MTF, *BIT*, and *COMB*: A Guide to Deterministic and Randomized Online Algorithms for the List Access Problem

Kevin Andrew
kandrew@cs.hmc.edu

David Gleich
dgleich@cs.hmc.edu

April 30, 2004

Abstract

In this survey, we discuss two randomized online algorithms for the list access problem. First, we review competitive analysis and show that the *MTF* algorithm is 2-competitive using a potential function. Then, we introduce randomized competitive analysis and the associated adversary models. We show that the randomized *BIT* algorithm is 7/4-competitive using a potential function argument. We then introduce the pairwise property and the *TIMESTAMP* algorithm to show that the *COMB* algorithm, a COMBination of the *BIT* and *TIMESTAMP* algorithms, is 8/5-competitive. *COMB* is the best known randomized algorithm for the list access program.

Contents

1 Introduction **2**

2 Competitive Analysis and the List Access Problem **2**
2.1 *MTF* is 2-competitive 3
2.2 Online Deterministic List Access Algorithms are 2-competitive 5

3 Randomized Competitive Analysis and the *BIT* Algorithm **7**
3.1 Adversary Models 7
3.2 The *BIT* Algorithm 7
3.3 *BIT* is 7/4-competitive! 8

4 The Pairwise Property and the *COMB* Algorithm **12**
4.1 The Pairwise Property 12
4.2 The *TIMESTAMP* Algorithm 14
4.3 The *COMB* Algorithm 15

5 Conclusion **20**

1 Introduction

The List Access Problem, originally proposed by Sleator and Tarjan [5], is a common problem used to introduce online algorithms. As we saw in class (and as we'll see in this paper), the best deterministic algorithm for this problem is 2-competitive and no algorithm can be any better. However, this bound does not apply to randomized algorithms. Later, we'll see two randomized algorithms: BIT and $COMB$. BIT [4] is 7/4-competitive and $COMB$ [1] is 8/5-competitive. Currently, $COMB$ is the best known randomized algorithm for the List Access Problem.

2 Competitive Analysis and the List Access Problem

Before we delve into the details of BIT and $COMB$, we'll spend some time reviewing the results from class. In this section, we review competitive analysis and reproduce the proofs that the MTF algorithm is 2-competitive and that any deterministic online algorithm is, at best, 2-competitive.

The List Access Problem is an online algorithm problem. There is a sequence of requests to elements in a list; however, the online algorithm does not know the entire sequence and must serve the request in order. In this case, it doesn't make sense to look at the actual cost of the algorithm because a suitable adversary could always ask for the "worst" element at every term. Instead, we examine the competitive ratio of the algorithm, that is, how much *worse* it is than the optimal offline algorithm (this is an algorithm that knows the entire sequence).

Consequently, we define the *competitive ratio* of an online algorithm as the smallest number c such that

$$ALG(\sigma) \leq c \cdot OPT(\sigma) + \alpha,$$

where $ALG(\sigma)$ is the cost of the online algorithm and $OPT(\sigma)$ is the cost of the offline algorithm, and α is a constant that does *not* depend on $|\sigma|$. An algorithm is *strictly c-competitive* if $\alpha = 0$. When we do the analysis, we typically try to find c by computing the ratio $\frac{ALG(\sigma)}{OPT(\sigma)}$ for an arbitrary sequence σ. The competitive ratio of an algorithm gives us one tool to analyze the efficiency of an online algorithm.

Now, we formally define the List Access Problem.

The List Access Problem Given an unordered list with ℓ elements, and a length m sequence of requests $\sigma = \sigma_1 \sigma_2 \ldots \sigma_m$ for elements in the list, an algorithm must retrieve each element σ_i in order. If element x is in position i in the list, then the cost of finding x and retrieving it is i, i.e. $FIND(x) = i$. If an element x is not in the list, then $FIND(x)$ costs $\ell + 1$. After a $FIND$ operation on an element x, an algorithm may move x forward — this operation is called a *free transposition.* Alternatively, an algorithm may transpose two adjacent items in the list for cost 1, which is called a *paid transposition.*

Typically, the list access problem is modeled as in Figure 1. We present a detailed example of the costs associated with each operation in the list access problem in Example 2.1.

$$\sigma = 1\,5\,3\,8\,7\,8\,7\,7\,4, \ldots$$

3	8	7	4	1	5	2	10	6	9

Figure 1: In the list access problem, the list is typically pictured as a length ℓ array with an associated request sequence σ. See Example 2.1 for a more elaborate example involving the actual operations. The double bar signifies the start of the list.

When we analyze algorithms for the list access problem, we assume that each request σ_i is for an element in the list. Intuitively, it makes sense that we can assume this without loss of generality because both ALG and OPT will both pay $\ell + 1$ for that access. Consequently, if f is the total cost in ALG and OPT for all requests to elements not in the list, then we show that if ALG is c-competitive without these requests, ALG is still c competitive with them.

Lemma. *If ALG is c-competitive on a request sequence without requests to elements not in the list, then ALG is c-competitive on a request sequence with requests to elements not in the list.*

Proof. If the list has length ℓ and there are k requests to elements not in the list, then both ALG and OPT must pay $k(\ell + 1)$ for each of those requests. If σ_- is a request sequence without requests to elements not in the list, and σ_+ is σ_- with the k requests to elements not in the list, then if ALG is c-competitive on σ_-, we have

$$ALG(\sigma_-) \leq c \cdot OPT(\sigma_-) + \alpha.$$

Now, $ALG(\sigma_+) = ALG(\sigma_-) + k(\ell + 1)$ and likewise $OPT(\sigma_+) = OPT(\sigma_-) + k(\ell + 1)$. If we add $k(\ell + 1)$ to the previous inequality, we get

$$ALG(\sigma_-) + k(\ell + 1) \leq c \cdot OPT(\sigma_-) + \alpha + k(\ell + 1).$$

