3 Online Algorithms

In this chapter we will consider randomized online algorithms. It is closely related to a script by Susanne Albers [1]. An *online algorithm* processes its input piece by piece in a serial fashion, i.e., in the order that the input is fed into the algorithm, and bases its decisions only on the piece of the input seen so far. In contrast, an *offline algorithm* is given the whole input data from the beginning and is required to output an answer which solves the entire problem at hand. Online algorithms are very useful for scenarios where the entire input is not available a priori simply because future pieces of the input are not known yet.

3.1 Examples

- Ski rental problem: A pair of skies can either be bought for 500 Euros or rented per day for 50 Euros. For how many days should the skies be rented before buying them if it is not known in advance how many days they will be used? An optimal online strategy is to rent the skies until the rental costs are equal to the cost of buying them.
- Money exchange problem: An amount of money needs to be exchanged into a different currency. At which point in time should it be exchanged if it is unknown how the exchange rate develops over time?
- Paging/Caching: Here, the goal is to process (read and/or write) requests to a two-tier storage system which consists of a fast memory of small storage capacity and a slow memory of large storage capacity. A page fault occurs if the requested page is not in the fast memory. In this case it first has to be loaded from the slow memory into the fast memory. If there is not enough room for it in the fast memory, some other page needs to be evicted instead. Which page should be evicted in this case to minimize the number of page faults?
- Distributed systems: Data are to be placed in a distributed system so that the communication cost (i.e., the distance a message needs to travel) for transferring data is minimized. Where should the data be placed, and how should it be copied or replaced in order to minimize the communication cost?
- Scheduling: Jobs with a known processing time arrive one by one, and each job needs to be processed on one of m machines. Many optimization goals are possible here. A classical goal is to minimize the makespan, i.e., the time needed until all jobs have been processed.

3.2 Formal model

An online algorithm A needs to serve some request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(t)$. More precisely, it needs to decide what to do with request $\sigma(i)$ without knowing $\sigma(i+1), \ldots, \sigma(t)$. The goal is to serve the requests in such a way that some given objective function gets minimized or maximized. In order to measure the quality of an online algorithm, competitive analysis is used. Competitive analysis compares the performance of the online algorithm with the performance of a best possible offline algorithm called OPT, i.e., an algorithm that knows σ in advance. The cost of A for some input σ is given by $A(\sigma)$, and the optimal cost is called $\mathrm{OPT}(\sigma)$.

Definition 3.1 *A deterministic online algorithm A is called* c-competitive *if there is a b independent of t so that for all inputs σ,*

$$A(\sigma) \leq c \cdot \mathrm{OPT}(\sigma) + b$$

If A is a randomized algorithm, then $A(\sigma)$ is a random variable, so our goal will be to determine $\mathbb{E}[A(\sigma)]$. Different kinds of adversaries are considered when bounding the competitive ratio of A:

- An *oblivious adversary* has to set σ before seeing the random choices of A.
- An *adaptive adversary* can base its selection of $\sigma(t)$ on the random choices that A has made for $\sigma(1), \ldots, \sigma(t-1)$.

Moreover, one distinguishes between an *adaptive online adversary*, which has to serve its generated input sequences in an online fashion, and an *adaptive offline adversary*, which is allowed to serve its input sequence in an offline fashion (i.e., it can wait with processing the sequence until it is finished).

Definition 3.2 *An algorithm A is called* c-competitive *against an oblivious adversary or adaptive offline adversary if there is a b independent of t so that for all inputs σ,*

$$\mathbb{E}[A(\sigma)] \leq c \cdot \mathrm{OPT}(\sigma) + b$$

A is c-competitive *against an adaptive online adversary ADV if there is a b independent of t so that for all inputs σ,*

$$\mathbb{E}[A(\sigma)] \leq c \cdot \mathbb{E}[\mathrm{ADV}(\sigma)] + b$$

The following properties have been shown [2]:

Theorem 3.3 *For all randomized online algorithms A,*

$$c_{\text{oblivious adversary}} \leq c_{\text{adaptive online adversary}} \leq c_{\text{adaptive offline adversary}}$$

Theorem 3.4 *Let A be a randomized online algorithm that is c-competitive against an adaptive offline adversary. Then there is a c-competitive deterministic algorithm for the problem.*

Theorem 3.5 *Let A be a randomized online algorithm that is c-competitive against an adaptive online adversary. If there is also a d-competitive algorithm against an oblivious adversary, then A is $(c \cdot d)$-competitive against adaptive offline adversaries.*

Hence, randomization does not help against adaptive offline adversaries so that we will ignore them when looking for improved competitive ratios. An immediate consequence of the theorems above is:

Corollary 3.6 *If A is c-competitive against adaptive online adversaries, then there is a c^2-competitive deterministic algorithm for the problem.*

3.3 Paging

In the Paging problem we are given a fast memory (simply called *cache*) of size k, i.e., a cache that can store k pages. The input sequence σ consists of requests to specific pages. The following algorithm is an optimal offline algorithm for this problem.

Algorithm MIN:

For each request to a page that is not in the cache, remove the page whose request is the furthest in the future.

Figure 1: The MIN Algorithm.

