Algorithmica (1994) 11: 15–32

Algorithmica

Randomized Competitive Algorithms for the List Update Problem[1]

Nick Reingold,[2] Jeffery Westbrook,[3] and Daniel D. Sleator[4]

Abstract. We prove upper and lower bounds on the competitiveness of randomized algorithms for the list update problem of Sleator and Tarjan. We give a simple and elegant randomized algorithm that is more competitive than the best previous randomized algorithm due to Irani. Our algorithm uses randomness only during an initialization phase, and from then on runs completely deterministically. It is the first randomized competitive algorithm with this property to beat the deterministic lower bound. We generalize our approach to a model in which access costs are fixed but update costs are scaled by an arbitrary constant d. We prove lower bounds for deterministic list update algorithms and for randomized algorithms against oblivious and adaptive on-line adversaries. In particular, we show that for this problem adaptive on-line and adaptive off-line adversaries are equally powerful.

Key Words. Sequential search, List-update, On-line algorithms, Competitive analysis, Randomized algorithms.

1. Introduction. Recently much attention has been given to *competitive analysis* of on-line algorithms [7], [20], [22], [25]. Roughly speaking, an on-line algorithm is c-competitive if, for any request sequence, its cost is no more than c times the cost of the optimum off-line algorithm for that sequence. In their seminal work on competitive analysis [25], Sleator and Tarjan studied heuristics commonly used in system software to maintain a set of items as an unsorted linear list. This problem is called the *list update* or *sequential search* problem. The cost of accessing an item is equal to its distance from the front of the list, and the list may be rearranged (at a cost of one per swap of adjacent elements) during the processing of a sequence of requests so that later accesses will be cheaper; for example, a commonly requested item may be moved closer to the front.

Maintaining a dictionary as a linear list is frequently used in practice because of its great simplicity. Furthermore, self-adjusting rules are effective because they take advantage of the locality of reference found in real systems. List update techniques have also been used to develop data compression algorithms [5], as

[1] A preliminary version of these results appeared in a joint paper with S. Irani in the *Proceedings of the 2nd Symposium on Discrete Algorithms*, 1991 [17].

[2] AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974-0636, USA.This research was partially supported by NSF Grants CCR-8808949 and CCR-8958528.

[3] Department of Computer Science, Yale University, New Haven, CT 06520–2158, USA. This research was partially supported by NSF Grant CCR-9009753.

[4] Carnegie-Mellon University, Pittsburgh, PA, USA. This research was supported in part by the National Science Foundation under Grant CCR-8658139, by DIMACS, a National Science Foundation Science and Technology center, Grant No. NSF-STC88-09648.

Received September 18, 1990; revised November 22, 1991. Communicated by Prabhakar Raghavan.

well as fast and simple algorithms for computing point maxima and convex hulls [3], [14]. For all these reasons, the list update problem has been extensively studied [4], [8], [15], [21], [24]. Sleator and Tarjan [25] demonstrated that the move-to-front algorithm, which uses the simple rule of moving an item to the front of the list each time it is accessed, is 2-competitive. Subsequently Karp and Raghavan (private communication, 1990) noted that no deterministic algorithm for the list update problem can be better than 2-competitive, so in a very strong sense move-to-front is as good as any deterministic on-line algorithm.

A great deal of recent work has focused on the use of randomization to improve—sometimes exponentially—the competitiveness of on-line algorithms [2], [12], [13], [18]. Karp and Raghavan (private communication, 1990) inaugurated the study of randomized list update algorithms by showing a lower bound of 1.18 on the competitiveness of any randomized algorithm. Irani discovered a 1.875-competitive randomized algorithm [16], thus exhibiting the first randomized algorithm to beat the deterministic lower bound.

In this paper we examine the effect of randomization in greater depth. We present a very simple randomized algorithm, BIT, that is 1.75-competitive. Our BIT algorithm is not only simple and fast, but, rather remarkably, it makes random choices only during an initialization phase, using exactly one random bit for each item in the list. From then on BIT runs completely deterministically: to process an access to item x, BIT first complements the bit of x, and then moves x to the front if the bit is 1. We call an algorithm that uses a bounded number of random bits regardless of the number of requests *barely random.* Such barely random algorithms have practical value since random bits can be an expensive resource. To our knowledge, BIT is the first barely random algorithm for any on-line problem that provably has a better competitive ratio than any deterministic algorithm for that problem. Recently Alon *et al.* have given a barely random algorithm for k-servers on a circle that is $2k$-competitive [1]. However, it is known for $k = 2$, and conjectured for all k, that a k-competitive deterministic k-server algorithm exists, matching the known lower bound of k [9], [20].

We generalize the BIT algorithm to a family of COUNTER algorithms. Using a COUNTER algorithm, we are able to achieve our best result, a $\sqrt{3}$-competitive algorithm. We also consider a generalized list update model in which the access cost is the same as the standard model but in which the cost of rearrangement is scaled up by some arbitrarily large value d. (This is a very natural extension, because there is no *a priori* reason to assume that the execution time of the program that swaps a pair of adjacent elements is the same as that of the program that does one iteration of the search loop.) For arbitrary list length and swap cost d, the best-known deterministic algorithm is 5-competitive. We give a family of COUNTER algorithms that are always better than 2.75-competitive and that in fact become more competitive as d increases. This gives evidence that the *scaling conjecture* of Manasse *et al.* [20] may apply to randomized algorithms. This version of the list update problem is similar to the replication/migration problems studied by Black and Sleator [6]. We also show a lower bound of 3 on the competitiveness of any deterministic algorithm in this model, so again our randomized algorithms beat the deterministic lower bound.

An important question in the competitive analysis of randomized algorithms is the relationship between oblivious, on-line adaptive and off-line adaptive adversaries. Our upper bounds hold against oblivious adversaries. We show that in the standard model of the list update problem no randomized algorithm can be better than 2-competitive against either an adaptive on-line adversary or an adaptive off-line adversary. These results complement those found in page caching [22] and metrical task systems [12]. In all these applications, randomization helps against oblivious adversaries, and is no use against either kind of adaptive adversaries. Both kinds of adaptive adversaries are equivalent in power. We also show that in the general model scaled by d, no randomized algorithm can be better than 3-competitive against either adaptive adversary.