Since c is always greater than 1 we can replace $k(\ell + 1)$ on the right with $c \cdot k(\ell + 1)$, and we have

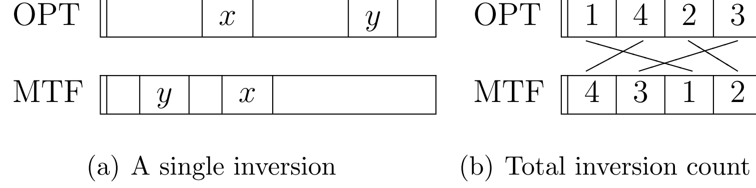
$$ALG(\sigma_+) \leq c \cdot OPT(\sigma_+) + \alpha,$$

or that ALG is c-competitive with requests to elements not in the list. □

We now use the result of this lemma to analyze the MTF algorithm for the list access problem.

2.1 MTF is 2-competitive

Before we can begin the analysis that shows MTF is 2-competitive, we have to define the MTF algorithm! The MTF, or Move-To-Front, algorithm is simple. After a FIND operation on an element x, move x to the front of the list using a free transposition. The MTF algorithm never engages in paid transpositions. Example 2.2 shows how MTF handles a request sequence.



(a) A single inversion (b) Total inversion count

Figure 2: In the first figure, we illustrate an inversion of two elements y and x. In the second figure, we show how the elements are permuted between the two lists. If we count the number of intersections between the lines, we count the number of inversions. Hence, there are three inversions in the second figure. The non-inverted elements are $(1, 2)$, $(2, 4)$, and $(3, 4)$.

Theorem 2.1. *MTF is 2-competitive.*

Proof. With the definition of the MTF algorithm, and the previous lemma, we can show that MTF is 2-competitive by analyzing a list with ℓ element, over a request sequence σ without accesses to elements not in the list. Interestingly, we do *not* need to know the optimal offline algorithm, OPT. To prove this result we use a potential function argument. A potential function Φ is a function of some aspect of the problem such that at the start of the algorithm, $\Phi = 0$, and at the end of the algorithm $\Phi \geq 0$. If we have operations $1, \ldots, f$, then we can get an upper bound on the cost of the algorithm by examining the *amortized cost*

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \qquad 1 \leq i \leq f,$$

where c_i is the true cost of operation i, and Φ_i is the potential after operation i. Notice that the total amortized cost is $\sum_i \hat{c}_i = \sum_i (c_i + \Phi_i - \Phi_{i-1}) = \sum_i c_i + \sum_i (\Phi_i - \Phi_{i-1}) = \sum_i c_i + \Phi_f$, where the last equality is because we had a telescoping sum. Because $\Phi_f \geq 0$ according to the definition of the potential function, we have that $\sum_i \hat{c}_i \geq \sum_i c_i$, or that the sum of all $\hat{c}_i$ upper bounds the actual cost of the algorithm.

In the following analysis, we will use the potential function

$$\Phi = \text{the number of } \textit{inversions} \text{ between } MTF \text{ and } OPT.$$

Formally, an *inversion* is an *ordered* pair of items (y, x) such that y precedes x in MTF's list and x precedes y in OPT's list. In Figure 2 we graphically illustrate an inversion and provide a helpful way to count the total number of inversions.

With the potential function Φ and the definition of inversion, we can now show that MTF is 2-competitive. Let $\sigma = \sigma_1 \sigma_2 \ldots \sigma_m$. On $\sigma_i, 1 \leq i \leq m$, MTF finds σ_i in the list and moves it to the front, whereas OPT makes a series of 0 or more paid transpositions, finds σ_i, and may make a free transposition. We will show that on each $\sigma_i, \hat{c}_i \leq 2\text{OPTCOST}_i$ where $\hat{c}_i$ is the amortized cost of MTF, and $OPTCOST_i$ is the cost for all of OPT's work.

Recall $\hat{c}_i = c_i + \Delta\Phi = c_i + \Delta\Phi_{\text{MTF}} + \Delta\Phi_{\text{OPT}}$, where c_i is the cost of MTF to find element σ_i, $\Delta\Phi_{\text{MTF}}$ is the change in potential due to MTF moving x to the front, and $\Delta\Phi_{\text{OPT}}$ is the change in potential due to moves by OPT. Let $\sigma_i = x$. If x is in position k, then $c_i = k$. Let v be the number of elements in front of x in MTF's list, but after x

in OPT's list. Since x is at position k, then there are $k-1-v$ other elements in front of x in MTF's list. Hence, $\Delta\Phi_{\text{MTF}} = -v + (k-1-v)$, because we remove inversions due to the v elements which are "realigned," but introduce inversions in the $k-1-v$ other elements. However, notice that if x is in position j in OPT's list, then $k - v \leq j$, because v was everything in front of x in MTF's list, but not in OPT's. So, what is left $(k-v)$ can be no larger than what is left in front of x in OPT's list.

Now, if OPT makes any paid transposition, those, at worst, increase the potential by 1 for each transposition. If OPT moves x to the front, then this just decreases the potential because we remove all inversions due to x. If

$$\text{OPTCOST}_i = \text{OPT_FIND}_i + \text{OPT_PAID}_i,$$

then because $k - v \leq j$, $\text{OPT_FIND}_i \leq j$ — thus we have $k - v$ as a lower bound on the cost for OPT_FIND_i. Also, $\Delta\Phi_{\text{OPT}} \leq \text{OPT_PAID}_i$.

