LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-297

A RANDOMIZED DATA STRUCTURE FOR ORDERED SETS

JON L. BENTLEY
F. THOMSON LEIGHTON
MARGARET LEPLEY
DONALD F. STANAT
J. MICHAEL STEELE

MAY 1986

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Randomized Data Structure for Ordered Sets

Jon L. Bentley[1]
F. Thomson Leighton[2]
Margaret Lepley[3]
Donald F. Stanat[4]
J. Michael Steele[5]

[1]AT&T Bell Labs
Murray Hill, NJ 07974

[2]Mathematics Department and
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

[3]Hughes Aircraft Corporation
Playa Del Rey, CA 90293

[4]Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

[5]Department of Mechanical Engineering
Princeton University
Princeton, NJ 08540

Abstract: In this paper, we consider a simple randomized data structure for representing ordered sets, and give a precise combinatorial analysis of the time required to perform various operations. In addition to a practical data structure, this work provides new and nontrivial probabilistic lower bounds and an instance of a practical problem whose randomized complexity is provably less than its deterministic complexity.

This research was supported in part by ONR Contract N00014-76-C-0370, NSF Grant MCS-78-07736 and an NSF Presidential Young Investigator Award with matching funds from Xerox and IBM. Parts of this paper were presented at the 19th and 20th Annual Allerton Conferences on Communication, Control and Computing [2, 6].

1. Introduction

In this paper, we consider the problem of maintaining a set from a totally ordered domain under the operations Member, Insert, Delete, Predecessor, Successor, Maximum and Minimum. The basic data structure that we use to represent such a set is a sorted linked list implemented by the two arrays Link $[0 \ldots N]$ and Value $[1 \ldots N]$ where Link [0] points to the first element in the list. Among other things, we will show that Member, Insert, Predecessor, Successor and Maximum can all be accomplished in $2\sqrt{N} - c$ expected steps where c is a small constant, and that this bound is optimal within the constraints imposed by the data structure. We also show that Delete requires just $4\sqrt{N} - 2c$ expected steps and that Minimum requires just one step (Link[0] points to the minimum element). All of these bounds (except for Minimum) are dramatically better than the worst case bound of N steps.

Although quite simple, the randomized linked list data structure is surprisingly efficient. In fact, it is superior to all of those described by Knuth [5] for certain applications. The salient attributes of such applications are listed below.

- Space is important. This structure uses only one extra word of storage per element, while binary search trees use at least two extra words, and various hashing schemes use varying amounts of extra storage. However, the storage for this structure must be available in a single contiguous block.
- The "orderedness" operations of Successor, Predecessor, Minimum and Maximum are frequent; these are not possible in most hashing schemes.
- Insertions and deletions are frequent. If the data structure changes rarely, binary search in a sorted array is very efficient.
- Program simplicity is important. Each operation on this structure requires only about a dozen lines of code, while some operations on balanced binary search trees require over one hundred lines of code.
- Run time is important for problems of medium size (where medium means that N is between, say, 100 and 10,000). If N is below that range, simple sequential strategies are probably efficient enough. If N is above that range, then the logarithmic search time of binary search will be necessary for many applications. When N is in the medium range, though, the low constant factors of this structure will make it competitive with binary search trees.

Of course, the simple linked list is one of the most basic and well-known data structures, and has arisen in countless contexts. Most relevant to this paper is the prior work of Janko [3, 4] who studied randomized algorithms for sorting using linked lists and obtained an $O(N^{3/2})$ bound on the expected time needed to sort N items.

The remainder of the paper is divided into sections as follows. For the most part, we concentrate our efforts on the analysis of a simple algorithm for Member Search. This is because the algorithm for Member Search can be easily transformed into an efficient algorithm for each of the other operations. In Section 2, we define the problem more precisely and observe that the worst-case complexity of performing a Member Search is linear in N. In Section 3, we describe the randomized data structure and explain its relationship to a simple Guess-Decrement game. We also describe an optimal randomized algorithm for Member Search and show how it can be extended to form efficient algorithms for Insert, Delete and the other operations. Section 4 considers some natural extensions of the basic model and contains some additional probabilistic analysis.