Lastly, we give some lower bounds for randomized algorithms against an oblivious adversary. We extend the approach of Karp and Raghavan and show a lower bound of 1.27, using numerical techniques. Experimental evidence suggests the lower bound is at least 1.4. This still leaves a substantial gap between the upper and lower bounds.

2. List Update and Competitive Algorithms. The list update problem is that of storing a dictionary as a linear list. A *request* is either an access to an item, an insertion of an item, or a deletion of an item. A list update algorithm must search for an accessed item by starting at the front of the list and inspecting each item in turn until the requested item is found. An insertion is done by searching the entire list to ensure that the item is not already present, and then inserting the new item at the back of the list. A deletion is done by searching for the item and then removing it. At any time, the algorithm may exchange the position of any two adjacent items in the list. In the *standard model* [25] an access or deletion of the ith item in the list costs i and an insertion costs $n + 1$, where n is the length of the list prior to the insertion. Immediately following an access or insertion, the item can be moved any distance forward in the list. These exchanges cost nothing and are called *free* exchanges. Any other exchange costs 1 and is called a *paid* exchange. If there are n items in the list, we assume they are named by the numbers from 1 to n. For a list update algorithm A, the *cost* to service a sequence of requests σ, denoted $A(\sigma)$, is the sum of all costs due to paid exchanges, accesses, insertions, and deletions in the sequence. For any request sequence, σ, there is a minimum cost to service σ, which we denote by $OPT(\sigma)$.

In the *paid exchange* model there are no free exchanges. Furthermore, it is worthwhile to consider models in which the cost of an exchange is significantly greater than the cost of an access, since there is no *a priori* reason to assume that the execution time of the program that swaps a pair of adjacent elements is the same as that of the program that does one iteration of the search loop. Problems such as that of how to arrange data on a storage tape conform to this model. We denote by P^d the paid exchange model in which each paid exchange has cost d, for some real-valued constant $d \geq 1$. Accesses have the same cost as in the standard model. We do not consider insertions and deletions.

We can think of any algorithm as servicing each request as follows: first some number of paid exchanges are performed, then the request is satisfied, and then any number of free exchanges are done. An *on-line* list update algorithm must service each request without any knowledge of future requests. An *off-line* algorithm is shown the entire sequence in advance; the optimum cost can always be achieved by an off-line algorithm. Following [7] and [20] we say a deterministic list update algorithm, A, is *c-competitive* if there is a constant b such that for all size lists and all request sequences σ,

$$A(\sigma) \leq c \cdot OPT(\sigma) + b.$$

For randomized list update algorithms, competitiveness is defined with respect to the model of an *adversary*. Two factors differentiate adversaries: how the request sequences are generated, and how the adversary is charged for servicing the sequence. Following [2] and [22] we consider three kinds of adversary: oblivious (weak), adaptive on-line (medium), and adaptive off-line (strong).

The oblivious adversary chooses a complete request sequence before the on-line algorithm begins to process it. A randomized on-line algorithm, A, is *c-competitive against an oblivious adversary* if there is a constant b such that, for all lists and for all finite request sequences σ,

$$E[A(\sigma)] \leq c \cdot OPT(\sigma) + b.$$

The expectation is over the random choices made by the on-line algorithm.

An *adaptive* adversary is allowed to watch the on-line algorithm in action, and generate the next request based on all previous moves made by the on-line algorithm. That is, an adaptive adversary $\hat{A}$ is a function that takes as input a sequence of $k-1$ requests and corresponding actions by the on-line algorithm, and outputs the kth request, up to a maximum number of requests, m. (Each adversary has its own value of m.) The on-line algorithm and the adversary together generate a probability distribution over request sequences σ, based on the random moves of the algorithm. The adaptive *off-line* adversary is charged the optimum cost for the sequence that is generated. The adaptive *on-line* adversary, however, must service the requests on-line. The sequence of events is this (for each request):

(1) the adverary generates the request,
(2) the adversary services the request, and
(3) the on-line algorithm services the request.

The adaptive adversary models a situation in which the random choices of the algorithm may affect the future request sequence. A randomized on-line algorithm, A, is *c-competitive against an adaptive on-line* (*resp. off-line*) adversary if there is a constant b such that for all size lists, and all adversaries, $\hat{A}$,

$$E[A(\sigma) - c \cdot \hat{A}(\sigma)] \leq b.$$

(Here by σ we mean the request sequence generated by the random choices of the on-line algorithm, over which the expectation is taken.)

DEFINITION. The *competitive ratio against oblivious* (resp. *adaptive on-line, adaptive off-line*) *adversaries* for a randomized on-line algorithm, A, is the infimum of all c for which A is c-competitive against oblivious (resp. adaptive on-line, adaptive off-line) adversaries. We also make the obvious definitions for competitiveness for a fixed size list.

We typically use "c-competitive" as an abbreviation for "c-competitive against an oblivious adversary." When an adaptive adversary is intended we will state so explicitly.

3. The BIT and COUNTER Algorithms. In this section we describe and analyze *BIT*, a very simple randomized algorithm for the list update problem that is 1.75-competitive. We also generalize BIT to a class of algorithms called *COUNTER* algorithms, which (for appropriate parameters) achieve a slightly smaller competitive ratio.

3.1. BIT in the Standard Model. Algorithm BIT works as follows. Associated with each element of the list is a bit which is complemented whenever that item is accessed. We let $b(x)$ denote the bit corresponding to an item x. If an access causes a bit to change to 1 the accessed item is moved to the front, otherwise the list remains unchanged. The n bits are initialized uniformly at random. Roughly speaking BIT is "move-to-front every other access." Notice that BIT uses n random bits regardless of the length of the request sequence.

THEOREM 3.1. *Let σ be any sequence of m accesses and let OPT be the optimum off-line algorithm for σ. The expected cost of the BIT algorithm on σ is at most* $1.75 \cdot \mathrm{OPT}(\sigma) - 3m/4$.