At this point, we have all the pieces we need.

$$\begin{aligned}
\hat{c}_i &= c_i + \Delta\Phi_{\text{MTF}} + \Delta\Phi_{\text{OPT}} \\
&\leq k + (-v) + (k-1-v) + \text{OPT_PAID}_i \\
&= 2(k-v) - 1 + \text{OPT_PAID}_i \\
&\leq 2\text{OPT_FIND}_i + 2\text{OPT_PAID}_i \\
&= 2\text{OPTCOST}_i.
\end{aligned}$$

Thus,

$$MTF(\sigma) = \sum_i c_i \leq \sum_i \hat{c}_i \leq \sum_i 2\text{OPTCOST}_i = 2\sum_i \text{OPTCOST}_i = 2OPT(\sigma),$$

or MTF is 2-competitive. □

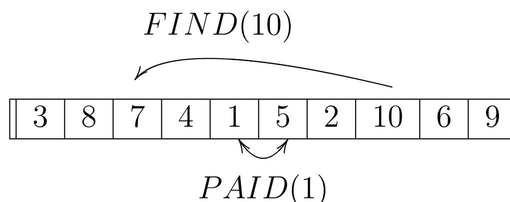
2.2 Online Deterministic List Access Algorithms are 2-competitive

Although we don't formally prove the lower bound for the deterministic List Access problem, we do sketch the result.

Claim. *For the list accessing problem with a list of ℓ items, any deterministic online algorithm has a competitive ratio of at least* $2 - \frac{2}{\ell+1}$.

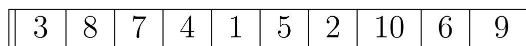
The proof proceeds using an adversary that requests the last element in ALG's list at every step, so ALG costs ℓm on a request sequence with m elements. Then, it uses an averaging argument over offline algorithms that permute the list initially and then never reorder. With this setup, the cost of accessing any item in the list (over all $\ell!$ algorithms) is $\frac{\ell(\ell+1)}{2} \cdot (\ell-1)!$. Taking the average over all algorithms for m requests shows that an algorithm cannot be better than $2 - \frac{2}{\ell+1}$-competitive against the average of a set of offline algorithms, and cannot be any more competitive against a true optimal offline algorithm.

Example 2.1 Starting with the list from Figure 1, we demonstrate the cost of various operations and the subsequent state of the list. After a $FIND(10)$ operation, an algorithm may move the element 10 to any forward position. At any point, any algorithm may swap two adjacent elements with a paid transposition, such as $PAID(1)$ which swaps 1 and 5.

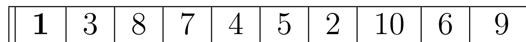


Example 2.2 When MTF services this request sequence σ there are no paid transpositions, all the cost is from the FIND operations. The bold number at the head of the list is the element that was moved. In this example, $MTF(\sigma_{1-5}) = 5 + 6 + 3 + 4 + 5 = 23$, we leave computing the total cost of MTF on σ to the reader.

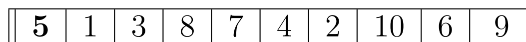
$$\sigma = 1\,5\,3\,8\,7\,8\,7\,7\,4.$$



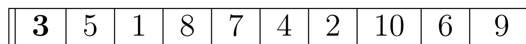
↓ $FIND(1) = 5$



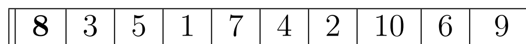
↓ $FIND(5) = 6$



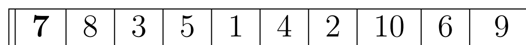
↓ $FIND(3) = 3$



↓ $FIND(8) = 4$



↓ $FIND(7) = 5$



3 Randomized Competitive Analysis and the BIT Algorithm

Randomized competitive analysis is more complicated than traditional, deterministic competitive analysis due to the uncertainty in the randomized algorithm. Instead of examining the worst-case performance over all input sequences σ, we examine the *expected value* of the algorithm over each input sequence σ. Thus, a randomized online algorithm is c-competitive if

$$E[ALG(\sigma)] \leq c \cdot E[OPT(\sigma)] + \alpha.$$

In this formula, we need to take the expected value of the optimal algorithm as well, because different types of analysis give the optimal algorithm and associated adversary different properties.

3.1 Adversary Models

Another distinction of randomized online algorithms compared to deterministic ones is the range of adversary models available. For deterministic algorithms, an adversary cannot gain any new knowledge by changing the input sequence in response to the online algorithm because the algorithm is deterministic! However, in the randomized case, the adversary could gain valuable information by watching the online algorithm's behavior. Therefore, there are three main adversary models studied: oblivious, adaptive online, and adaptive offline.

- The **oblivious** adversary acts much in the same way as the traditional adversary. The entire input sequence is created in advance and the adversary may service it in the optimal offline manner. The adversary knows the structure of the online algorithm (including any probability distributions used), but none of its random choices.
- The **adaptive online** adversary gets to create the input sequence one item at a time, while observing the choices made by the online algorithm. Therefore, it can create the most heinous request possible at each step. However, it must then service the request online as well, with no knowledge of future requests.
- The **adaptive offline** adversary creates the input sequence in the same manner as the adaptive online adversary. However, it gets to service the request sequence offline, after creating it all.

Notice that an adaptive online adversary could simulate an oblivious adversary by completely ignoring the online algorithm's behavior, and servicing all the requests exactly as the oblivious adversary does. Also, an adaptive offline adversary could simulate an adaptive online algorithm by servicing the requests (offline) exactly as the online adversary would. Therefore, we see that the oblivious adversary is the weakest adversary, and the adaptive offline adversary is the strongest.

3.2 The BIT Algorithm

The BIT algorithm uses a very simple random addition to the MTF algorithm. When the BIT algorithm begins, it allocates a bit for each element in the list and randomly

3	8	7	4	1	5	2	10	6	9
0	1	1	0	0	0	1	0	1	1

Figure 3: A visual display of the data structure the *BIT* algorithm uses. The values below the elements in the list are the random bits, $b(x)$.

initializes this bit to 0 or 1. For an element x, let $b(x)$ be the current value of its bit. Thus, *BIT* allocates a total of ℓ random bits on initialization. Figure 3 shows how this arrangement might work. On an access for element x, *BIT* first searches for x. After *BIT* finds x, it complements $b(x)$ and if $b(x) = 1$, then *BIT* moves element x to the front of the list, and otherwise leaves it in its position. Example 3.1 demonstrates how *BIT* handles a sequence of requests.