2. The Problem and its Deterministic Complexity

The data structure we will use is a sorted linked list implemented in contiguous storage by the two arrays Link[$0 \ldots N$] and Value[$1 \ldots N$]. The pointer Link[0] points to the first element of the list, Value[Link[0]]. The next element can be found in Value[Link[Link[0]]], and so forth. The end of the list is denoted by an element whose Link field contains -1. Furthermore, we will insist that the array is *dense:* Value[$1 \ldots N$] must contain N elements of the represented set. The sortedness of the linked list implies that if Link[I] is not -1, then Value[I] $\leq$ Value[Link[I]]. We will often refer to Value[I] and Link[I] together as node I. Figure 1 illustrates the array representation of the sorted linked list <2.6, 3.1, 4.1, 5.3, 5.8, 5.9, 9.7>.

I	0	1	2	3	4	5	6	7
Value[I]		3.1	4.1	5.9	2.6	5.3	5.8	9.7
Link[I]	4	2	5	7	1	6	3	−1

Figure 1: *An array representation of the sorted linked list <2.6, 3.1, 4.1, 5.3, 5.8, 5.9, 9.7 >.*

It is clear that performing a Member Search in such an array requires accessing at most N elements of the array (either by following Link fields through the list or simply by interating through Value fields of the array). We will now show that in the worst case, this much time is necessary to locate whether a given element is in the list. We will assume that a (deterministic) search algorithm is composed of operations of the following types, each with unit cost. (Note that if operations of type 2 have no cost, then binary search can be used to solve the problem in logarithmic time.)

1. Determine the index of the node at the head of the list (by accessing Link[0]). There is one operation of this type.
2. Determine the successor of node I for $1 \leq I \leq N$ (by accessing Link[I]). There are N operations of this type.
3. Determine the Value of node I, for $1 \leq I \leq N$ (by accessing Value [I]). There are N operations of this type.

Our model assumes that a protagonist specifies a sequence of the above operations while an

adversary ensures that N operations will be required. We will assume that the adversary knows the value of the key the protagonist seeks, which we will call V, and that other key values may be assigned arbitrarily by the adversary. We will describe a strategy that enables the adversary to delay returning V until the protagonist has specified a sequence of N operations. Without loss of generality, we will assume that whenever one of Value[I] or Link[I] is asked, both of Value[I] and Link[I] are provided at a cost of a single step ($1 \leq I \leq N$). The value of V will be the maximum element in the list. There are two cases depending on whether or not the protagonist asks the type 1 question.

Case 1: The type 1 question is not asked.

The adversary always answers questions so that the protagonist has queried a contiguous subset of the ordered list. In particular, assume that the protagonist asks about node I where I has not yet been queried. (Remember that Value[I] and Link[I] are always provided together, at a total cost of one.) If $I =$ Link[J] where J is the node at the head of the contiguous subset of previously queried nodes, then the adversary assigns the largest value yet given (but less than V, of course) to Value[I] and assigns an as yet unqueried node to Link[I]. If $I \neq$ Link[J], then the adversary assigns the smallest value yet given to Value[I] and sets Link[I] $= K$ where K is the smallest node in the contiguous subset of previously queried nodes. In either case, the set of queried nodes continues to form a contiguous subset of the ordered list, with all values less then V. The argument continues in this fashion until $N - 1$ nodes have been queried. At this point, the remaining unqueried node is $I =$ Link[J] where J is the largest queried node. In order to resolve Member Search for V, the protagonist must still ask the value of node I, making for a total of N queries overall.

Case 2: The type 1 question is asked.