PROOF. This theorem immediately implies that BIT is 1.75-competitive against an oblivious adversary. The proof uses components of Sleator and Tarjan's analysis of the move-to-front heuristic. We imagine that BIT and OPT are running side by side on σ, and partition the actions of both algorithms into two kinds of events: the servicing of an access by both BIT and OPT, possibly involving free exchanges; and a paid exchange made by OPT. Together these events account for all costs incurred by either algorithm. A given σ fixes OPT, and hence fixes the sequence of events. We denote by bit_i and opt_i the cost of event i to BIT and OPT, respectively.

LEMMA 3.2. *For any item x and any j, after the jth event the value of $b(x)$ is equally likely to be 0 or 1, is independent of the position of x in OPT's list, and is independent of the bits of the other items.*

PROOF. The initial assignment of values to bits is chosen uniformly at random from among the 2^n possibilities. The value of $b(x)$ after the jth event is just the number of times x was accessed in events 1 through j plus the initial value of $b(x)$ modulo 2. Therefore the distribution of bit assignments at any time remains the uniform distribution. The lemma follows immediately from these observations. □

Let Φ be a *potential function* that maps a two-tuple consisting of BIT's list and OPT's list to the nonnegative integers, and let Φ_i be the value of Φ immediately after event i. The *amortized cost*, a_i, of event i is defined as

$$a_i = \text{bit}_i + \Phi_i - \Phi_{i-1}.$$

Since BIT is randomized, a_i is a random variable. If $\Phi_0 = 0$, then $\text{BIT}(\sigma) \leq \sum_i a_i$. To prove that BIT is c-competitive it suffices to show that, for each event i, $\text{E}[a_i] \leq c \cdot \text{opt}_i$. Then

$$\text{E}[\text{BIT}(\sigma)] \leq \text{E}\left[\sum_i a_i\right] = \sum_i \text{E}[a_i] \leq c \cdot \sum_i \text{opt}_i = c \cdot \text{OPT}(\sigma).$$

We now define the potential function. An *inversion* is an ordered pair of items (y, x) such that x occurs after y in the list maintained by BIT while x occurs before y in the list maintained by OPT (the optimum ordering of the pair). The set of inversions changes with time as BIT and OPT rearrange their lists. The cost to BIT of an access to item x is given by the cost of the access to OPT plus the number of inversions of the form (y, x) minus the number of inversions of the form (x, y), where y denotes any item other than x.

Inversion (y, x) is called a *type 1* inversion if $b(x) = 0$ and is called a *type 2* inversion if $b(x) = 1$. The type of the inversion (y, x) is the number of accesses to x before x next moves to the front, and is given by $1 + b(x)$. Let φ_1 denote the number of type 1 inversions and let φ_2 denote the number of type 2 inversions. Then

$$\Phi = 2\varphi_2 + \varphi_1.$$

Note that $\Phi_0 = 0$, since both BIT and OPT begin with the same list. Using the potential function, we bound the cost of the two types of event.

Case 1: Event i is an access to item x. Let k be the position of x in OPT's list; then $\text{opt}_i = k$. Let R be a random variable that counts the number of inversions of the form (y, x) at the time of the access. Then bit_i, the actual cost to BIT, is at most $k + R$.

We write $\Delta\Phi = \Phi_i - \Phi_{i-1} = A + B + C$, where A is a random variable giving the change in potential due to new inversions created during the access, B is a random variable giving the change in potential due to old inversions removed

during the access, and C is a random variable giving the change in potential due to old inversions that change type during the access.

Suppose that $R = r$ and consider the value of $B + C$. If $b(x) = 1$, then $B = 0$ and $C = -r$, since x stays in place and each inversion (y, x) goes from type 2 to type 1. If $b(x) = 0$, then $B = -r$ and $C = 0$, since x moves to the front and removes all inversions (y, x) and each such inversion is of type 1. In both cases $B + C = -r = -R$. Hence

$$\begin{aligned} \mathrm{E}[a_i] &= \mathrm{E}[\mathrm{bit}_i + \Delta\Phi] \\ &\le \mathrm{E}[(k + R) + (A - R)] \\ &= k + \mathrm{E}[A]. \end{aligned}$$

The value of A depends on both BIT and OPT, since either may move x forward using free exchanges. Let $z_1, z_2, \ldots, z_{k-1}$ be the items preceding x in OPT's list prior to the access. A new inversion can be created only when, for some i, z_i also precedes x in BIT's list, and one of BIT or OPT, but not both, move x forward past z_i. Suppose that OPT moves x forward to position k'. Let Z_i be a random variable that measures the change in potential due to each pair $\{x, z_i\}$.

If $b(x) = 0$, then x moves to the front of BIT's list. In the worst case a new inversion (x, z_i) of type $1 + b(z_i)$ is created for $1 \le i \le k' - 1$. This implies that $Z_i \le 1 + b(z_i)$ for $1 \le i \le k' - 1$ and $Z_i \le 0$ for $k' \le i \le k - 1$.

If $b(x) = 1$, then x does not move and $b(x)$ becomes 0. In the worst case a new inversion (z_i, x) of type 1 is created for $k' \le i \le k - 1$. This implies that $Z_i = 0$ for $1 \le i \le k' - 1$ and $Z_i \le 1$ for $k' \le i \le k - 1$.

By Lemma 3.2, $\mathrm{E}[b(y)] = \frac{1}{2}$ for all items y, and hence

$$\begin{aligned} \mathrm{E}[A] &= \sum_{i=1}^{k-1} \mathrm{E}[Z_i] \\ &\le \sum_{i=1}^{k'-1} \tfrac{1}{2}(\tfrac{1}{2} \cdot 2 + \tfrac{1}{2} \cdot 1) + \sum_{i=k'}^{k-1} \tfrac{1}{2} \cdot 1 \\ &\le \tfrac{3}{4}(k - 1). \end{aligned}$$

Thus the expected amortized cost of the access is no more than $1.75k - \frac{3}{4}$, i.e., $\mathrm{E}[a_i] \le 1.75 \cdot \mathrm{opt}_i - \frac{3}{4}$.

Case 2: Event i is a paid exchange by OPT of items x and y. OPT pays 1 for the exchange. In the worst case the exchange creates an inversion (y, x). Again applying Lemma 3.2, this inversion increases the value of Φ by 2 with probability 1/2 (the probability that $b(x) = 1$) and by 1 with probability 1/2. Hence $\mathrm{E}[a_i] \le 1.5 \cdot \mathrm{opt}_i$.

