(n)$ questions iff there is a recursive left-finite binary tree whose nth leaf from the left has depth $f(n)$ for all n.*

Proof: Suppose that there is an unbounded searching algorithm that asks exactly $f(n)$ questions. As noted in [BY76], it is easy to see that there exists a prefix-free encoding of the positive integers that uses $f(n)$ bits to encode n. Furthermore, that encoding is computable and lexicographic. We let the tree T consist of all strings that encode a positive integer, and all prefixes of such strings. It is easy to see that T has the desired properties.

Conversely, suppose that T is a recursive binary tree whose nth leaf from the left has depth $f(n)$ for all n. If the root of T has only one child, then the unbounded searching algorithm asks a rhetorical question. If the root has two children, then the left child has only finitely many descendants, so the algorithm asks if n is greater than the left child's rightmost descendant. The algorithm chooses the left subtree or the right subtree accordingly and continues recursively. ∎

2.1. Recursion Theoretic Aspects

One might ask whether all of the assumptions in Theorem 3 are necessary. In order to answer this question, it is helpful to know which real numbers can be expressed in the form $\mathrm{Kr}(f)$ for some recursive function f. We identify each real number x in the interval $[0, 1)$ with the set of positive integers whose characteristic sequence is obtained by omitting the decimal point from the binary representation of x. Thus we can talk about a real number being recursive, r.e., etc.

In this section, we assume some familiarity with recursion theory. Let K denote the halting set. We write $A(z) = 1$ if $z \in A$, and $A(z) = 0$ if $z \notin A$. We write $x(z)$ to denote the zth bit in the binary expansion of the real number x.

Proposition 8

- i. *If x is r.e., then there exists a monotone, recursive function f such that $x = \mathrm{Kr}(f)$.*
- ii. *If $\mathrm{Kr}(f) + \mathrm{Kr}(g)$ is recursive, then $\mathrm{Kr}(f)$ is recursive and $\mathrm{Kr}(g)$ is recursive.*
- iii. *If x is nonrecursive and co-r.e., then x is not equal to $\mathrm{Kr}(f)$ for any recursive f.*
- iv. *If f is recursive, then $\mathrm{Kr}(f)$ is truth-table reducible to K.*
- v. *If x is truth-table reducible to K, then there is a monotone, recursive function f such that x is 1-reducible to $\mathrm{Kr}(f)$.*

Proof:

- i. Let A be the r.e. set corresponding to x. If A is finite, then f is easy to construct, so assume that A is infinite. Then there exists a total recursive 1-1 function g whose range is A. We define f by an easy initial segment argument. At stage s we will contribute 2^{-z} to $\mathrm{Kr}(f)$ where z is the sth element of A. Thus $\mathrm{Kr}(f)$ will be the real number corresponding to the characteristic sequence of A.

  Stage 0: Let $n = 1$. Let $k = 1$.

  Stage $s + 1$: Let $z = g(s)$. Let $k = \max(k, z)$. Let $t = 2^{k-z}$. For $n \le i < n+t$, define $f(i) = k$. Finally, let $n = n + t$. (This contributes $t2^{-k} = 2^{-z}$ to $\mathrm{Kr}(f)$.)

  Then, for $z \ge 1$, $\mathrm{Kr}(f)(z) = A(z)$.

- ii. Let

$$\begin{aligned} t_k &= \sum_{1 \le n \le k} 2^{-f(n)}, \\ u_k &= \sum_{1 \le n \le k} 2^{-g(n)}. \end{aligned}$$

Then $\mathrm{Kr}(f)(z) = \lim_{k\to\infty} t_k(z)$, and $\mathrm{Kr}(g)(z) = \lim_{k\to\infty} u_k(z)$. We will compute $\mathrm{Kr}(f)(z)$ and $\mathrm{Kr}(g)(z)$ by choosing k large enough so that the first z bits of t_k and u_k have converged to their limiting value. Let $s_k = t_k + u_k$. Then $\{s_k\}_{k=1,2,\ldots}$ approaches $\mathrm{Kr}(f) + \mathrm{Kr}(g)$ from below. Find k such that the first z bits of the larger binary expansion of s_k agree with the first z bits of the smaller binary expansion of $\mathrm{Kr}(f) + \mathrm{Kr}(g)$. For any $j \geq k$, the first z bits of t_j and u_j must agree with the first z bits of t_k and u_k respectively, for otherwise $s_j > \mathrm{Kr}(f) + \mathrm{Kr}(g)$. Thus $\mathrm{Kr}(f)(z) = t_k(z)$ and $\mathrm{Kr}(g)(z) = u_k(z)$.

iii. By part (i), there is a recursive function g such that $1 - x = \mathrm{Kr}(g)$. If there is a recursive function f such that $x = \mathrm{Kr}(f)$, then $\mathrm{Kr}(f) + \mathrm{Kr}(g) = 1$, which is recursive, so $\mathrm{Kr}(f)$ is recursive by part (ii).

iv. Let $s_k = \sum_{1\leq n\leq k} 2^{-f(n)}$. Let $g(z,k) = s_k(z)$. Then $\mathrm{Kr}(f)(z) = \lim_{k\to\infty} g(z,k)$. Since $g(z,k+1) \neq g(z,k)$ for at most $2^z - 1$ values of k, $\mathrm{Kr}(f)$ is truth-table reducible to K.

v. Since x is truth-table reducible to K, there are recursive functions g and c such that $g(z,0) = 0$, $x(z) = \lim_{s\to\infty} g(z,s)$, and $|\{s : g(z,s) \neq g(z,s+1)\}| \leq c(z)$. We define f by an easy initial segment argument.