In this case, we will show that $N - 1$ nodes must be queried, making for a total of N steps. The argument proceeds as before until the protagonist asks the type 1 question. In response, the adversary reveals that Link[0] $= K$ where K is the smallest node in the contiguous subset of previously queried nodes. From this point on, the adversary will answer questions so that the queried nodes form at most two contiguous subsets of the ordered list, one of them beginning with Link[0]. The subset beginning with Link[0] will always have values smaller than the other contiguous subset, and all values will be less than V. The details of the adversary's responses are similar to before until a total of $N - 2$ node queries have been made. At this point the protagonist must still query the value of node I where $I =$ Link[J] and J is the node with the largest value seen so far. Hence $N - 1$ queries need to be made, accounting for N operations overall.

3. Randomized Algorithms

In what follows, we focus on algorithms that allow probabilistic access to nodes in addition to deterministic and Link access. Although the worst case performance of such randomized algorithms is no different than that for deterministic algorithms, we will find that the average case performance

is much better.

The section is divided into subsections as follows. In Section 3.1, we define a class of simple randomized algorithms for Member Search. We model the performance of algorithms in this class with a Guess-Decrement game in Section 3.2. In Section 3.3, we use the game model to show that the expected running time for the optimal Member Search Algorithm is $2\sqrt{N} - c$, where c is a small constant. We extend the algorithm for Member Search to other operations in Section 3.4.

3.1. A Class of Randomized Algorithms for Member Search

By combining probabilistic access with access by predecessor link, a wide range of algorithms can be considered. For example, the following pseudo-Pascal program searches for the element E, using the order of operations specified by the array Step. When Step[J] is zero, a random sample occurs. Otherwise the program follows the next link in the list. Note that when a random sample is chosen, the position in the list is updated only if the random position is closer to (but not at or beyond) the location of E. This strategy ensures that the updated position in the list never worsens and that when E is eventually found, its predecessor will also have been found (since E will have been reached via a link). (This particular code assumes that Value[0] is $-\infty$ and Value[-1] is ∞.)

```
P := 0
J := 0
do until exit
    J := J + 1
    if Step[J]= 0 then do
            R :=Random(1,N)
            if Value[R]< E and Value[R]>Value[P] then P := R
        else do
            if Value [Link[P]]=E then exit (*E is at Link[P]*)
            if Value [Link[P]]> E then exit (*E is not in the list*)
            if Value [Link[P]]< E then P :=Link[P]
```

For any specified Step array, the expected performance of the associated algorithm will depend on the value of E being searched. For example, if E is less than or equal to the smallest item in the list, then the algorithm will terminate on or before the first Link access. For the time being, we will focus on the more interesting case when E is bigger than the largest item in the list. This is, in fact, the worst case for any step array in terms of expected running time, and is representative of the case when E has a random rank. For expediency, we will defer the proof of these assertions to Section 4, where we consider search values with arbitrary index.

3.2. Modelling Algorithms as Strategies

Before proceeding to construct an optimal algorithm from the class described in Section 3.1, it is useful to associate algorithms in the class with strategies for a simple probabilistic "Guessing-Decrement" game. The *G-D Game* involves two integers, i and N. The value of N remains fixed throughout the game, and the value of i is originally N. The goal of the player is to reduce the value of i to zero in the minimum expected number of steps. A step consists of performing one of the following two operations:

- D (for Decrement): If $i > 0$, then replace i by $i - 1$.
- G (for Guess): Choose j to be an integer uniform from $1 \ldots N$ and replace i by j if $j < i$. The value of i is unknown to the player, except at the beginning of the game when $i = N$, and at the end when he is notified that i has reached zero.

The value of i represents the distance from the current element in the linked list—denoted by P in the above code—to the end of the list. The value N is the number of items in the list. We start with $i = N$ because we assume that we are searching for the largest element in the list. In the general case, we would start with i equal to the rank of E, if known.

Each Guess corresponds to a random access in the above code, whereas each Decrement corresponds to a link access. A strategy or sequence of operations will be denoted by a character string σ composed of G's and D's to be performed in order from left to right. A sequence of G's and D's corresponds naturally to a Step array. A sequence is said to be *complete* if it contains at least N D's. Note that operations written after the first N D's are superfluous and need not appear. For convenience, however, we will often end complete sequences with D^N.