Summing over all events, noting that m of the events are accesses, completes the proof of Theorem 3.1. □

The analysis of case 1 can be simplified by using a theorem of Reingold and Westbrook [23] which states that for any request sequence σ there is an optimum algorithm that does only paid exchanges. For thoroughness, however, we consider free exchanges performed by OPT, since some researchers have studied a variant of the standard model in which only free exchanges are allowed [16].

The extension of BIT to handle insertions and deletions is straightforward. On an insertion, BIT inserts the item at the back of the list, randomly initializes the bit of the new item, then moves the item to the front if its bit is 1. Suppose the list has size n. The insertion is clearly equivalent to the first access to the $(n+1)$st item in a list of size $n+1$. Thus the analysis applied above to accesses can be applied here to show that an insertion has the same expected cost as an access. (This observation is due to Sleator and Tarjan [25].) Similarly, a deletion is like an access, except that all inversions involving the deleted item disappear, so the potential function decreases even more than during an access.

COROLLARY 3.3. *Let σ be any sequence of m accesses, insertions, and deletions to an initially empty list, and let OPT be any deterministic off-line algorithm that services σ. The expected cost of the BIT algorithm on σ is at most*

$$1.75 \cdot \mathrm{OPT}(\sigma) - 3m/4.$$

The results for BIT can be extended to a closely related list update model of interest in which an access to the ith item in the list has cost $i-1$ rather than i. Any algorithm that costs C on some sequence σ of accesses in the i cost model will cost $C - m$ in the $i-1$ model, where $m = |\sigma|$.

COROLLARY 3.4. *For any sequence of accesses, σ, the expected cost of the BIT algorithm on σ in the model in which an access to the* ith *item costs $i-1$ is at most* $1.75 \cdot \mathrm{OPT}(\sigma)$.

PROOF. Applying Theorem 3.1, we have that in the $i-1$ model

$$\mathrm{BIT}(\sigma) + m \leq 1.75(\mathrm{OPT}(\sigma) + m) - 3m/4,$$

which implies that $\mathrm{BIT}(\sigma) \leq 1.75 \cdot \mathrm{OPT}(\sigma)$. □

An alternate proof for the $i-1$ cost model can be given as follows. The problem for arbitrary length lists can be factored into $\binom{n}{2}$ different problems on lists of length two. If an algorithm is c-competitive on each of these two-element lists, then it is c-competitive on the list as a whole. This *pairwise independence property* of move-to-front-type heuristics was first observed by Bentley and McGeoch [4], who made use of it to prove that the deterministic move-to-front algorithm is within a factor of 2 of any static list.

The upper bound for BIT is tight in the $i-1$ cost model. Consider a list of size

two, initially ordered 1, 2. A worst-case access sequence of length $3k$ is given by $\sigma = \gamma^k$, where $\gamma = \langle 2, 1, 1 \rangle$. Each γ can be serviced at cost 1, by always leaving the list in position 1, 2. The expected cost to BIT, however, is 1.75, which can be seen as follows. At the start of each γ, BIT's list is ordered 1, 2. BIT pays 1 for the access to item 2. With probability 1/2, BIT makes no exchange, and pays 0 for the two accesses to item 1. With probability 1/2, however, BIT flips the list to 2, 1. In this case, BIT pays 1 for the first access to item 1, and with probability 1/2 pays 1 for the second access. Thus the total expected cost of the two accesses to item 1 is 0.75. We do not know if the upper bound is tight in the i cost model.

Our upper bounds can also be generalized to a convex cost model, in which an access to the ith item has cost $f(i)$ and a paid exchange of the ith item with its successor has cost $f(i+1) - f(i)$, for some convex function f.

3.2. COUNTER in the Standard Model. It is possible to modify the BIT algorithm to improve the competitive ratio, at the expense of making the algorithm more complicated. Let s be a positive integer, and let S be any nonempty subset of $\{0, 1, \ldots, s-1\}$. The algorithm COUNTER(s, S) keeps a mod s counter for each item. Initially, each counter is randomly set to some number in $\{0, 1, \ldots, s-1\}$, each value chosen independently and with equal probability. At a request to item x, COUNTER decrements the x's counter mod s, and then moves x to the front via free exchanges if x's counter is in S. BIT is COUNTER(2, $\{1\}$). Any COUNTER algorithm has the property that, for a fixed size list, it uses a fixed number of random bits regardless of the size of the request sequence.

For an item x, let $c(x)$ be the number of accesses to x before x moves to the front. For $j = 1, 2, \ldots, s$, let $p_j = (1/s)|\{i: c(i) = j\}|$; this is the probability that an item will next move to the front after j accesses. After the initialization phase and before any accesses, $\Pr[c(x) = j]$ is p_j for any item x.

THEOREM 3.5. *COUNTER(s, S) is* $\max\{\sum_{j=1}^{s-1} jp_j, 1 + p_1 \sum_{j=1}^{s-1} jp_j\}$*-competitive.*

The analysis of COUNTER(s, S) is very similar to the analysis of BIT, so we just sketch the differences. The analogue of Lemma 3.2, namely, that at any time the probability that $c(x) = j$ is p_j, independent of the position of x in OPT's list, is easily verified.

An inversion (y, x) is of *type* j if $c(x) = j$. Let φ_j denote the number of inversions of type j. Our potential function is $\Phi = \sum_{j=1}^{s} j \cdot \varphi_j$.

Consider the expected amortized cost of an access to x. If x does not move to the front then, since $c(x)$ decreases by one, the decrease in potential due to inversions which change type is exactly the number of inversions (w, x). If x moves to the front the number of inversions destroyed is exactly the number of inversions (w, x). In either case the expected amortized cost of the access is at most the position of x in OPT's list plus the expected cost of new inversions created. It is not hard to see that the expected cost of the inversions created is at most $(k-1)p_1 \sum_{j=1}^{s} jp_j$ (where k is the position of x in OPT's list), so the cost to COUNTER(s, S) is at most $(1 + p_1 \sum_{j=1}^{s} jp_j)$ times the cost to OPT.