Stage -1: Let $b_0 = 1$. Let $k = 1$. Let $n = 1$.

Stage $\langle z, 0\rangle$: Let $t = \lceil \log_2 c(z) + 1 \rceil$. Let $b_{z+1} = b_z + t$.

Stage $\langle z, s+1\rangle$: If $g(z,s+1) = g(z,s)$ then go to the next stage. Otherwise let $w = b_{z+1} - 1$. Let $k = \max(k,w)$. Let $n' = n + 2^{k-w}$. Let $f(m) = k$ for $n \leq m < n'$. Let $n = n'$.

At stage $\langle z, 0\rangle$ we decide to store $x(z)$ in $\mathrm{Kr}(f)(w)$, where $w = b_{z+1} - 1$. Whenever $g(z,s+1) \neq g(z,s)$, we contribute $2^{k-w}2^{-k} = 2^{-w}$ to the value of $\mathrm{Kr}(f)$, so as to complement $\mathrm{Kr}(f)(w)$. But at stage $\langle z+1, 0\rangle$ we carefully set aside a long enough block of unused bits so that there will be no overflow from the location storing the $x(z+1)$ into the location storing $x(z)$. Thus $x(z) = \mathrm{Kr}(f)(b_{z+1} - 1)$.

∎

Although we know a lot about which real numbers are $\mathrm{Kr}(f)$ for some f, it would be intersting to characterize that class of real numbers exactly.

Corollary 9 *There exists a monotone, recursive function f such that $\mathrm{Kr}(f) < 1$, but $\mathrm{Kr}(f)$ is not recursive.*

Proof: Let x be a nonrecursive r.e. real number in part (i) of the proposition above. ∎

Proposition 10 *If there is an algorithm that solves the unbounded searching problem by asking exactly $f(n)$ questions then* *i.* f *is recursive.*

ii. $\mathrm{Kr}(f)$ *is recursive.*

Proof:

i. To compute $f(n)$, run the unbounded searching algorithm on n. $f(n)$ is the number of questions asked.

ii. $\mathrm{Kr}(f)$ is approximated from below by the sequence $\{s_k\}_{k\geq 1}$ where $s_k = \sum_{1\leq n\leq k} 2^{-f(n)}$.

We can also approximate $\mathrm{Kr}(f)$ from above. For $k \geq 1$, let n_k be the number obtained by the unbounded searching algorithm when the first k answers are "yes" and all remaining answers are "no." Then $\sum_{n\geq n_k} 2^{-f(n)} < 2^{-k}$. Let $t_k = \sum_{1\leq n<n_k} 2^{-f(n)} + 2^{-k}$. Then $\mathrm{Kr}(f)$ is approximated from above by $\{t_k\}_{k\geq 1}$.

To compute $\mathrm{Kr}(f)(z)$ find the least k such that the first k bits of the larger binary expansion of s_k agree with the first k bits of the smaller binary expansion of t_k. Then $\mathrm{Kr}(f)(z) = s_k(z)$.

∎

2.2. Are all Conditions Necessary?

We show that the requirement that f be monotone in Theorem 3 is not necessary, but not superfluous.

Proposition 11

i. There is a non-monotone function f *such that there is an unbounded searching algorithm that asks exactly* $f(n)$ *questions.*

ii. There is a non-monotone, recursive function f *such that* $\mathrm{Kr}(f) = 1$, *but there is no unbounded searching algorithm that asks at most* $f(n)$ *questions.*

Proof:

i. Let

$$f(n) = \begin{cases} 3 & \text{if } n = 1 \text{ or } n = 2 \\ 2 & \text{if } n = 3 \\ n-2 & \text{otherwise.} \end{cases}$$

It is readily verified that there is an unbounded searching algorithm that asks exactly $f(n)$ questions.

ii. Let

$$f(n) = \begin{cases} 2 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ n & \text{otherwise.} \end{cases}$$

It is readily verified that there is no unbounded searching algorithm that asks at most $f(n)$ questions, and that $\mathrm{Kr}(f) = 1$.

∎

A corollary to Proposition 10 is the following strengthened version of Theorem 3.

Theorem 3′ *Let f be a monotone function. There is an algorithm that solves the unbounded searching problem by asking exactly $f(n)$ questions if and only if f is recursive,* $\mathrm{Kr}(f) \leq 1$, *and* $\mathrm{Kr}(f)$ *is recursive.*

An important question is whether the assumptions are still necessary if we require merely that the unbounded searching algorithm ask $\leq f(n)$ questions, rather than exactly $f(n)$ questions. We show that in this case, the assumption that f is recursive is not necessary, but not superfluous.

Proposition 12

i. There exists a nonrecursive, monotone function f with $\mathrm{Kr}(f) < 1$ *such that there exists an unbounded searching algorithm that asks fewer than $f(n)$ questions.*

ii. There exists a nonrecursive, monotone function f with $\mathrm{Kr}(f) = 1$ *such that there exists no unbounded searching algorithm that asks at most $f(n)$ questions.*

Proof:

i. Let A be any nonrecursive set. Let g be any recursive function such that there is an unbounded searching algorithm that asks exactly $g(n)$ questions. Let $f(n) = g(n) + p_A(n)$, where $p_A(n)$ is nth smallest element of A. Then $A \leq_T f$ so f is nonrecursive, $g(n) < f(n)$ for all n, and f is monotone because g and p_A are monotone.

ii. Let A be any nonrecursive set. We will construct f such that $A \leq_T f$. Define f as follows: If $n \in A$ then $f(12n - 11) = n + 2$, $f(12n - 10) = n + 3$, and $f(12n - 9) = f(12n - 8) = n + 4$. If $n \notin A$ then $f(12n - 11) = f(12n - 10) = f(12n-9) = f(12n-8) = n+3$. In either case, $f(12n-7) = \cdots = f(12n) = n+4$. Since $\mathrm{Kr}(f) = 1$, there is no unbounded searching algorithm that asks fewer than $f(n)$ questions. Since f is nonrecursive there is no unbounded searching algorithm that asks exactly $f(n)$ questions.