A complete sequence will always reach $i = 0$ and terminate the game after some number of steps t. The expected termination point for a complete sequence σ is denoted by

$$\mathrm{E}(\sigma) = \sum_{j=1}^{\infty} j \Pr[t = j] = \sum_{j=0}^{\infty} \mathrm{Q}_j(\sigma)$$

where $\mathrm{Q}_j(\sigma) = \Pr[t > j]$. The object is to minimize $\mathrm{E}(\sigma)$ over all complete sequences σ. We denote the minimum by $S(N)$.

Note that $\mathrm{E}(\sigma)$ also denotes the expected running time of the Member Search algorithm for the corresponding Step array. Hence, determining the optimal σ is equivalent to determining the optimal algorithm from the class described in Section 3.1. For simplicity, we will use the G-D notation henceforth.

3.3. An Optimal Strategy

The task of finding an optimal strategy for the G-D game will proceed in two steps. The first step consists of finding the best strategy from among those of the form G^kD^N for some k. The second, and more difficult, step consists of showing that there is an optimal strategy having this form.

We commence by analyzing strategies of the form G^kD^N. Within this restricted class, it is easy to determine the best value of k and the minimum of $E(G^kD^N)$.

Theorem 1: *For any integer* $k \geq 0$, $E(G^kD^N) \geq E(G^rD^N)$ *where* $r = \sqrt{N} - 1 - 1/(24\sqrt{N}) + O(1/N)$.

Proof: From the definition,

$$E(G^kD^N) = \sum_{j=0}^{k+N} Q_j(G^kD^N) .$$

Since the first k operations are Guesses the game cannot end there, so $Q_j(G^kD^N) = 1$ for $j = 0, \ldots, k$. The probability of not terminating after the first d Decrements is $(N-d)^kN^{-k}$, since all the Guesses must be larger than d in order not to terminate. Therefore

$$\begin{aligned} E(G^kD^N) &= k + 1 + \sum_{d+1}^{N-1} (N-d)^k N^{-k} \\ &= k + 1 + N^{-k} \sum_{d=1}^{N-1} d^k \\ &= k + 1 + N/(k+1) - 1/2 + k/(12N) + O(k^2/N^2) \end{aligned}$$

The minimum occurs when

$$1 - N/(k+1)^2 + 1/(12N) + O(k/N^2) = 0$$

and thus when

$$k = \sqrt{N} - 1 - 1/(24\sqrt{N}) + O(1/N) . \blacksquare$$

Theorem 1 provides an upper bound of $S(N) \leq 2\sqrt{N} - 1/2 + 1/(12\sqrt{N}) + O(1/N)$ expected operations for the G-D game. Of course, the optimal value of $k = \sqrt{N} - 1 - 1/(24\sqrt{N}) + O(1/N)$ may have to be rounded to a neighboring integer, so we should conclude only that $S(N) \leq 2\sqrt{N} - c$ where c is a small constant that tends to $1/2$ as N grows large. In what follows, we will show that this bound is tight by proving that there is an optimal strategy of the form G^kD^N. We commence with some definitions and lemmas.

When a sequence ω is not complete, the value of i after the operations in ω have been performed may remain undetermined. Instead of knowing the exact value of i we define a probability vector

$$P_j(\omega) = \Pr[i > j \text{ after executing the sequence } \omega] .$$

We can see from the definition that $P_j(\omega) \geq P_{j+1}(\omega)$. Moreover the vector can be computed for any sequence ω.