For a paid exchange by OPT, the worst case is that one new inversion is created. The expected increase in potential is $\sum_{j=1}^{s} jp_j$. Therefore we have that in this case the cost to COUNTER(s, S) is no more than $(\sum_{j=1}^{s} jp_j)$ times the cost to OPT. It is also possible to show that the same competitive ratio is achieved if the cost of accessing the ith item is $i - 1$. This completes the analysis of COUNTER. □

For some choices of s and S, we can get a better competitive factor than 1.75. For example, COUNTER(7, {0, 2, 4}) is 85/49-competitive (≈ 1.735-competitive). In fact, it is possible to modify the COUNTER algorithm to achieve a competitive ratio of $\sqrt{3}$. Let a RANDOM RESET algorithm be as follows. Keep a counter from 1 to s for each item. When an accessed item has a counter which is bigger than 1, make no move and decrement the counter. If the counter is 1, move that item to the front and reset the counter to j with some probability π_j. Thus each counter follows a simple Markov chain; in the stationary distribution of this chain the counter has value j with probability p_j (where these are as defined above). The value of the counters must be initialized according to the stationary distribution. For example, COUNTER(7, {0, 2, 4}) can be simulated by a random reset algorithm that keeps a counter from 1 to 3. When an accessed item has counter value 1 it is moved to the front and the counter is reset to 3 with probability 1/3 and to 2 with probability 2/3. Each counter is initialized by choosing the value 1 with probability 3/7, 2 with probability 3/7, and 3 with probability 1/7.

The same analysis as for COUNTER works for RANDOM RESET algorithms. We wish to choose a resetting distribution which gives the lowest possible competitive ratio as guaranteed by Theorem 3.5. The best ratio possible is $\sqrt{3}$ (we do not prove this here), and it is easy to achieve. This yields the best randomized algorithm known for the list update problem.

THEOREM 3.6. *The following resetting distribution yields a $\sqrt{3}$-competitive algorithm: reset to* 3 *with probability* $(-1 + \sqrt{3})/2$ *and to* 2 *with probability* $(3 - \sqrt{3})/2$.

The RANDOM RESET algorithm has the drawback that it uses $\Omega(m)$ random bits to process a request sequence of length m. It is possible, however, to find s and S such that the competitive ratio of COUNTER(s, S) guaranteed by Theorem 3.5 is as close to $\sqrt{3}$ as desired (though s must tend to infinity). Although such a COUNTER algorithm potentially has a very large finite state machine, the total number of random bits needed is still $O(n)$.

3.3. COUNTER in P^d Models. In this subsection we consider COUNTER algorithms for list update in P^d models. Consider the algorithm COUNTER(k, $\{k-1\}$): For each item in the list keep a mod k counter. Initially, the counters are set by choosing values uniformly and independently from $\{0, 1, \ldots, k-1\}$. On an access to an item, decrement the counter mod k and then move the requested item to the front if its counter is $k - 1$.

THEOREM 3.7. *In the model P^d, COUNTER$(k, \{k-1\})$ is c-competitive, where c is the maximum of* $1 + (k+1)/2d$ *and* $1 + (1/k)(2d + (k+1)/2)$.

PROOF. The analysis is very similar to the analysis for BIT. We say an inversion (y, x) is of type i if the value of x's counter is $i - 1$. As our potential function, we take $\Phi = \sum_{i=1}^{k}(d + i)\varphi_i$, where φ_i is the number of inversions of type i. The factor of i in the potential pays for the increased access cost due to the inversion, and the factor of d pays for the cost of removing that inversion when the item is moved to the front.

Consider the amortized cost of an access to x. The cost of this access to the optimum algorithm is j, where j is the position of x in OPT's list. The amortized cost for the access is at most $j + R + A + B + C + D$, where R is the number of inversions (w, x) for some w at the time of the access, A is the change in potential due to new inversions created, B is the change in potential due to old inversions destroyed, C is the change in potential due to inversions which change type, and D is the cost of exchanges made.

Suppose that at the time of the access, x's counter is 0, and that $R = r$. Then $B = -r(d+1)$, $C = 0$, and $D \le d(j + r - 1)$, so that $R + B + C + D \le (j-1)d$. If x's counter is not 0, then $B = 0$, $C = -r$, and $D = 0$, so that $R + B + C + D = 0$. Therefore $\mathrm{E}[R + B + C + D] \le (j-1)d/k$. The expected value of A is no more than $(j-1)(1/k)\sum_{i=1}^{k}(d+i)(1/k) = (j-1)(1/k)(d + (k+1)/2)$. Therefore, the expected amortized cost of the access is

$$\mathrm{E}[j + R + A + B + C + D] = \mathrm{E}[j] + \mathrm{E}[R + B + C + D] + \mathrm{E}[A]$$

which is no more than

(1)
$$j + \frac{(j-1)d}{k} + \left(\frac{j-1}{k}\right)\left(d + \frac{k+1}{s}\right) = j + \left(\frac{j}{k}\right)\left(2d + \frac{k+1}{2}\right) - \left(\frac{1}{k}\right)\left(2d + \frac{k+1}{2}\right).$$

This shows that the expected amortized cost for the access is no more than

$$1 + \left(\frac{1}{k}\right)\left(2d + \frac{k+1}{2}\right)$$

times the cost to OPT.

The cost of an exchange by OPT is handled similarly. In the worst case, one inversion is created, and the expected change in potential due to that inversion is $d + (k+1)/2$. Since OPT pays d for the exchange, COUNTER$(k, \{k-1\})$ pays no more than $1 + (k+1)/2d$ times the optimum cost. □

COUNTER$(k, \{k-1\})$ achieves the competitive ratio given above in the

Table 1. Best competitive ratios in P^d for COUNTER and RANDOM RESET.

	COUNTER		RANDOM RESET		
d	Best k	Competitive ratio	Best k	Competitive ratio	Pr[reset to k]
1	2	2.75	3	$\sqrt{7} \approx 2.64$	0.215
2	5	2.50	5	$\sqrt{6} \approx 2.45$	0.760
3	7	2.43	8	≈ 2.39	0.314
4	10	2.38	10	≈ 2.36	0.878
5	12	2.38	13	≈ 2.34	0.433
6	15	7/3	15	$7/3 \approx 2.33$	1 (exactly)

model where the cost of accessing the ith item in the list costs $i - 1$, just as in Corollary 3.4.