One interesting aspect of the *BIT* algorithm is that it is *barely random*, that is, the number of random bits it uses is bounded by the size of the list. Even on a request sequence with thousands of requests, the *BIT* algorithm will only use the ℓ initial random bits. In contrast, the *RMTF* algorithm, which moves an element to the front of the list based on a random bit for each request, uses m bits where m is the length of the request sequence.[1] In fact, the *BIT* algorithm was the first barely random algorithm to achieve a competitive ratio smaller than a deterministic algorithm [4].

3.3 *BIT* is 7/4-competitive!

In this section, we prove the amazing result that *BIT* is 7/4-competitive using a potential argument (similar to how we showed *MTF* was 2-competitive). In the next section, we will show the *COMB* algorithm is 8/5-competitive, and as a by-product, show that *BIT* is 7/4-competitive a different way. Hence, if you only want to learn about the *COMB* algorithm, feel free to skip this section of our paper.

Before we can prove that *BIT* is 7/4-competitive, we must first prove that the actual bits, i.e. $b(x)$'s, stay independent throughout the computation.

Lemma. *The value of* $b(x)$ *at any request is equally likely to be* 0 *or* 1 *(i.e. its value is uniformly distributed in* $\{0, 1\}$*).*

Proof. Because *BIT* initially chose each $b(x)$ randomly with a uniform distribution, the values are initially equally likely to 0 or 1. In the implementation, $b(x)$ actually counts the accesses to $x \mod 2$. At any request, there are equally likely to be an even or odd number of accesses to x, hence $b(x)$ stays equally likely to be 0 or 1 at any point. □

Now, we get to the first big theorem.

[1] In addition to using more random bits than the *BIT* algorithm, the *RMTF* algorithm is amazingly only 2-competitive [3].

Theorem 3.1. *BIT is 7/4-competitive against oblivious adversaries.*

Proof. First, we note that the value of $b(x)$ for any x is unknown to the adversary because the adversary is oblivious. Thus, we can see that BIT might be able to beat the deterministic bound because the adversary cannot deterministically foil it. In the remainder of the proof, we first introduce the potential function $\Phi(x)$. We then divide the request sequence σ into a series of requests for elements and paid transpositions (by the OPT algorithm). For these two cases, we show that $E[BIT(\sigma)] \leq 7/4OPT(\sigma)$, and finally conclude that BIT is 7/4-competitive.

Before defining the potential function, we must define an inversion. An *inversion* is an *ordered* pair of items (y, x) such that y precedes x in BIT's list and x precedes y in OPT's list. This definition is analogous to the definition used in the proof that MTF was 2-competitive. For this problem, we call an inversion a *type 1 inversion* if $b(x) = 0$, and a *type 2 inversion* if $b(x) = 1$. In fact, the type of an inversion counts the number of accesses before that element will be moved to the front of the list. If ϕ_1 is the number of type 1 inversions and ϕ_2 is the number of type 2 inversions, then to show BIT is 7/4-competitive, we use the potential function

$$\Phi = \phi_1 + 2\phi_2.$$

With this potential function, we divide the cost of the BIT and OPT algorithms into a series of $FIND$ operations and paid transpositions. If operation i is a FIND operation, then the amortized cost is $\hat{c}_i = c_{\text{FIND}} + \Delta\Phi$. Let x be the element sought in the $FIND$ operation and let R be a random variable that counts the number of inversions of the form (y, x). In other words, R counts the number of items in front of x in BIT's list, but not in OPT's list. Finally, let k be the cost of OPT on this particular find operation. Then, $c_{\text{FIND}} = k + R$.

Now, let $\Delta\Phi = A + B + C$ where A is a random variable counting the change in potential due to new inversions, B is a random variable counting the change in potential due to the removal of old inversions, and C is a random variable counting the change in potential due to inversions changing type. These scenarios account for all the change in potential. We now show that $E[B+C] \leq -R$. Consider what happens if BIT does not move x to the front of the list. Then all the inversions involving x were type 2 inversions, so when BIT changed $b(x)$ to 0, all these inversions change to type 1 which decreases the potential by R. If instead, BIT moves x to the front of the list, then all R inversions will disappear. There could also be additional inversions removed when OPT moves x forward, but these only decrease the potential further. Since only one of these two cases can occur on each $FIND$ operation, $E[B + C] \leq -R$.

To complete the proof, we need to calculate $E[A]$. Only OPT can create new inversions when it moves x forward. If the element x was originally at position k in OPT's list, let $k' \leq k$ be the new position of x after OPT moves x forward. Since OPT is oblivious, it doesn't know if BIT moved x forward or not. Hence, let's look at each element $z_1, z_2, \ldots, z_{k-1}$ in OPT's list in front of x. The algorithms introduce a new inversion if there is an element z_i where z_i precedes x in BIT's list, and either BIT or OPT move x in front of z_i. Let Z_i be a random variable that counts the change in potential due to each pair of elements (x, z_i). To help ease the confusion, we have provided Figure 4 which we hope will make the following relationships more clear.