∎

Open Question 13 *Let f be a recursive, monotone function such that* $\mathrm{Kr}(f)$ *is nonrecursive and less than one. When is there an unbounded searching algorithm that asks fewer than $f(n)$ questions?*

This question could be restated as follows: When does there exist a recursive, monotone function g such that f dominates g and $\mathrm{Kr}(g)$ is recursive? Functions satisfying the stated assumptions exist by Proposition 8(v); however the answer is not known for any such function. If the answer is "yes" for all f, then we could simplify Theorem 3 by omitting the requirement that $\mathrm{Kr}(f)$ be recursive, and instead allowing the unbounded searching algorithm to ask $\leq f(n)$ questions. If the answer is "no" for all f, then Theorem 3 is tight even if we allow the unbounded searching algorithm to ask $\leq f(n)$ questions.

3. Some Slowly Converging Series

In this section we show that the upper bounds obtained by Bentley and Yao follow from Theorem 3. We also obtain improved upper bounds. Good upper bounds depend on series that converge slowly; we can define such series in terms of iterated logarithms.

Definition 14

$$\log_b^{(i)}(x) = \begin{cases} x & \text{if } i = 0 \\ \log_b \log_b^{(i-1)}(x) & \text{otherwise.} \end{cases}$$

Definition 15

$$\log_b^* x = \min\{t : \log_b^{(t)}(x) \leq 1\}$$

Definition 16

$$\mathrm{logsum}_b(x) = \sum_{1 \leq i \leq \log_b^* x} \log_b^{(i)}(x)$$

Definition 17

$$\mathrm{logprod}_b(x) = \prod_{0 \leq i < \log_b^* x} \log_b^{(i)}(x)$$

Note that $\mathrm{logsum}_b(x) = \log_b(\mathrm{logprod}_b(x))$.

We will use the following lemma when approximating a sum by an integral.

Lemma 18

$$\frac{1}{\prod_{0 \leq i \leq n} \log_b^{(i)}(x)} = \frac{d}{dx}((\ln b)^{n+1} \log_b^{(n+1)}(x)).$$

Proof: This follows from the chain rule for differentiation. ∎

Definition 19

$$\mathrm{tow}_b(n) = \begin{cases} 1 & \text{if } n = 0 \\ b^{\mathrm{tow}_b(n-1)} & \text{otherwise.} \end{cases}$$

The next lemma will allow us to show that $\sum 1/(\mathrm{logprod}_2(k) c^{\log_2^* k})$ converges when $c > \ln 2$.

Lemma 20 *For every function g,*

$$\sum_{\mathrm{tow}_2(n) < k \leq \mathrm{tow}_2(n+1)} \frac{1}{\mathrm{logprod}_2(k) g(\log_2^* k)} \leq \frac{(\ln 2)^{n+1}}{g(n+1)}.$$

Proof:

$$\begin{aligned}
&\sum_{\mathrm{tow}_2(n) < k \leq \mathrm{tow}_2(n+1)} \frac{1}{\mathrm{logprod}_2(k) g(\log_2^* k)} \\
&= \sum_{\mathrm{tow}_2(n) < k \leq \mathrm{tow}_2(n+1)} \frac{1}{(\prod_{0 \leq j < \log_2^* k} \log_2^{(j)}(k)) g(\log_2^* k)} \\
&\quad \text{by the definition of logprod} \\
&= \sum_{\mathrm{tow}_2(n) < k \leq \mathrm{tow}_2(n+1)} \frac{1}{(\prod_{0 \leq j < n+1} \log_2^{(j)}(k)) g(n+1)} \\
&\quad \text{because } \log^* k = n+1 \\
&\leq \frac{1}{g(n+1)} \int_{\mathrm{tow}_2(n)}^{\mathrm{tow}_2(n+1)} \left(\frac{1}{\prod_{0 \leq j < n+1} \log_2^{(j)} x} \right) dx \\
&\quad \text{by the rectangle rule for sums} \\
&= \frac{1}{g(n+1)} (\ln 2)^{n+1} \log_2^{(n+1)}(x) \Big|_{\mathrm{tow}_2(n)}^{\mathrm{tow}_2(n+1)} \qquad \text{by Lemma 18} \\
&= \frac{1}{g(n+1)} (\ln 2)^{n+1} - 0 \\
&= \frac{(\ln 2)^{n+1}}{g(n+1)}.
\end{aligned}$$

■

Lemma 21

i. Let $c > \ln 2$.

$$\sum_{k > \mathrm{tow}_2(t)} \frac{1}{\mathrm{logprod}_2(k) c^{\log_2^* k}} \leq \left(\frac{\ln 2}{c} \right)^{t+1} \left(\frac{1}{1 - \frac{\ln 2}{c}} \right).$$

ii.

$$\sum_{k \geq 1} \frac{1}{\mathrm{logprod}_2(k)} \leq \frac{1}{1 - \ln 2}.$$

Proof:

i.