Table 1 shows, for various d, the best competitive ratio for

$$\text{COUNTER}(k, \{k - 1\})$$

guaranteed by Theorem 3.7. There values were derived as follows. Set the access and exchange ratios derived above equal to each other, and solve for k in terms of d. It is interesting that as d tends to infinity, the best competitive ratio decreases and tends to $(5 + \sqrt{17})/4 \approx 2.28$. This gives strong evidence that the scaling conjecture of Manasse *et al.* [20] is true for the list update problem.

It is possible to find RANDOM RESET algorithms which do slightly better than the COUNTER competitive ratios given in Table 1. These results are also shown in Table 1. The best resetting distributions we have found all have the property that the counter is reset to either the largest or the second largest possible value. Table 1 also lists, for various d, the best k, the best competitive ratio, and the probability of resetting to k. Notice, however, that while all COUNTER algorithms use $O(n)$ random bits regardless of the length of the request sequence, these random resetting algorithms use $\Omega(m)$ random bits for a request sequence of length m. Any random resetting algorithm can be approximated to any desired degree of accuracy by a COUNTER algorithm with a large enough s.

In the following section we prove that no deterministic P^d algorithm can be better than 3-competitive. Thus our randomized algorithms beat the deterministic bound. We have recently found a 5-competitive deterministic algorithm for all d; an open problem is to tighten the deterministic upper bound.

4. Lower Bounds Against Adaptive Adversaries. In this section we prove that for randomized list update algorithms the adaptive on-line and adaptive off-line adversaries are equally powerful. An unpublished result of Karp and Raghavan shows that if A is a deterministic, c-competitive list update algorithm in the standard model, then $c \geq 2$. By a theorem of Ben-David *et al.* [2], a lower bound

for deterministic algorithms implies the same lower bound for randomized algorithms against adaptive off-line (strong) adversaries. Note that the move-to-front algorithm is 2-competitive. Now we show that even against an adaptive on-line (medium) adversary, no randomized algorithm is better than 2-competitive.

THEOREM 4.1. *If A is c-competitive against an adaptive on-line (medium) adversary, then $c \geq 2$.*

PROOF. The adversary's strategy is to generate an access sequence of length m by always accessing the last item in A's list. The cost to A is mn. Consider all $n!$ *static* algorithms, i.e., algorithms which initially arrange a list of length n in some order, and make no other exchanges. Suppose that on a given request sequence, we run all static algorithms simultaneously. For any one access, the sum of the costs to each of these algorithms is

$$\sum_{i=1}^{n} i(n-1)! = (n-1)!\, n(n+1)/2 = n!\,(n+1)/2.$$

Therefore, for any sequence of m accesses the sum of the costs to these algorithms is $mn!\,(n+1)/2 + B$, where B is the sum of the initial costs of rearranging the list. The medium adversary which selects one of these static algorithms uniformly at random achieves an expected cost of $m(n+1)/2 + B/n!$ on any access sequence of length m. Therefore, for any ε the adversary can choose m so large that the ratio of A's cost to the adversary's cost is at least $2n/(n+1) - \varepsilon$. Since A must be c-competitive for all list sizes, c cannot be less than 2. □

Notice that if we fix the size of the list to be n we get a lower bound of $2n/(n+1)$ for the competitive ratio.

The above proof shows that for any request sequence there is a static algorithm that achieves an average cost of $(n+1)/2$ per access. (It is not hard to show that the algorithm which arranges the list in order of decreasing access frequency achieves this.) The above proof thus works for deterministic on-line algorithms and for randomized on-line algorithms against strong adversaries.

We now show that 3 is a lower bound on the competitive factor for problems in the P^d models. This result generalizes that of Karlin *et al.* [19], whose Theorem 3.3 can be interpreted as showing a lower bound of 3 on the competitive factor for lists of length 2 in the $i-1$ cost model.

THEOREM 4.2. *Let A be an on-line algorithm for list update in a P^d model. If A is deterministic and c-competitive, then $c \geq 3$. If A is randomized and c-competitive against adaptive off-line adversaries, then $c \geq 3$.*

PROOF. In either case, the adversary's strategy is to generate an access sequence of length m by always accessing the last item in A's list. The cost to A is $mn + D$, where D is the total cost of all exchanges made by A. Let $b = D/mn$. Suppose

$b \geq 1/2$. As in the theorem above, the adversary can arrange to pay arbitrarily close to $m(n+1)/2$. Since D is by assumption at least $mn/2$, A cannot be more than $3n/(n+1)$-competitive. On the other hand, suppose $b < \frac{1}{2}$. Now the adversary adopts the following strategy: at all times it maintains its list in the exact reverse order of A's list. That is, the adversary initially reverses its list and then does the reverse of every exchange A makes. The total cost to the adversary is at most $m + D + dn(n-1)/2$, since each access costs it 1. Ignoring the initial cost of reversal, the ratio of A's cost to the adversary's cost is $n(1+b)/(1+bn)$, which is at least $3n/(n+2)$ for $b \leq \frac{1}{2}$. Since one of the two strategies is applicable for all n, A cannot be better than 3-competitive. □

Notice that if we fix the size of the list to be n we get a lower bound of $3n/(n+2)$. Theorem 4.2 can be extended to the model in which an access to item i has cost $i-1$. In this situation the lower bound is exactly 3 regardless of n. The technique used in the proof of Theorem 4.2 can be extended to handle randomized algorithms against adaptive on-line adversaries.

THEOREM 4.3. *Let A be an on-line algorithm for list update in a P^d model. If A is randomized and c-competitive against adaptive on-line adversaries, then $c \geq 3$.*

PROOF. Suppose that A is an on-line algorithm. Consider $\hat{A}$, an adaptive on-line adversary, which behaves as follows. $\hat{A}$ first simulates A on all possible choices of A's random bits, and for each such choice of random bits, creates a request sequence of length m such that each request is to the last item in A's list. Consider the collection of all request sequences constructed in this way. The choice of A's random bits induces a probability distribution on these (indeed, all) sequences of length m. That is, the probability of a sequence is the probability that $\hat{A}$ will generate that sequence when it runs A and uses the strategy of always accessing the last item in A's list. $\hat{A}$ can compute the expected value of $b = D/mn$. Based on the expected value of b, $\hat{A}$ chooses one or the other strategy as in the deterministic case, and achieves the same lower bounds. □

The results of this section show that no randomized on-line algorithm can beat the deterministic lower bound against an adaptive on-line algorithm. It is known that the analogous theorem to Theorem 4.1 holds for other types of on-line situations such as the caching, but it is not known to hold in more general settings. The relationship between on-line adaptive and off-line adaptive adversaries in more general settings is an interesting open problem.