Lemma 1: *For any sequence* $\omega = G^{a_1}D^{b_1}\ldots G^{a_s}D^{b_s}$, *with* $b = b_1+\cdots+b_s$ *and* $a = a_1+\cdots+a_s$

$$P_j(\omega) = \begin{cases} (N-j-b)^{a_1}(N-j-b+b_1)^{a_2}\ldots(N-j-b_s)^{a_s}N^{-a} & \text{for } j < N-b \\ 0 & \text{for } j \geq N-b \end{cases}$$

Proof: During each block of Guesses, G^{a_m}, the value i must remain above j plus the number of D's which are still be to be performed, $b_m+\cdots+b_s$. The probability that all the Guesses in the block are between $j+b_m+\cdots+b_s+1$ and N is $(N-j-b_m-\cdots-b_s)N^{-a_m}$. ■

These probabilities are important in determining the optimal strategy. The following lemma states one of the most useful properties of this vector.

Lemma 2: *For any sequence* ω, $P_j(\omega D)/P_0(\omega D) \leq P_j(\omega)/P_0(\omega)$ *for* $0 \leq j \leq N$.

Proof: Each ratio is a product of terms of the form

$$[(N-j-b_i-\cdots-b_s)/(N-b_i-\cdots-b_s)]^{a_i} .$$

The sequence ωD has one more D in the last block than ω, so b_s increases by one in ωD. When $P_0(\omega D) > 0$, a comparison of these terms, letting $K = N-b_i-\cdots-b_s$, reveals that

$$[(K-j-1)/(K-1)]^{a_i} < [(K-j)/K]^{a_i}$$

and thus that $P_j(\omega D)/P_0(\omega D) \leq P_j(\omega)/P_0(\omega D)$. If $P_0(\omega D) = 0$, we define the ratio to be zero and the inequality still holds. ■

Remember that our goal is to prove that the optimal strategy has the form G^kD^N. To do this, we next analyze the effect of minor variations in strategy on the expected number of operations. Then we will show that if a small variation improves the strategy, then a larger change could mean even more improvement. The two sequences which we will compare first are $\sigma = \omega DG^kD^N$ and $\sigma^* = \omega G^kD^N$. The only difference between σ and σ^* is the position of the block G^k. The following lemma gives a method for comparing these two strings.

Lemma 3: $E(\omega DG^kD^N) \leq E(\omega G^kD^N)$ *if and only if* $V_k(\omega) \leq 1$, *where*

$$V_k(\omega) = \begin{cases} P_1(\omega)/P_0(\omega) + \sum_{d=1}^{N-1} P_d(\omega)/P_0(\omega)\left[(N-d+1)^k-(N-d)^k\right]k^{-1}N^{-k} & \text{if } P_0(\omega) > 0 \\ 0 & \text{if } P_0(\omega) = 0 \end{cases}$$

Proof: Let σ and σ^* be defined as above. We would like to know when $E(\sigma) - E(\sigma^*) \leq 0$. The following values for $Q_m(\sigma)$ and $Q_m(\sigma^*)$ can be easily verified.

$$Q_m(\sigma) = \begin{cases} Q_m(\sigma^*) & \text{if } m \leq |\omega| \\ P_1(\omega) & \text{if } |\omega|+1 \leq m \leq |\omega|+k+1 \\ P_{m-|\omega|-k}(\omega)(N-m+|\omega|+k+1)^k N^{-k} & \text{if } m > |\omega|+k+1 \end{cases}$$

$$Q_m(\sigma^*) = \begin{cases} Q_m(\sigma) & \text{if } m \leq |\omega| \\ P_0(\omega) & \text{if } |\omega|+1 \leq m \leq |\omega|+k \\ P_{m-|\omega|-k}(\omega)(N-m+|\omega|+k)^k N^{-k} & \text{if } m > |\omega|+k \end{cases}$$

Thus $E(\sigma) - E(\sigma^*) = \sum_{m=0}^{\infty} [Q_m(\sigma) - Q_m(\sigma^*)] \leq 0$ is equivalent to

$$(k+1)P_1(\omega) - kP_0(\omega) + \sum_{d=1}^{N-1} P_d(\omega) \left[(N-d+1)^k - (N-d)^k\right] N^{-k} - P_1(\omega) \leq 0 .$$

Rearranging terms slightly gives

$$P_1(\omega) + \sum_{d=1}^{N-1} P_d(\omega) \left[(N-d+1)^k - (N-d)^k\right] N^{-k} k^{-1} \leq P_0(\omega) . \quad \blacksquare$$

Combining Lemmas 2 and 3 enables us to extend a minor variation of the string into a more radical change. Specifically, if the last black of G's is more efficient when it is moved to the right one place, then it is best to remove the block of Guesses altogether.