$$\begin{aligned}
\sum_{k>\mathrm{tow}_2(t)} \frac{1}{\mathrm{logprod}_2(k)c^{\log_2^* k}} &= \sum_{n\geq t} \sum_{\mathrm{tow}_2(n)<k\leq \mathrm{tow}_2(n+1)} \frac{1}{\mathrm{logprod}_2(k)c^{\log_2^* k}} \\
&\leq \sum_{n\geq t} \left(\frac{\ln 2}{c}\right)^{n+1} \qquad \text{by Lemma 20} \\
&= \left(\frac{\ln 2}{c}\right)^{t+1} \left(\frac{1}{1-\frac{\ln 2}{c}}\right).
\end{aligned}$$

ii.

$$\begin{aligned}
\sum_{k\geq 1} \frac{1}{\mathrm{logprod}_2(k)} &= \frac{1}{\mathrm{logprod}_2(1)} + \sum_{k>1} \frac{1}{\mathrm{logprod}_2(k)} \\
&= 1 + \sum_{k>\mathrm{tow}(0)} \frac{1}{\mathrm{logprod}_2(k)} \\
&\leq 1 + \frac{\ln 2}{1-\ln 2} \qquad \text{by part (i)} \\
&= \frac{1}{1-\ln 2}.
\end{aligned}$$

∎

Theorem 22

i. There is an unbounded searching algorithm that asks exactly $\lceil \mathrm{logsum}_2(n) + 2\rceil$ *questions.*

ii. Let e *be the base of the natural logarithms, let* $0 < \epsilon < e$*, and let*

$$f(n) = \lceil \mathrm{logsum}_2(n) - (\log_2 \log_2 (e-\epsilon)) \log_2^* n\rceil.$$

There is an unbounded searching algorithm that asks $f(n) + O(1)$ *questions.*

Proof:

i. Because $f(n) \geq \mathrm{logsum}_2(n) + 2$,

$$\begin{aligned}
\mathrm{Kr}(f) &\leq \sum_{n \geq 1} \frac{1}{4\,\mathrm{logprod}_2(n)} \\
&\leq \frac{1}{4(1-\ln 2)} \qquad \text{by Lemma 21(ii)} \\
&< 1 \qquad \text{by direct calculation.}
\end{aligned}$$

Lemma 21(i) provides an effective bound on the tail of the sum. Thus Lemma 5 and Theorem 3 imply the existence of an unbounded searching algorithm that asks $f(n)$ questions.

ii. Because $f(n) \geq \mathrm{logsum}_2(n) - (\log_2 \log_2 (e-\epsilon)) \log_2^* n$,

$$\begin{aligned}
\sum_{n > \mathrm{tow}_2(t)} 2^{-f(n)} &\leq \sum_{n > \mathrm{tow}_2(t)} \frac{(\log_2 (e-\epsilon))^{\log_2^* n}}{\mathrm{logprod}_2(n)} \\
&\leq \left(\frac{\ln 2}{\log_{e-\epsilon} 2}\right)^{t+1} \left(\frac{1}{1 - \frac{\ln 2}{\log_{e-\epsilon} 2}}\right) \qquad \text{by Lemma 21(i)} \\
&= (\ln (e-\epsilon))^{t+1} \left(\frac{1}{1 - \ln (e-\epsilon)}\right).
\end{aligned}$$

This provides an effective bound on the tail of the series, so $\mathrm{Kr}(f)$ converges. Let its sum be at most 2^s, where s is a positive integer. Then

$$\sum_{n \geq 1} 2^{-(f(n)+s)} \leq 1$$

and we have an effective bound on the tail of the sum. By Lemma 5 and Theorem 3, there is an unbounded searching algorithm that asks at most $f(n)+s$ questions.

∎

These searching algorithms are comparable to Bentley and Yao's "ultimate" algorithm U [BY76]. Bentley and Yao's algorithm is more practical than ours because theirs was produced constructively. However, it is interesting that we can derive ours directly from Theorem 3 and lemmas about convergence. Our algorithms are also interesting because there is a remark in [BY76], attributed to Chung and Graham, stating that unbounded search could not be accomplished with as few as

$$\mathrm{logsum}_2 n + \log_2 \log_2 (\log^* n) + O(1)$$

questions; we, however, have shown that unbounded search can be accomplished with asymptotically fewer than $\mathrm{logsum}_2 n$ questions. (A related comment appears in [RS88].)

3.1. Other Bases

We can prove similar results using bases different from 2.

Lemma 23 *For all $b \geq 2$,*

$$\sum_{\mathrm{tow}_b(n) < k \leq \mathrm{tow}_b(n+1)} \frac{1}{\mathrm{logprod}_b(k)} \leq b^{-n} + (\ln b)^{n+1}.$$

Proof:

$$\begin{aligned}
&\sum_{\mathrm{tow}_b(n) < k \leq \mathrm{tow}_b(n+1)} \frac{1}{\mathrm{logprod}_b(k)} \\
&= \sum_{\lfloor \mathrm{tow}_b(n) \rfloor < k \leq \lfloor \mathrm{tow}_b(n+1) \rfloor} \frac{1}{\mathrm{logprod}_b(k)} \\
&= \frac{1}{\mathrm{logprod}_b(\lfloor \mathrm{tow}_b(n) \rfloor + 1)} + \sum_{\lfloor \mathrm{tow}_b(n) \rfloor + 1 < k \leq \lfloor \mathrm{tow}_b(n+1) \rfloor} \frac{1}{\mathrm{logprod}_b(k)} \\
&\leq \frac{1}{\mathrm{logprod}_b(\mathrm{tow}_b(n))} + \sum_{\lfloor \mathrm{tow}_b(n) \rfloor + 1 < k \leq \lfloor \mathrm{tow}_b(n+1) \rfloor} \frac{1}{\mathrm{logprod}_b(k)} \\
&\leq \frac{1}{\mathrm{logprod}_b(\mathrm{tow}_b(n))} + \int_{\lfloor \mathrm{tow}_b(n) \rfloor + 1}^{\lfloor \mathrm{tow}_b(n+1) \rfloor} \left(\frac{1}{\mathrm{logprod}_b(x)} \right) dx \\
&\leq \frac{1}{\mathrm{logprod}_b(\mathrm{tow}_b(n))} + \int_{\mathrm{tow}_b(n)}^{\mathrm{tow}_b(n+1)} \left(\frac{1}{\mathrm{logprod}_b(x)} \right) dx \\
&= \frac{1}{\mathrm{logprod}_b(\mathrm{tow}_b(n))} + \left((\ln b)^{n+1} \log_b^{(n+1)}(x) \Big|_{\mathrm{tow}_b(n)}^{\mathrm{tow}_b(n+1)} \right) \qquad \text{by Lemma 18} \\
&= \frac{1}{\mathrm{logprod}_b(\mathrm{tow}_b(n))} + (\ln b)^{n+1} \\
&\leq \frac{1}{\mathrm{tow}_b(n)} + (\ln b)^{n+1} \\
&\leq \frac{1}{b^n} + (\ln b)^{n+1}.
\end{aligned}$$

∎

Corollary 24 *If $2 \leq b < e$ then*

i.

$$\sum_{k > \mathrm{tow}_b(t)} \frac{1}{\mathrm{logprod}_b(k)} \leq \frac{b^{-t}}{1 - \frac{1}{b}} + \frac{(\ln b)^{t+1}}{1 - \ln b}.$$

ii.

$$\sum_{k \geq 1} \frac{1}{\mathrm{logprod}_b(k)} \leq \frac{b}{b-1} + \frac{1}{1 - \ln b}.$$

Proof:

i.

$$\begin{aligned}
\sum_{k>\mathrm{tow}_b(t)} \frac{1}{\mathrm{logprod}_b(k)} &= \sum_{n\geq t} \sum_{\mathrm{tow}_b(n)<k\leq \mathrm{tow}_b(n+1)} \frac{1}{\mathrm{logprod}_b(k)} \\
&\leq \sum_{n\geq t} \left(b^{-n} + (\ln b)^{n+1}\right) \qquad \text{by Lemma 23} \\
&= \frac{b^{-t}}{1-\frac{1}{b}} + \frac{(\ln b)^{t+1}}{1-\ln b}.
\end{aligned}$$

ii.

$$\begin{aligned}
\sum_{k\geq 1} \frac{1}{\mathrm{logprod}_b(k)} &= \frac{1}{\mathrm{logprod}_b(1)} + \sum_{k>1} \frac{1}{\mathrm{logprod}_b(k)} \\
&= 1 + \sum_{k>\mathrm{tow}(0)} \frac{1}{\mathrm{logprod}_b(k)} \\
&\leq 1 + \frac{1}{1-\frac{1}{b}} + \frac{\ln b}{1-\ln b} \qquad \text{by part (i)} \\
&= \frac{b}{b-1} + \frac{1}{1-\ln b} \qquad \text{by arithmetic.}
\end{aligned}$$

∎

Theorem 25 *If $2 < b < e$ then there is an unbounded searching algorithm that asks $f(n) = \lceil (\log_2 b)\mathrm{logsum}_b(n) + \log_2(\frac{b}{b-1} + \frac{1}{1-\ln b})\rceil$ questions.*

Proof:

$$\begin{aligned}
\mathrm{Kr}(f) &\leq \sum_{n\geq 1} \frac{1}{\mathrm{logprod}_b(n)} \frac{1}{\frac{b}{b-1} + \frac{1}{1-\ln b}} \\
&\leq 1 \qquad \text{by Corollary 24(ii).}
\end{aligned}$$

Corollary 24(ii) provides an effective bound on the tail of the sum. Thus Lemma 5 and Theorem 3 imply the existence of an unbounded searching algorithm that asks $f(n)$ questions. ∎

4. Some Slowly Diverging Series

In this section we produce some series that diverge very slowly, in order to prove some strong lower bounds on unbounded search.

Lemma 26 *For all $b \geq 2$,*

$$\sum_{\mathrm{tow}_b(n)<k\leq \mathrm{tow}_b(n+1)} \frac{1}{\mathrm{logprod}_b(k)} \geq \frac{-1}{b^n-1} + (\ln b)^{n+1}.$$

Proof:

$$\begin{aligned}
& \sum_{\mathrm{tow}_b(n)<k\leq \mathrm{tow}_b(n+1)} \frac{1}{\mathrm{logprod}_b(k)} \\
= \quad & \sum_{\lfloor \mathrm{tow}_b(n)\rfloor<k\leq \lfloor \mathrm{tow}_b(n+1)\rfloor} \frac{1}{\mathrm{logprod}_b(k)} \\
= \quad & \frac{-1}{\mathrm{logprod}_b(\lfloor \mathrm{tow}_b(n)\rfloor)} + \sum_{\lfloor \mathrm{tow}_b(n)\rfloor\leq k< \lfloor \mathrm{tow}_b(n+1)\rfloor+1} \frac{1}{\mathrm{logprod}_b(k)} \\
\geq \quad & \frac{-1}{\mathrm{logprod}_b(\lfloor \mathrm{tow}_b(n)\rfloor)} + \int_{\lfloor \mathrm{tow}_b(n)\rfloor}^{\lfloor \mathrm{tow}_b(n+1)\rfloor+1} \left(\frac{1}{\mathrm{logprod}_b(x)}\right) dx \\
\geq \quad & \frac{-1}{\mathrm{logprod}_b(\lfloor \mathrm{tow}_b(n)\rfloor)} + \int_{\mathrm{tow}_b(n)}^{\mathrm{tow}_b(n+1)} \left(\frac{1}{\mathrm{logprod}_b(x)}\right) dx \\
\geq \quad & \frac{-1}{\lfloor \mathrm{tow}_b(n)\rfloor} + \int_{\mathrm{tow}_b(n)}^{\mathrm{tow}_b(n+1)} \left(\frac{1}{\mathrm{logprod}_b(x)}\right) dx \\
\geq \quad & \frac{-1}{\mathrm{tow}_b(n)-1} + \int_{\mathrm{tow}_b(n)}^{\mathrm{tow}_b(n+1)} \left(\frac{1}{\mathrm{logprod}_b(x)}\right) dx \\
\geq \quad & \frac{-1}{b^n-1} + \int_{\mathrm{tow}_b(n)}^{\mathrm{tow}_b(n+1)} \left(\frac{1}{\mathrm{logprod}_b(x)}\right) dx \\
= \quad & \frac{-1}{b^n-1} + \left((\ln b)^{n+1} \log_b^{(n+1)}(x)\Big|_{\mathrm{tow}_b(n)}^{\mathrm{tow}_b(n+1)}\right) \qquad \text{by Lemma 18} \\
= \quad & \frac{-1}{b^n-1} + (\ln b)^{n+1}
\end{aligned}$$

∎

Lemma 27 *If $b \geq e$ then*

$$\sum_{k\geq 1} \frac{1}{\mathrm{logprod}_b(k)}$$

diverges.

Proof:

$$\begin{aligned}
\sum_{k\geq 1} \frac{1}{\mathrm{logprod}_b(k)} \quad & \geq \quad \sum_{n\geq 0} \sum_{\mathrm{tow}_b(n)<k\leq \mathrm{tow}_b(n+1)} \frac{1}{\mathrm{logprod}_b(k)} \\
& \geq \quad \sum_{n\geq 0} \left(\frac{-1}{b^n-1} + (\ln b)^{n+1}\right) \qquad \text{by Lemma 26.}
\end{aligned}$$

That sum diverges to positive infinity because its nth term approaches 1 (for $b = e$) or positive infinity (for $b > e$) in the limit. ∎

Theorem 28 *If $b \geq e$ then there is no unbounded searching algorithm that asks at most*

$$f(n) = (\log_2 b)\mathrm{logsum}_b n + c$$

questions, for any constant c.

Proof: Assume the contrary. Then Theorem 1 implies that $\mathrm{Kr}(f) \leq 1$. However, Lemma 27 implies that $\mathrm{Kr}(f)$ diverges. ▮

The preceding result shows that Theorem 25 is rather tight.

Lemma 29 *If*

$$\sum_{k\geq 1} \frac{1}{h(k)}$$

diverges, then

$$\sum_{k\geq 1} \frac{1}{\mathrm{logprod}_2(k)(\ln(2))^{\log_2^*(k)} h(\log_2^*(k))}$$

diverges.

Proof:

$$\begin{aligned}
&\sum_{k\geq 1} \frac{1}{\mathrm{logprod}_2(k)(\ln 2)^{\log_2^*(k)} h(\log_2^*(k))} \\
&\quad = \sum_{n\geq 0} \sum_{\mathrm{tow}_2(n)<k\leq \mathrm{tow}_2(n+1)} \frac{1}{\mathrm{logprod}_2(k)(\ln 2)^{\log_2^*(k)} h(\log_2^*(k))} \\
&\quad = \sum_{n\geq 0} \sum_{\mathrm{tow}_2(n)<k\leq \mathrm{tow}_2(n+1)} \frac{1}{\mathrm{logprod}_2(k)(\ln 2)^{n+1} h(n+1)} \\
&\quad = \sum_{n\geq 0} \frac{1}{(\ln 2)^{n+1} h(n+1)} \sum_{\mathrm{tow}_2(n)<k\leq \mathrm{tow}_2(n+1)} \frac{1}{\mathrm{logprod}_2(k)} \\
&\quad \geq \sum_{n\geq 0} \frac{1}{(\ln 2)^{n+1} h(n+1)} \left(\frac{-1}{2^n - 1} + (\ln 2)^{n+1}\right) \qquad \text{by Lemma 26.} \\
&\quad = \sum_{n\geq 0} \frac{1}{h(n+1)} \left(\frac{-(\log_2 e)^{n+1}}{2^n - 1} + 1\right).
\end{aligned}$$

For large n the nth term of the sum approaches $\frac{1}{h(n+1)}$. Therefore the sum diverges. ▮

Corollary 30

$$\sum_{k\geq 1} \frac{1}{\mathrm{logprod}_2(k)(\ln(2))^{\log_2^*(k)}}$$

diverges.

Proof: Let $h(n) = 1$ in Lemma 29. ∎

The following theorem improves a lower bound from [BY76].

Theorem 31 *There is no unbounded searching algorithm that asks at most $f(n)$ questions, where*

$$f(n) = \mathrm{logsum}_2 n - (\log_2 \log_2 e) \log_2^* n + c,$$

for any constant c.