If $b(x) = 0$, then BIT will move x to the front of the list. In the worst case, this creates a new inversion of type $1 + b(z_i)$ for each of the $k' - 1$ elements in front

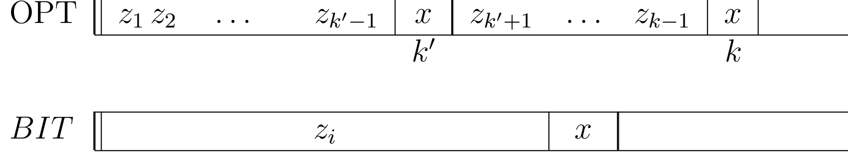


Figure 4: In this figure, we have omitted the values of the bits, since they are random variables unknown to the oblivious adversary. In the top list here, we are simulating OPT's movement of x from position k to a position k'. In the bottom list, we are visualizing BIT's list. The key things to notice here for the analysis. If BIT leaves x in place, then we create new inversions with $z_{k'+1}, \ldots, z_{k-1}$. Alternatively, if BIT moves x to the front, then we create new inversions with $z_1, \ldots, z_{k'-1}$.

the the position where OPT moved x. In other words, the only elements that might contribute to creating new inversions are those in front of x's new position in OPT. Hence, $Z_i \leq 1 + b(z_i)$ for $1 \leq i \leq k' - 1$. Since the lists are not changed past the k' position, we have that $Z_i \leq 0$ for $k' \leq i \leq k$.

If $b(x) = 1$, then BIT will not move x and $b(x)$ changes to 0. This action could potentially create a new inversion because OPT moved x forward. However, the elements involved in the inversion are those between the new position of x in OPT and the old position of x in OPT. Hence, this could create inversions of type 1 (since $b(x) = 0$) with $z_{k'}, \ldots, z_{k-1}$, i.e. $Z_i \leq 1$ for $k' \leq i \leq k - 1$. Since the lists are equivalent prior to k' we have that $Z_i \leq 0$ for $1 \leq i \leq k' - 1$.

We can enumerate the expected cost of A as

$$E[A] = \sum_{i=1}^{k} E[Z_i] \leq \sum_{i=1}^{k'-1} \frac{1}{2}\left(1 + E[b(z_i)]\right) + \sum_{i=k'}^{k-1} \frac{1}{2} \cdot 1 \leq \frac{3}{4}(k-1) \leq \frac{3}{4} OPT.$$

In the second step, half the time Z_i, $1 \leq i \leq k' - 1$ is positive, and the other half, it is negative. Likewise for the other portion of the list. $E(b(z_i))] = \frac{1}{2}$ by the previous lemma.

The only remaining case is that OPT performs a paid transposition. If OPT performs a paid transposition, then OPT pays 1. In the worst case, this introduces a new inversion between OPT and BIT. Since BIT has 0 cost (it never does paid transpositions), the amortized cost of the operation is just the change in potential, which is the expected increase in potential due to the new inversion. This expectation is just $\frac{3}{2}$ since half the time it increases 1 (type 1 inversion) and the other half it increase by 2 (type 2 inversion).

Hence, for all operations, $\hat{c}_i$ performed by BIT and OPT, we have that the amortized cost of BIT is less than $\frac{7}{4}$ times the amortized cost. Since this holds for each operation, it holds for all operations, and we know the amortized cost is an upper bound on the real cost. Hence,

$$E[BIT(\sigma)] \leq \frac{7}{4} OPT(\sigma). \qquad \square$$

Example 3.1 Here we see how BIT behaves differently from the MTF algorithm. Notice that element 8 is not moved to the front when it is accessed.

$$\sigma = 1\,5\,3\,8\,7\,8\,7\,7\,4.$$

3	8	7	4	1	5	2	10	6	9
0	1	1	0	0	0	1	0	1	1

$\downarrow \quad FIND(1) = 5$

1	3	8	7	4	5	2	10	6	9
1	0	1	1	0	0	1	0	1	1

$\downarrow \quad FIND(5) = 6$

5	1	3	8	7	4	2	10	6	9
1	1	0	1	1	0	1	0	1	1

$\downarrow \quad FIND(3) = 3$

3	5	1	8	7	4	2	10	6	9
1	1	1	1	1	0	1	0	1	1

$\downarrow \quad FIND(8) = 4$

3	5	1	8	7	4	2	10	6	9
1	1	1	0	1	0	1	0	1	1

4 The Pairwise Property and the $COMB$ Algorithm

The eventual proof that $COMB$ is 8/5-competitive depends upon a series of results. The first of these is a particular property of some algorithms called the *pairwise property.* Essentially, if an algorithm has the pairwise property, then we can do all the analysis for the algorithm on request sequences for only two elements in a two item list. This greatly simplifies the analysis because the optimal behavior of an algorithm is often easy to determine in these cases. In contrast, the previous two results did not explicitly know the behavior of the optimal algorithm.

4.1 The Pairwise Property

Before we state the definition of the pairwise property, we introduce a new *cost model* and a set of new notation. First, until now we have used the *full cost model* for the list access problem. In this cost model, a $FIND$ operation on an element x in position i costs i. In the *partial cost mode* a $FIND$ operation for an element x in position i costs $i-1$. One way of viewing this difference is if we make each element before x pay for impeding access to x because the algorithm must do some sort of comparison. In the full cost model, x must pay for comparing to itself. In the partial cost model, we relax this restriction and do not require x to pay for itself. We now show that analysis under the partial cost model is sufficient.

Lemma. *If ALG is c-competitive on a request sequence in the partial cost model, then ALG is c-competitive in the full cost model.*

Proof. In essence, this proof is exactly like the proof that ALG is c-competitive without accesses to items not in the list from the first section. If the request sequence has m FIND operations, then the difference in cost between an algorithm in the partial cost model and in the full cost model is just m, i.e. $ALG_f(\sigma) = ALG_p(\sigma) + m$ where $ALG_f(\sigma)$ is the cost of ALG in the full cost model and $ALG_p(\sigma)$ is the cost of ALG in the partial cost model. Likewise, $OPT_f(\sigma) = OPT_p(\sigma) + m$. Hence,

$$ALG_p(\sigma) \leq c \cdot OPT_p(\sigma) + \alpha.$$

Now, add m to the left side and $c \cdot m$ to the right side, which is alright since $c \geq 1$,

$$ALG_p(\sigma) + m \leq c \cdot OPT_p(\sigma) + c \cdot m + \alpha,$$

$$ALG_f(\sigma) \leq c \cdot OPT_f(\sigma) + \alpha. \qquad \square$$

From this point forward, all analysis will be done with the partial cost model!