5. Lower Bounds Against Oblivous Adversaries. In this section we discuss a technique for deriving lower bounds for randomized on-line algorithms against oblivious adversaries. In this section when we refer to competitive ratios we mean competitive ratio against an oblivious adversary.

Karp and Raghavan (private communication, 1990) use the following strategy:

find an off-line algorithm with a small average cost per request assuming that the requests are drawn uniformly at random from $\{1, 2, \ldots, n\}$. If the average cost per request is no more than d, then no on-line algorithm can be better than $(n + 1)/2d$-competitive. Karp used this technique to obtain a lower bound of 9/8 for the competitive ratio of any on-line algorithm for lists of two items, and Raghavan (private communication, 1990) showed a lower bound of 1.18 for three-item lists. We extend these results by considering larger lists, and more complicated off-line algorithms. Fix the size of the list, n, and a *lookahead* number, k. Suppose we have an off-line algorithm that bases its decision on only the next $k + 1$ requests. Following Raghavan's approach, we can model the behavior of such an algorithm for randomly generated request sequences as follows. Consider a Markov chain whose states are k-tuples of request positions. State $\langle r_1, r_2, \ldots, r_k \rangle$ corresponds to the situation where the current *positions* of the next k requests are, respectively, $r_1, r_2, \ldots, r_k$. The transitions are conditioned on the current position of the $(k + 1)$st request. If we know the algorithm, we can construct the transition matrix for the chain.

Each state transition has a cost (the cost of any paid exchanges made just prior to the access plus the cost of the access), so for each state we can compute an expected cost. Let the vector of these expected costs be $\mathbf{c}$. Suppose the chain has stationary distribution π. Then the expected cost per access is $\mathbf{c} \cdot \pi$, if the chain is in steady state.

For $n \geq 3$ there is no obvious strategy for a k-lookahead algorithm, since no bounded lookahead algorithm can be optimum [23]. Furthermore, the size of the transition matrix is n^k, so for values of n and k even a little bigger than 3 it is infeasible to try different strategies by hand and compute the steady-state vector symbolically. Thus there are two problems: determining a good way to find off-line algorithms and generate transition rules for large lists, and finding the steady-state distribution of the resultant transition matrix.

Our approach is to write a program that generates the entries of the matrix corresponding to a particular class of good off-line algorithms, MARKOV(n, k), and then compute the steady-state vector numerically. Suppose we have a program OFF(n, k) that, given a sequence of exactly $k + 1$ accesses to an n-item list, generates a sequence of moves to service that request sequence. MARKOV(n, k) works as follows: start with the next k requests; look at the $(k + 1)$st request; service the next request in the same way that OFF(n, k) would service that request given the same sequence of $k + 1$ requests on the same list. Given an implementation of OFF(n, k), the entries of the transition matrix and cost vector for MARKOV(n, k) are filled in as follows. If the current state is $\langle r_1, r_2, \ldots, r_k \rangle$ (that is, if the next k requests are to the items in positions $r_1, r_2, \ldots, r_k$), and the next request is to the item in position r_{k+1}, transit to whatever state corresponds to the list that results from OFF(n, k) servicing request r_1 in the sequence $\langle r_1, r_2, \ldots, r_{k+1} \rangle$, assuming its list is $1, 2, \ldots, n$. The probability of that transition is $1/n$, since the request sequence is uniformly random. The cost of that state is $1/n$ times the sum over all choices of r_{k+1} of the cost of servicing the first request in $\langle r_1, r_2, \ldots, r_{k+1} \rangle$. Thus to generate the matrix we must generate all possible sequences of $k + 1$ requests on n items and run OFF(n, k) on each of them.

Any choice of an off-line algorithm gives a valid MARKOV(n, k) algorithm and transition matrix, but some choices are better than others. Our best results were achieved using a program to compute an optimum off-line algorithm for the $k + 1$ requests that has the following properties:

1. It only does paid exchanges.
2. It services a request to item x by choosing some subset of the items preceding x in the list and moving them in an order-preserving way to immediately after x. Then it pays for the access to x and goes on to the next request.
3. Whenever an item is requested twice in a row that item is moved to the front on its first access. This means that MARKOV(n, k) has the same property and ensures that the Markov chain has at most one stationary distribution (see Lemma 5.1).
4. Among optimum algorithms that have the above three properties, our algorithm services the first request with the lowest cost. This tends to make MARKOV(n, k)'s cost per access smaller, so we get a better lower bound.

It is shown in [23] that there is an optimum algorithm with the first three properties. There is no known way to compute the optimum algorithm for a given request sequence that is polynomial in both n and k, however, so the time and space to generate MARKOV(n, k) grow exponentially.

LEMMA 5.1. *For any n and k, the Markov chain corresponding to MARKOV(n, k) is irreducible. That is, for any two states, the probability of transiting from one to the other in some finite time is positive.*

PROOF. Suppose we want to get to state $\langle r_1, r_2, \ldots, r_k \rangle$. Consider MARKOV($n$, k)'s action on request sequence $\langle n, n, n-1, n-1, \ldots, 2, 2, 1, 1 \rangle$. MARKOV($n$, k) will move each item to the front, since each item is requested twice in a row. After that, MARKOV(n, k)'s list must be $1, 2, \ldots, n$, so if the next k requests are $\langle r_1, r_2, \ldots, r_k \rangle$, MARKOV($n$, k) will be in state $\langle r_1, r_2, \ldots, r_k \rangle$. □

By standard probability theory, Lemma 5.1 implies that the steady-state distribution is unique, and it is given by the (unique) eigenvector of the transition matrix corresponding to the eigenvalue 1. We compute this eigenvector from the matrix using the power method [11]. Table 2 shows the best results we have obtained for $n = 3, 4, 5$, and 6. In all cases, the number of iterations in the power method necessary to get the distance between successive iterates less than 10^{-7}

Table 2. Lower bound results.

n	k	Lower bound
3	10	1.1998
4	8	1.2467
5	7	1.2728
6	5	1.268

was no more than 30. As shown in the table, the largest lower bound we have been able to compute is approximately 1.27. We were unable to achieve higher results due to limitations on computational resources. From various simulations using nonoptimum off-line algorithms we believe the true lower bound to be at least 1.4.