Proof: Assume the contrary. Then Theorem 1 implies that $\mathrm{Kr}(f) \leq 1$. However, Lemma 26 implies that $\mathrm{Kr}(f)$ diverges. ∎

The lower bound from the preceding Theorem and the upper bound from Theorem 22(ii) differ by less than any positive constant times $\log_2^* n$. In [Knu81], Knuth has obtained bounds that differ by less than $(\log_2^*)^{(m)} n$ for any m; therefore at least one of Knuth's bounds is superior to ours. His bounds involve iterates of the function $\lfloor \log_2 n \rfloor$; it would be interesting to know if both of Knuth's bounds are superior to ours. In the next section we will see how to produce algorithms that are as close as practically possible to being optimal.

5. Nearly Optimal Algorithms

The algorithms presented by ourselves and by others are very close to optimal. In this section, we will show that given any algorithm for unbounded search, there is an asymptotically better algorithm. In contrast, a result of Raoult and Vuillemin [RV79] says that there exist algorithms that are as close as we like to optimal.

There is an unbounded searching algorithm with $f(n) = n$. Because

$$\sum_{n \geq 1} 2^{-n} = 1,$$

that algorithm can not be improved for any value of n without making it worse for some other value of n. However, we have already seen algorithms that are asymptotically much better. If we are willing to ask a larger number of questions in order to determine the first few numbers, then we can improve that algorithm asymptotically by an additive constant c. Furthermore, given any unbounded searching algorithm that asks exactly $f(n)$ questions, where f is monotone, we can easily improve that algorithm asymptotically by an additive constant c. Choose t such that

$$\sum_{n > t} 2^{-f(n)} \leq 2^{-(c+1)},$$

choose m such that $t 2^{-m} \leq \frac{1}{2}$, and let

$$g(n) = \begin{cases} m & \text{if } 1 \leq n \leq t \\ \max(f(n) - c, m) & \text{if } n > t. \end{cases}$$

The function g is monotone, because f is monotone. Furthermore

$$\begin{aligned}
\mathrm{Kr}(g) &= \sum_{1 \le n \le t} 2^{-g(n)} + \sum_{n>t} 2^{-g(n)} \\
&\le \frac{1}{2} + \sum_{n>t} 2^{-g(n)} \\
&\le \frac{1}{2} + \sum_{n>t} 2^{-(f(n)-c)} \\
&\le \frac{1}{2} + 2^c 2^{-(c+1)} \\
&= 1.
\end{aligned}$$

We note that $\mathrm{Kr}(g)$ is recursive because $\mathrm{Kr}(f)$ is recursive and $f(n)-g(n)$ is constant for large n. An effective bound on the tail of the sum is easily found. Therefore we can perform unbounded search by asking exactly $g(n)$ questions.

In fact, we can improve every unbounded searching algorithm asymptotically by more than a constant.

Theorem 32 *Suppose that we can perform unbounded search by asking exactly $f(n)$ questions. Then there exists a function g such that we can perform unbounded search by asking $g(n)$ questions and such that*

$$\lim_{n\to\infty} f(n) - g(n) = \infty.$$

If f were monotone, then we could prove this result in the same way as the preceding result. However, we do not assume that f is monotone. We prove the result via tree rotations.

Proof: By Proposition 7, let T be a recursive left-finite binary tree whose nth leaf has depth $f(n)$. Let v be the first node on T's infinite path that has exactly four grandchildren. Let a be the left child of v, and let b and c be the right grandchildren of a. We rotate the subtree rooted at v as follows: a and b become the left grandchildren of v, and c becomes the right child of v. This has the effect of lowering finitely many leaves, but raising infinitely many leaves. We repeat the process recursively on c. Let $g(n)$ be the depth of the nth leaf in the recursive tree produced in this way. It easy to verify that g has the desired properties. ∎

The next result shows that there exist unbounded searching algorithms that differ from the elusive optimum by less than any given recursive function g, provided that g grows towards infinity. Although it was originally proved by Raoult and Vuillemin, we include the proof for the sake of completeness.

Theorem 33 [RV79] *Let $g(n)$ be any monotone, recursive function such that*

$$\lim_{n\to\infty} g(n) = \infty.$$

Then there is a function f such that there is an unbounded searching algorithm that asks exactly $f(n)$ questions, but there is no unbounded searching algorithm that asks at most $f(n) - g(n) + c$ questions, for any constant c.

Proof: We construct $h = f - g$ via an easy initial segment argument.

Stage 0: Let $t_1 = 1$.

Stage $s \geq 1$: Choose $k_s > k_{s-1}$ so that $g(t_s + 2^{k_s}) \geq s + 1$. Let $t_{s+1} = t_s + 2^{k_s}$. Let $h(n) = k_s + s - g(n)$ for $t_s \leq n < t_{s+1}$.

By construction, $h(n) + g(n) = k_s + s$ for $t_s \leq n < t_{s+1}$, and $k_{s+1} > k_s$. Therefore $h + g$ is monotone. In addition, since $t_{s+1} - t_s = 2^{k_s}$,

$$\sum_{t_s \leq n < t_{s+1}} 2^{-(h(n)+g(n))} = 2^{k_s} 2^{-(k_s+s)} = 2^{-s}.$$

Summing over s, we find that $\mathrm{Kr}(h + g) = 1$. Consequently, there is an unbounded searching algorithm that asks exactly $h(n) + g(n)$ questions.