Now, we introduce ALG^* notation. Let $ALG^*(x, j)$ be 1 when item x precedes the element σ_j (i.e. the jth element requested in σ) in the list on the jth request, and 0 otherwise, including when $x = \sigma_j$. $ALG^*(x, j)$ could be thought of as the charge on element x for impeding access to element σ_j in the partial cost model. If the request

sequence σ has length m and L is the set of elements in the list, then using ALG^* we can write the total cost of ALG,

$$ALG(\sigma) = \sum_{j=1}^{m} \sum_{x \in L} ALG^*(x, j).$$

Next, we interchange the order of the summation. Instead of summing over each request, we sum over each element and look at its total cost for all requests.

$$ALG(\sigma) = \sum_{x \in L} \sum_{j=1}^{m} ALG^*(x, j).$$

Observe that we can enumerate the set $\{1, \ldots, m\}$ as $\{j \mid \text{there exists } y \text{ such that } \sigma_j = y\}$. Thus, we can transform the cost of ALG into

$$ALG(\sigma) = \sum_{x \in L} \sum_{y \in L} \sum_{j \mid \sigma_j = y} ALG^*(x, j).$$

While this transformation seems strange, for each element x we are summing the total cost that x incurs impeding access to each element y. Now, if we look at pairs of elements x and y, we can restate the cost.

$$ALG(\sigma) = \sum_{\{x,y\} \subseteq L} \; \sum_{j \mid \sigma_j \in \{x,y\}} \left(ALG^*(x, j) + ALG^*(y, j) \right).$$

At this point, we have a very odd sum. For each pair of elements x and y, we compute the cost due to x impeding access to y, and the cost due to y impeding access to x. Because this is in the partial cost model, in $ALG^*(x, j) + ALG^*(y, j)$, one of the two values will always be 0! Nevertheless, this equation is a valid method to compute the total cost of the algorithm.

If we define

$$ALG_{xy}(\sigma) = \sum_{j \mid \sigma_j \in \{x,y\}} \left(ALG^*(x, j) + ALG^*(y, j) \right),$$

then the cost of the algorithm simplifies to

$$ALG(\sigma) = \sum_{\{x,y\} \subseteq L} ALG_{xy}(\sigma).$$

To try and get some intuition about the quantity $ALG_{xy}(\sigma)$ see Example 4.1.

We now state the pairwise property. An algorithm ALG has the *pairwise property* if

$$ALG_{xy}(\sigma) = ALG(\sigma_{xy}),$$

where $ALG(\sigma_{xy})$ is the cost of ALG on the (boring) two element list of x and y over the arbitrarily long sequence of requests to x and y from σ. In other words, if we project the list and the request sequence onto the items x and y (i.e. remove everything else from the list and σ), then $ALG(\sigma_{xy})$ is the cost of ALG on the projected list and request sequence. We now characterize the pairwise property.

Lemma. *An algorithm satisfies the pairwise property if and only if for every request sequence σ, when ALG serves σ, the relative order of every two elements x and y in the list is the same as their relative order when ALG serves σ_{xy}.*

Proof. In the forward direction, we show that if for every request sequence σ, when ALG serves σ, the relative order of every two elements x and y is the same as their relative order when ALG serves σ_{xy}, then ALG satisfies the pairwise property. Consider $ALG_{xy}(\sigma) = \sum_{j|\sigma_j \in \{x,y\}} (ALG^*(x,j) + ALG^*(y,j))$. Since the relative order of the items is always the same, $ALG^*(x,j) = 1$ when y is in front of x in L_{xy} and likewise, $ALG^*(y,j) = 1$ when y is in front of x in L_{xy}. Hence, $ALG_{xy}(\sigma) = ALG(\sigma_{xy})$.

In the reverse direction, we show that if an algorithm satisfies the pairwise property, then when ALG serves σ, the relative order of every two elements x and y is the same as their relative order when ALG serves σ_{xy}. If, by way of contradiction, we assume this isn't true, then there exists a step when the relative order of x and y in the list and in σ_{xy} are out of sync. Let σ' be all the requests prior to the step that is out of sync. Consider the next request after σ', request σ_k. Since the relative orders are out of sync, one of the partial costs ($ALG(x,k) + ALG(y,k)$ or the cost of σ_k) will be 1 and the other will be 0. However, then the algorithms do not have the pairwise property since we can end σ at this step. Thus we have our contradiction. □

We now show that BIT has the pairwise property.

Lemma. *BIT has the pairwise property.*

Proof. Since BIT is a randomized algorithm, we need to show that the pairwise property holds when the bits associated with x and y are shared. Thus, since the behavior of BIT is independent of the order of the list (e.g. the front of the list does not depend on the order), BIT has the pairwise property. □

4.2 The $TIMESTAMP$ Algorithm

On a request for item x in the list, this (complicated) algorithm moves it directly in front of the first item in the list that was accessed at most once since the last request for x. If there is no such item, or x has not been requested before, do nothing. First, we notice that $TIMESTAMP$ is deterministic. Also, $TIMESTAMP$ is 2-competitive (as will be shown later, in the analysis of $COMB$), and so its competitive ratio is tight (as is MTF's).

Lemma. *After the $TIMESTAMP$ algorithm has served a request sequence σ, element x is before element y if and only if the sequence σ_{xy} terminates in the subsequence xx, xyx, or xxy, or if x was before y initially and y was requested at most once.*

Proof.