Lemma 4: *For all ω, if $E(\omega D G^k D^N) \leq E(\omega G^k D^N)$, then $E(\omega D^N) \leq E(\omega D G^k D^N)$.*

Proof: If $E(\omega D^j G^k D^N) \leq E(\omega D^{j-1} G^k D^N)$ for some $j \geq 1$, then $V_k(\omega D^{j-1}) \leq 1$ by Lemma 3. By Lemma 2 and the definition of $V_k(\omega)$, we know that $V_k(\omega D^j) \leq V_k(\omega D^{j-1})$, and thus that $V_k(\omega D^j) \leq 1$. Thus, we can conclude by Lemma 3 that $E(\omega D^{j+1} G^k D^N) \leq E(\omega D^j G^k D^N)$. The proof of the lemma is completed by applying this process inductively. ■

It is now a simple matter to prove our main result.

Theorem 2: *For every starting sequence ω, there exists an integer $r \geq 0$ such that $E(\omega G^r D^N) \leq E(\omega\sigma)$ for every completed sequence $\omega\sigma$.*

Proof: Let σ denote the shortest (in length) sequence for which $\omega\sigma$ is complete and $E(\omega\sigma) = \min_\gamma E(\omega\gamma)$. If σ is of the form $G^r D^N$, then we are done. Otherwise $\sigma = \omega^* D G^k D^j$, where G^k is the last block of Guesses and $j > 0$. By the optimality of σ, we know that $E(\omega\omega^* D G^k D^j) \leq E(\omega\omega^* G^k D^{j+1})$. Thus by Lemma 4, we know that $E(\omega\omega^* D^{j+1}) \leq E(\omega\omega^* D G^k D^j)$ which contradicts the minimality of σ. ■

Thus the best way to finish any initial sequence is by a block of Guesses followed by Decrements. By letting ω be the empty string, we find that the optimal strategy for the game is $G^k D^N$. Recalling

Theorem 1, we find that the optimal value is near $\sqrt{N} - 1 - 1/(24\sqrt{N}) + O(1/N)$ and thus $S(N) = 2\sqrt{N} - c$ where c is a constant that tends to $1/2$ as N gets large. Hence the expected number of steps for Member Search is at most $2\sqrt{N}$. As a consequence, it is not difficult to show that this is within one or two steps of optimal whenever we are searching for an item that is bigger than the median. Searches for items less than the maximum are discussed more thoroughly in Section 4.

3.4. Algorithms for the Other Operations

Thus far we have considered only the problem of searching the linked list to determine if it contains a given element. It is easy to perform many other set operations on this structure. The following list summarizes those operations, and describes their costs in terms of the number of Value elements accessed.

- **Member:** The previous sections studied the problem of searching to determine whether a given element is a member of the set represented by the linked list. *Cost:* $2\sqrt{N}$.
- **Insert:** A new element can be inserted in the list by using Member Search. *Cost:* $2\sqrt{N}$.
- **Delete:** The first step in deleting an element is to find that element by a search algorithm, and then modify the Link field of its predecessor to point to its successor. This takes $2\sqrt{N}$ references to the Value array. The next step must patch the "hole" created in the dense array by moving the last element of the array to the vacant position. Searching for the last element requires $2\sqrt{N}$ references. *Cost:* $4\sqrt{N}$.
- **Predecessor:** The element immediately preceding a given element can be found by a simple modification to the Member Search algorithm. *Cost:* $2\sqrt{N}$.
- **Successor:** The element immediately succeeding a given element can be found by a simple modification to the Member Search algorithm. *Cost:* $2\sqrt{N}$.
- **Minimum:** The minimum element in the set is pointed to by Link[0]. *Cost: 1.*
- **Maximum:** The maximum element in the set can be found by searching for infinity. *Cost:* $2\sqrt{N}$.