6. Remarks and Open Problems. For the standard model, we have given an extremely simple algorithm that is 1.75-competitive against an oblivious adversary, and constructed a slightly more complicated one which is $\sqrt{3}$-competitive. We have shown that no algorithm can be better than 1.27-competitive against such an adversary. (Recently, we have improved the lower bounds for three- and four-item lists to 1.2 and 1.25, respectively.) This leaves a substantial gap. It is possible that our COUNTER algorithms are better than we can currently prove, since we do not know of an instance in which the upper bound is tight.

OPEN QUESTION 1. What is the best competitive ratio a randomized list update algorithm can achieve against oblivious adversaries in the standard model?

In the P^d models we have given a lower bound of 3 for the competitiveness of deterministic algorithms, and for randomized algorithms against adaptive adversaries. We have constructed randomized algorithms with smaller competitive ratios against oblivious adversaries for these models.

OPEN QUESTION 2. What is the best competitive ratio a randomized list update algorithm can achieve against oblivious adversaries in these models?

Our BIT and COUNTER algorithms use only a bounded number of random bits regardless of the number of requests serviced, yet still beat the deterministic lower bound. Recently barely random algorithms have also been found for the migration problem [10], [26].

OPEN QUESTION 3. For which other on-line problems do such algorithms exist?

Our results in Section 4 give evidence for the conjecture that, for a large class of applications, adaptive on-line and adaptive off-line adversaries are equally powerful. Similar results have been obtained for page caching [22] and metrical task systems [12]. On the other hand, the results of [10] and [26] show that this does not hold in general.

OPEN QUESTION 4. For what other classes of on-line problems are these two adversaries equivalent?

Acknowledgments. We thank Richard Beigel, Sandy Irani, Prabhakar Raghavan, and Neal Young for useful discussions.

References

[1] N. Alon, R. M. Karp, D. Peleg, and D. West. A graph-theoretic game and its application to the k-server problem. *Proc. DIMACS Workshop on On-line Algorithms*, pages 1–10. American Mathematical Society, Providence, RI, 1991.

[2] S. Ben-David, A. Borodin, R. M. Karp, G. Tárdos, and A. Wigderson. On the power of randomization in on-line algorithms. *Proc. 20th ACM Symp. on Theory of Computing*, pages 379–386, 1990.

[3] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. *Proc. 1st ACM–SIAM Symp. on Discrete Algorithms*, pages 179–187, 1990.

[4] J. L. Bentley and C. C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Comm. ACM*, 28(4):404–411, 1985.

[5] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Comm. ACM*, 29(4):320–330, 1986.

[6] D. L. Black and D. D. Sleator. Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[7] A. Borodin, N. Linial, and M. Saks. An optimal on-line algorithm for metrical task systems. *Proc. 19th ACM Symp. on Theory of Computing*, pages 373–382, 1987.

[8] P. J. Burville and J. F. C. Kingman. On a model for storage and search. *J. Appl. Probab.*, 10:697–701, 1973.

[9] M. Chrobak and L. Larmore. On fast algorithms for two servers. *J. Algorithms*, 12:607–614, 1991.

[10] M. Chrobak, L. L. Larmore, N. Reingold, and J. Westbrook. Optimal multiprocessor migration algorithms using work functions. Technical Report YALEU/DCS/TR-897, Department of Computer Science, Yale University, 1991.

[11] S. D. Conte and C. de Boor. *Elementary Numerical Analysis, An Algorithmic Approach*, 3rd edn. McGraw-Hill, New York, 1980.

[12] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir. Random walks on weighted graphs, and applications to on-line algorithms. *Proc. 20th ACM Symp. on Theory of Computing*, pages 369–377, 1990.

[13] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. D. Sleator, and N. Young. On competitive algorithms for paging problems. *J. Algorithms*, 12:685–699, 1991.

[14] M. J. Golin. Ph.D. thesis, Department of Computer Science, Princeton University, 1990. Technical Report CS-TR-266-90.

[15] W. J. Hendricks. An account of self-organizing systems. *SIAM J. Comput.*, 5(4):715–723, 1976.

[16] S. Irani. Two results on the list update problem. *Inform. Process. Lett.*, 38:301–306, 1991.

[17] S. Irani, N. Reingold, J. Westbrook, and D. D. Sleator. Randomized algorithms for the list update problem. *Proc. 2nd ACM–SIAM Symp. on Discrete Algorithms*, pages 251–260, 1991.

[18] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki. Competitive randomized algorithms for non-uniform problems. *Proc. 1st ACM–SIAM Symp. on Discrete Algorithms*, 1990.

[19] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[20] M. Manasse, L. A. McGeoch, and D. Sleator. Competitive algorithms for on-line problems. *Proc. 20th ACM Symp. on Theory of Computing*, pages 322–333, 1988.

[21] J. McCabe. On serial files with relocatable records. *Oper. Res.*, 13:609–618, 1965.

[22] P. Raghavan and M. Snir. Memory versus randomization in on-line algorithms. Research Report RC 15622 (No. 69444), IBM T. J. Watson Reseach Center, 1990.

[23] N. Reingold and J. Westbrook. Optimum off-line algorithms for the list update problem. Technical Report YALEU/DCS/TR-805, Yale University, 1990.

[24] R. Rivest. On self-organizing sequential search heuristics. *Comm. ACM*, 19(2):63–67, 1976.

[25] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, 1985.

[26] J. Westbrook. Randomized algorithms for multiprocessor page migration. *Proc. DIMACS Workshop on On-Line Algorithms*, pages 135–150. American Mathematical Society, Providence, RI, 1991.