- ($\Leftarrow$) Notice that in the cases where σ_{xy} ends in xx or xyx, y is requested at most once between the final two x's. Therefore, x must be moved in front of y at the end. If σ_{xy} ends in xxy, then x is requested twice in a row, which moves it in front of y, and at least twice between the final two y's in the sequence, if there are two. Therefore, the final request to y does not move it in front of x, and so x is before y. If y is requested at most once, then it will not be moved in front of x ever, and so if x starts before y, it will also end before y.

- (⇒) If element x is before element y in the list after $TIMESTAMP$ services σ_{xy}, then one of two things must have happened: either y was requested at most once between the final two requests for x, or – if there were fewer than 2 requests for x – x started before y and there were no more than 1 request for y in the sequence. So we see that x ending before y implies that σ_{xy} ends in xx, xyx, or xxy, or contains at most one y, with x starting before y in the list. This concludes the proof.

□

Lemma. *$TIMESTAMP$ has the pairwise property.*

Proof. This lemma follows from the previous lemma when we apply it to every step of the request sequence. Note that in the previous lemma, we specified the relative position of x and y solely based on the access pattern to just elements x and y. Hence, the relative position of x and y does not depend on the other elements in the request sequence or the list. □

4.3 The $COMB$ Algorithm

The $COMB$ algorithm is simple. With probability 4/5 serve the request sequence with BIT, with probability 1/5 serve the request sequence with $TIMESTAMP$. We would like to note that $COMB$ does not alternate between BIT and $COMB$, when $COMB$ "initializes," it picks either BIT or $COMB$ and serves the entire request sequence with that algorithm. Intuitively, the $COMB$ algorithm might be better because the worse case analysis for BIT and $TIMESTAMP$ might be different, thus an adversary could fool BIT or $TIMESTAMP$ but an oblivious adversary could not succeed against $COMB$.

Before proving that $COMB$ is 8/5-competitive, we need a lemma about BIT.

Lemma. *After serving the request sequence xyx, or the sequence yx on a list where x was before y initially, x is before y afterward with probability 3/4. The same theorem holds with x and y interchanged (i.e. y before x, yxy and xy).*

Proof. We prove this using an explicit case analysis. If x is before y and BIT serves the sequence xyx, the list will have the following orders.

		$b(y)$	
		0	1
$b(x)$	0	yx	xy
	1	xy	xy

If x is before y and BIT serves the sequence yx, then the list will have the following orders.

		$b(y)$	
		0	1
$b(x)$	0	xy	xy
	1	yx	xy

The lemma follows since the value of $b(x)$ and $b(y)$ are independent, and in 3 of 4 cases each time, x precedes y after the sequence is served. □

With this lemma, we can now prove that $COMB$ is 8/5-competitive.

Theorem 4.1. *$COMB$ is 8/5-competitive against oblivious adversaries.*

Proof. Since BIT and $TIMESTAMP$ both have the pairwise property, $COMB$ has the pairwise property. Thus, we can apply the pairwise property theorem and it suffices to show that $COMB$ is 8/5 competitive against an optimal algorithm on 2 element lists.

If σ_{xy} is an arbitrary request sequence for elements x and y, then we can uniquely partition σ_{xy} into subsequences terminated by two requests to the same item. First note that in both BIT and $TIMESTAMP$, if an element is requested twice in a row it will always be moved to the front of the list. If we assume that element x is first in the list, then a subsequence may have the form $x^l yy$, $x^l(yx)^k yy$, or $x^l(yx)^k x$ where $l \geq 0$ and $k \geq 1$. If this subsequence ends in xx, then in both BIT and $TIMESTAMP$, x will be first in the list. Hence, the next subsequence will be of the same form. If the subsequence ends in yy, then y will be first in the list, and the next subsequence will be of the form $y^l xx$, $y^l(xy)^k$, or $y^l(xy)^k y$, where $l \geq 0$ and $k \geq 1$. This partitioning is unique because one of the list forms must match at each step. If the request sequence does not end with two accesses to the same element, we simply repeat the last request so we can partition the sequence, this adds a negligible cost.

Since the two subsequence forms are equivalent if we simply replace x with y, we can restrict the analysis to only one case. Then, the following costs hold on a list where x is before y in the partial cost model.

request sequence	BIT	$TIMESTAMP$	OPT
$x^l yy$	$\frac{3}{2}$	2	1
$x^l(yx)^k yy$	$\frac{3}{2}k+1$	$2k$	$k+1$
$x^l(yx)^k x$	$\frac{3}{2}k+\frac{1}{4}$	$2k-1$	k

We show these costs by analyzing each algorithm independently. First, when BIT serves $x^l yy$, the expected cost is 3/2 because all accesses to x cost 0, and y will move forward after the first access with probability 1/2. After the accesses to the initial xs in $x^l(yx)^k yy$, the first access to y costs 1. The subsequence access to x costs 1/2 because half the time y moves to the front. At this point, we have seen the sequence yx on a list where x preceded y, hence x will be at the front of the list with probability 3/4 by our previous lemma. Subsequently, we assume that each alternating access to x and y has expected cost 3/4 by the previous lemma.[2] The final expected cost of the two ys is $3/4+1/4$. Thus, the total expected cost is $1+1/2+(3/4+3/4)(k-1)+1 = \frac{3}{2}k+1$. A similar argument shows the expected cost for BIT in the last case is $\frac{3}{2}k+\frac{1}{4}$.