Each of the above operations is straightforward to implement given the Member Search algorithm and basic techniques for dealing with data structures for searching (described, for example, by Knuth [5]). Furthermore, the simplicity of the code implies that the constant factors in the running time of the program will be relatively small. The only deviation that a programmer should make from the Member Search algorithm deals with the random number generation: since many random number generators are very expensive, it might be preferable to use some other approach to sample the k elements.

4. Extensions and Remarks

At first glance, it might appear that our proof technique depends on not being able to Guess 0. This is not the case. In fact, when Guesses of 0 are allowed, the optimal strategy differs from the above strategy by only $O(1/\sqrt{N})$ Guesses.

In the G-D game, the value i in the box can be interpreted as the number of links between our present position in the list and the position of the element we are searching for. When searching for the last element we start at $i = N$. In Member search we are searching for an element whose position is unknown and randomly distributed. Therefore we should start at an unknown, random i, or equivalently start at $i = N$ and do one Guess to randomize i before counting operations. By Theorem 2 the optimum strategy is then $\mathrm{G}^{r-1}\mathrm{D}^N$ as opposed to $\mathrm{G}^r\mathrm{D}^N$ when searching for the last element.

Sometimes the jth element in the list is needed. When j is small it is easy to follow links and when $j = N$ we can use the G-D strategy. Between these two extremes, the strategies used thus far are not necessarily valid since the value of the jth element is unknown. But if we know the value as well as the position of the element for which we are searching, then we can apply the above techniques to find an optimal G-D search time. Searching for an element at position j means starting the G-D game at $i = j$. This is equivalent to starting at $i = N$ and doing $N - j$ Decrements. By Theorem 2 the best way to continue from this point is $\mathrm{G}^k\mathrm{D}^N$ for some k. Thus it is only necessary to compute the optimal number of Guesses, $k = r(j)$. By modifying Theorem 1 it can be shown that $r(j) = c(j)\sqrt{N}$ where

$$c(j) = \begin{cases} 0 & \text{if } j \le \sqrt{2N} \\ (j-1)^{-1}\sqrt{N}[e^{(j-1)c(j)\sqrt{N}}(1-c(j)^2) - 1] - O(1/\sqrt{N}) & \text{if } \sqrt{2N} < j \le O(\sqrt{N}) \\ 1 - O(1/\sqrt{N}) & \text{if } j >> \Omega(\sqrt{N}) \ . \end{cases}$$

It is easy to determine $c(j)$ numerically and a graph of the function (e.g., see Figure 2) shows that (once non-zero) $c(j)$ approaches 1 exponentially fast.

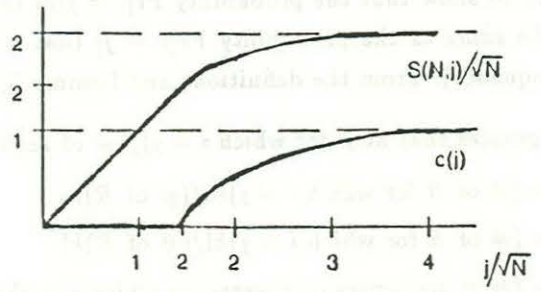

Figure 2: *Graph of $c(j)$ and $S(N,j)/\sqrt{N}$ where $S(N,j)$ is the optimal expected number of operations when searching for the j-th element.*

The G-D game can be modified in other ways. One particularly interesting modification allows the player to use the information about when a random sample is successful (i.e., closer to the target). For example, suppose the black box containing i is connected to a light which turns on every time i is decreased. At first glance, it appears as though such information could be quite useful in planning when to stop Guessing and start Decrementing. For instance, if the light flashed on for a series of early Guesses, then the player might be led to believe that the early Guesses were very good and thus that i had become very small. Hence, the player might think it wise to start Decrementing early. This is *not* the case, however, since if all the Guesses are required to be distinct, then it can be shown that the light does not add any useful information at all. It is worth remarking that the likelihood of two Guesses being identical is small and thus the constraint that all the Guesses be different has a negligible effect on the final result.