Also by construction, $h(n) \leq k_s$ for $t_s \leq n < t_{s+1}$. Therefore,

$$\sum_{t_s \leq n < t_{s+1}} 2^{-h(n)} \geq 1,$$

so $\mathrm{Kr}(h) = \infty$. Consequently, there is no unbounded searching algorithm that asks at most $h(n) + c$ questions for any constant c. ∎

6. Parallel Unbounded Search

Bentley and Yao suggest a variant of unbounded search where we are allowed to ask p simultaneous questions, *i.e.*, in one round we are allowed to ask "Is $n \geq t_1$?", "Is $n \geq t_2$?", ..., "Is $n \geq t_p$?". We call such questions higher-lower questions. How many rounds of higher-lower questions are needed in order to determine n? How many rounds of arbitrary questions are needed in order to determine n? The following solutions were suggested by Gasarch and Kruskal.

Theorem 1 generalizes easily to provide a lower bound.

Theorem 34 *If $f(n)$ rounds of p questions suffice to solve the unbounded search problem then*

$$\sum_{n \geq 1} (p + 1)^{-f(n)} \leq 1.$$

Proof: As Bentley and Yao pointed out, there are only $p+1$ different possibilities for the result of a round of p questions of the form "Is $n \geq t$?". Thus the results of all $f(n)$ rounds of questions form a prefix code for n in base p (see Bentley and Yao's proof when $p = 2$). The conclusion follows from Kraft's Theorem [Gal68]. ∎

Theorem 3 generalizes easily to provide a corresponding upper bound.

Theorem 35 *Let f be a monotone, recursive function such that*

$$\sum_{n\geq 1} (p+1)^{-f(n)}$$

is recursive and is less than or equal to 1. Then there is an algorithm that solves the unbounded searching problem by asking $f(n)$ rounds of p questions.

Proof: Same as the proof of Theorem 3, except that we use $(p+1)$-ary search [Kru83] instead of binary search. ∎

The two theorems above allow us to generalize our earlier results by simply replacing $f(n)$ by $f(n)/\log_2(p+1)$.

- $\lceil(\log_{p+1} 2)(\text{logsum}_2(n) - (\log_2 \log_2(e-\epsilon)) \log_2^* n)\rceil$ rounds of p simultaneous higher-lower questions are sufficient for unbounded search.
- $(\log_{p+1} 2)(\text{logsum}_2 n - (\log_2 \log_2 e) \log_2^* n) + O(1)$ rounds of p simultaneous higher-lower questions are not sufficient for unbounded search.
- Given any computable function g that grows without bound, there exists an unbounded searching algorithm (asking rounds of p simultaneous higher-lower questions) that is within $g(n)$ rounds of being optimal.

If we are allowed to ask rounds of p simultaneous questions, with no restriction on the form of the individual questions then we may guess the number n after asking significantly fewer questions. In fact, if there is a recursive binary prefix code of length $f(n)$ for the positive integers then we can guess n after asking $f(n)/p$ questions: Our ith question is "What are bits $(i-1)p+1$ through ip of the binary prefix code for n?" This bound is tight because the sequence of answers to all rounds of questions forms a recursive binary prefix code for n. This yields the following results:

- $\lceil\frac{1}{p}(\text{logsum}_2(n) - (\log_2 \log_2(e-\epsilon)) \log_2^* n)\rceil$ rounds of p simultaneous questions are sufficient for unbounded search.
- $\frac{1}{p}(\text{logsum}_2 n - (\log_2 \log_2 e) \log_2^* n) + O(1)$ rounds of p simultaneous questions are not sufficient for unbounded search.
- Given any computable function g that grows without bound, there exists an unbounded searching algorithm (asking rounds of p simultaneous questions) that is within $g(n)$ rounds of being optimal.

7. Conclusions

If we are trying to guess a positive integer, then we can determine that the number is n by asking

$$f(n) = \lceil \text{logsum}_2(n) - (\log_2 \log_2 (e-\epsilon)) \log_2^* n \rceil$$

questions of the form "Is n at least t?". However we cannot determine that the number is n by asking only

$$\mathrm{logsum}_2 n - (\log_2 \log_2 e) \log_2^* n + O(1)$$

questions of that form. Similarly tight bounds can be obtained for parallel unbounded search.

8. Acknowledgements

This work was motivated by a question posed by Bill Gasarch in [Gas85] and solved in [BG89]. Some of the ideas used in this paper are due to Bob Floyd's course on complexity theory, Stanford, 1983. I would also like to thank Bill Gasarch and Clyde Kruskal for helpful comments, especially on parallel unbounded search; Rao Kosaraju for pointing out the relationship to alphabetic coding; and Dan Willard for referring me to Bentley and Yao's paper.

References

[BG89] Richard Beigel and William I. Gasarch. On the complexity of finding the chromatic number of a recursive graph II: The unbounded case. *Annals of Pure and Applied Logic*, 45(3):227–246, December 1989.

[BY76] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *IPL*, 5(3):82–87, August 1976.

[Gal68] P. E. Gallager. *Information Theory and Reliable Communication.* Wiley, New York, 1968.

[Gas85] W. I. Gasarch. A hierarchy of functions with applications to recursive graph theory. Technical Report 1651, University of Maryland, Dept. of Computer Science, 1985.

[Knu81] Donald E. Knuth. Supernatural numbers. In David A. Klarner, editor, *The Mathematical Gardner*, pages 310–325. Wadsworth International, Belmont, California, 1981.

[Kru83] Clyde P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, C-32:942–946, October 1983.

[RS88] Edward M. Reingold and Xiao-Jun Shen. More nearly optimal algorithms for unbounded searching. Technical Report UIUCDCS-R-88-1471, University of Illinois at Urbana-Champaign, Dept. of Computer Science, November 1988.

[RV79] J.-C. Raoult and J. Vuillemin. Optimal unbounded search strategies. Technical Report 33, Laboratoire de Recherche en Informatique, Université de Paris-Sud, Orsay, France, 1979.