The cost of the $TIMESTAMP$ algorithm is easier to find because $TIMESTAMP$ is deterministic. Accessing the sequence $x^l yy$ has cost 2 because y is not moved to the front until the second access to y according to the lemma about $TIMESTAMP$. On the sequence $x^l(yx)^k yy$ the first access to y has cost 1, and the subsequent access to x costs 0. All further requests to x and y have cost 1 because they constantly move to the front of the list. Finally, the two ys at the end cost 1 total as the y is moved to the front after the first access. Hence, $TIMESTAMP$ costs $1+0+2(k-1)+1 = 2k$. By the same

[2]At this step, you may be concerned that the previous lemma doesn't apply because x doesn't have to be before y when serving x after an arbitrary yx. However, the expected cost from the lemma is the worst case, so we get an upper bound.

logic, $TIMESTAMP$ costs $2k-1$ on the final subsequence. This analysis, combined with the subsequent analysis of OPT, shows that $TIMESTAMP$ is 2-competitive as previously claimed.

The optimal algorithm has cost 1 on the first subsequence because the y is immediately moved to the front after the accesses to x. On the second subsequence, OPT must pay 1 for the k alternations and then 1 for the access to the two ys at the end. On the final subsequence, it is optimal just to keep x at the front of the list for a total cost of k, i.e. just pay for the ys.

Since the expectation is a linear operator, we just multiply the probability of selecting BIT by its expected cost and do the same for $TIMESTAMP$. Thus, on the first subsequence, $COMB$ has cost $\frac{4}{5} \cdot \frac{3}{2} + \frac{1}{5} \cdot 2 = \frac{8}{5}$. The second subsequence has cost $\frac{8}{5}k + \frac{4}{5}$ and the third has cost $\frac{8}{5}k$. Computing the ratios against OPT shows that $COMB$ is 8/5-competitive. $\square$

Example 4.1 The value $ALG_{xy}(\sigma)$ is slightly strange at first. In this example we provide a concrete example of this quantity when ALG is the MTF algorithm. Also, we compute the value of $ALG(\sigma_{xy})$ to build intuition about the pairwise property. On the right of each equation, we have provided the state of the list immediately prior to that operation. For example, on the second line, we show the list immediately prior to σ_2. If a particular operation does not contribute in the final sum, we don't list a value for that operation. Since MTF has the pairwise property, notice that the relative position of the two elements in each list is the same. Hence, $MTF_{24}(\sigma) = MTF(\sigma_{24})$.

In this example, we will use MTF as our algorithm.
Let $\ell = 4$ and

$$\sigma = 2\,1\,1\,4\,3\,1\,4\,4\,3\,2.$$

Then,

$MTF_{24}(\sigma) = \sum_{j \in \{1,4,7,8,10\}}$	$ALG^*(2,j) + ALG^*(4,j)$		$MTF(\sigma_{24}) =$	$MTF(2,4,4,4,2)$	
$= 0$	1 2 3 4	σ_1	$= 0$	2 4	σ_1
	2 1 3 4	σ_2			
	1 2 3 4	σ_3			
$+ 1$	1 2 3 4	σ_4	$+ 1$	2 4	σ_2
	4 1 2 3	σ_5			
	3 4 1 2	σ_6			
$+ 0$	1 3 4 2	σ_7	$+ 0$	4 2	σ_3
$+ 0$	4 1 3 2	σ_8	$+ 0$	4 2	σ_4
	4 1 3 2	σ_9			
$+ 1$	3 4 1 2	σ_{10}	$+ 1$	4 2	σ_5
	2 3 4 1			2 4	
$= 2$			$= 2$		

$$MTF_{24}(\sigma) = MTF(\sigma_{24}).$$

Example 4.2 $TIMESTAMP$ is a deterministic algorithm that keeps track of the last two accesses to each item. Here, we see $TIMESTAMP$ running on a particular request sequence σ. The first set of numbers represents the $RECENT$ counter for each element, or $R(x)$. The second set represents the $PREVIOUS$ counter, or $P(x)$. After the first three finds, only $R(x)$ changes. After the element 5 is requested twice, $TIMESTAMP$ moves 5 to the front of the list because it was "most previously." When 3 is eventually requested, 5 had a more previous access, so 3 moves forward, but not in front of 5.

$\sigma = 3\,5\,4\,5\,5\,3$

4	1	5	3	2
0	0	0	0	0
0	0	0	0	0

$\downarrow$ $FIND(3)$
$\downarrow$ $FIND(5)$
$\downarrow$ $FIND(4)$

4	1	5	3	2
3	0	**2**	**1**	0
0	0	0	0	0

$\downarrow$ $FIND(5)$

5	4	1	3	2
4	3	0	1	0
2	0	0	0	0

$\downarrow$ $FIND(5)$

5	4	1	3	2
5	3	0	1	0
4	0	0	0	0

$\downarrow$ $FIND(3)$

5	**3**	4	1	2
5	**6**	3	0	0
4	**1**	0	0	0

5 Conclusion

In this tour of list access algorithms, we have come across a number of really interesting ideas in algorithmic analysis. In the first two proofs, we saw how potential functions can significantly aid the analysis. In the final proof, we spent a lot of time developing the pairwise property so that we could restrict our attention to lists of only two elements. However, there is significantly more research into the list access problem available. Currently, the best known *lower* bound on the competitiveness ratio of a randomized algorithm is 1.50084 due to Ambühl, Gärtner, and von Stengel in 2000 [2]. In fact, Christoph Ambühl was awarded his doctorate in 2002 for a thesis entitled, *On The List Update Problem*. Closing the gap between the best known randomized algorithm $COMB$, $c = 1.6$, and the best known lower bound, $c = 1.50084$, is an area of active research.

For more information on this topic, we highly recommend the sources in the references. With our introduction to the topic, you should find them fairly accessible.

References

[1] Susanne Albers, Bernhard von Stengel, and Ralph Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. *Information Processing Letters*, 56(3):135–139, 1995.

[2] C. Ambühl, B. Gärtner, and B. von Stengel. A new lower bound for the list update problem in the partial cost model. *Theor. Comput. Sci.*, 268(1):3–16, 2001.

[3] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[4] Nick Reingold, Jeffery Westbrook, and Daniel Dominic Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1):15–32, 1994.

[5] D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.