Such a counter-intuitive result requires some justification. First notice that when all the Guesses are distinct, the sequence of guesses is just a permutation of a subset R of $\{1,\ldots,N\}$. Every sequence of guesses produces a unique light sequence, β, but one light sequence can be produced by many different guess sequences. In particular,

Lemma 5: *For every light sequence β of length k there exists an integer m, such that for any set of guesses $R \subseteq \{1,\ldots,N\}$ of size k there are exactly m permutations of R which have light sequence β.*

Proof: Let $K = \{1,\ldots,k\}$ and set m to be the number of permutations of K which produce the light sequence β. Now consider any other subset R of length k. There is a $1-1$ order-preserving mapping between K and R, so there is also a $1-1$ mapping between the permutations of the two sets which preserves light sequences. This means that there are also m permutations of R which fit β. ■

We can now prove

Theorem 3: *When all the Guesses in the G-D game are distinct, then the light sequence adds no extra information about the value of i after a sequence of Guesses.*

Proof: It is sufficient to show that the probability $\Pr[i = j|\beta]$ that $i = j$ after k guesses given a light sequence β, is the same as the probability $\Pr[i = j]$ that $i = j$ after k guesses (with no knowledge of the light sequence). From the definitions and Lemma 5,

$$\begin{aligned}
\Pr[i = j \mid \beta] &= (\#\text{ of guesses that fit } \beta \text{ for which } i = j)/(\#\text{ of sequences of guesses that fit } \beta) \\
&= (\#\text{ of } R \text{ for which } i = j)m/(\#\text{ of } R)m \\
&= (\#\text{ of } R \text{ for which } i = j)k!/(\#\text{ of } R)k! \\
&= (\#\text{ of sequences of guesses for which } i = j)/(\#\text{ of} \\
&\qquad \text{of sequences of guesses}) \\
&= \Pr[i = j]\ . \quad \blacksquare
\end{aligned}$$

The G-D game might also be played with two different operations $\bigcirc_1$ and $\bigcirc_2$. It would be interesting to know what properties of $\bigcirc_1$ and $\bigcirc_2$ give an optimal sequence of the form $\bigcirc_1^k \bigcirc_2^m$, for some k and m. In addition to operators which act directly upon i, we might also consider comparison operations which compare i to a given input, n, and answer the question, "$i \leq n$?" If the compare operation $i \leq \sqrt{2N}$ is added to G-D, then the optimal strategy is $O(N^{1/4})$ Guesses followed by a Compare, repeated until $i \leq \sqrt{2N}$, ending with Decrements. This strategy uses $\sqrt{2N} + O(N^{1/4})$ expected steps.

Acknowledgements

We would like to thank Gary Miller, Ron Rivest, Jim Saxe, and Mike Sipser for helpful discussions.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] J.L. Bentley, D.F. Stanat, J.M. Steele, "Analysis of a randomized data structure for representing ordered sets," *Proc. 19th Annual Allerton Conference on Communication, Control, and Computing,* Oct. 1981, pp. 364-372.

[3] W. Janko, "A list insertion sort for keys with arbitrary key distribution," *ACM Transactions on Mathematical Software 2,* 2, June 1976, pp. 143-153.

[4] W. Janko, "An insertion sort for uniformly distributed keys based on stopping theory," *International Computing Symposium,* April 1977, North-Holland Publishing, pp. 373-379.

[5] D.E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching,* Addison-Wesley, Reading, MA 1973.

[6] T. Leighton and M. Lepley, "Probabilistic searching in sorted linked lists," Proc. 20th Annual Allerton Conference on Communication, Control and Computing, Oct. 1982, pp. 500-506.