Theorem 3.7 *Algorithm MIN is an optimal offline algorithm, i.e., it generates the smallest possible number of page faults for any request sequence σ.*

Proof. Let σ be an arbitrary request sequence and A be an algorithm that generated the smallest number of page faults for σ. As we will show, A can be transformed in such a way that it works like MIN. During that transformation the number of page faults is not increased. In the next lemma we will specify how to do the transformation.

Lemma 3.8 *Suppose that A and MIN work in the same way on the first $i-1$ requests but deviate for the ith request. Then A can be transformed into an algorithm A' so that it holds:*

- *A' and MIN work in the same way on the first i requests.*
- *$A'(\sigma) \leq A(\sigma)$.*

Proof. Let the ith request refer to page x. To serve that request, A may remove the page a from the cache while MIN removes page b. We now specify an algorithm A' that works like A for the first $i-1$ requests and removes page b for the ith request. A' then simulates A until one of the following events happen:

1. If A replaces page b, then A' replaces page a instead. Afterwards, both algorithms are back to an identical configuration and A' continues to work like A. This does not change the number of page faults.

2. If A replaces some page c by a (which certainly has to happen if a is requested but may also happen before), then A' replaces c by b. Afterwards, both algorithms are back to the same configuration and have generated the same number of page faults.

The case that b is requested before case (2) occurs cannot happen since b can only be requested again after the next request to a. Otherwise, MIN would not have removed b from the cache.

If none of the cases above happen, A' can operate like A for the rest of σ, so in any case the lemma holds. □

Successive application of the lemma on the actions of A results in the actions of MIN without increasing the number of page faults, which means that Theorem 3.7 is correct. □

An online algorithm that is often used for paging is the so-called LRU (least recently used) algorithm (see Fig. 2).

Algorithm LRU:
For every request to some page that is not in the cache, remove the least recently used page from the cache to make room for the requested page.

Figure 2: The LRU algorithm.

Unfortuntely, LRU does not have a good performance in the worst case.

Theorem 3.9 *LRU is k-competitive.*

A lower bound can be generated by periodically requesting $k+1$ pages in the same order. MIN would only have a page fault every k requests while LRU would have a page fault for every request, resulting in the k-competitiveness. In general, one can show the following result.

Theorem 3.10 *For all deterministic online algorithms A, A is c-competitive for some $c \geq k$.*

The theorem can easily be derived from the LRU example and will be an exercise. We therefore have to resort to randomized algorithms if we want to obtain a better result. For that we start with the RANDOM algorithm given in Fig. 3.

Algorithm RANDOM:
For every request to some page that is not in the cache, remove a page chosen uniformly at random.

Figure 3: The RANDOM Algorithm.

Theorem 3.11 *RANDOM is not better than k-competitive for an oblivious adversary.*

Proof. Choose the following input sequence for some $N \geq k$:

$$\sigma = \underbrace{b_0 a_1 \dots a_{k-1}}_{P_0} \underbrace{(b_1 a_1 \dots a_{k-1})^N}_{P_1} \underbrace{(b_2 a_1 \dots a_{k-1})^N}_{P_2} \dots$$

As long as in phase P_i the page b_{i-1} is not evicted, there is at least one page fault in the sequence $b_i a_1 \dots a_{k-1}$. The probability that b_{i-1} is evicted in case of a page fault is $1/k$. Let X be the number of page faults in P_i. Certainly,

$\mathbb{P}[X = i] \geq (1 - 1/k)^{i-1}(1/k)$ for all $1 \leq i < N$ since the event $X = i$ contains as a special case the event that it takes i attempts until b_{i-1} is evicted. Thus, $\mathbb{E}[X] \geq \sum_{i=1}^{N-1} i(1 - 1/k)^{i-1}(1/k)$. Since

$$\sum_{i \geq 0}(i+1)\left(1 - \frac{1}{k}\right)^i \cdot \frac{1}{k} = \frac{1}{k} \cdot \left(\sum_{i \geq 0}\left(1 - \frac{1}{k}\right)^i\right)^2 = \frac{1}{k} \cdot \left(\frac{1}{1 - (1 - 1/k)}\right)^2 = k$$

it follows for $N \to \infty$ that $\mathbb{E}[X] \geq k$. On expectation, RANDOM will therefore produce at least k page faults in each phase P_i of σ. An optimal algorithm just replaces b_{i-1} in P_i by b_i so that just one page fault occurs. Thus, on expectation (and also with high probability for a sufficiently long sequence), RANDOM is not better than k-competitive for an oblivious adversary. □

Therefore, RANDOM is not an improvement over LRU. A slight modification of RANDOM, however, leads to much better results (see Fig. 4).

Algorithm MARKING:

The algorithm works in phases. At the beginning of a phase, all pages in the cache are unmarked. If a page is requested, then it is marked. In case of a page fault, a page is chosen uniformly at random among all unmarked pages for eviction. A phase ends right before a page fault if all pages in the cache are marked.

Figure 4: The MARKING Algorithm.

Theorem 3.12 *MARKING is $2H_k$-competitive against oblivious adversaries, where H_k is the k-th harmonic number.*

Proof. Let σ be an arbitrary input sequence. Algorithm MARKING partitions σ into phases $P(1), P(2), \ldots, P(m)$. Each phase $P(i)$ contains requests to exactly k different pages (the marked pages). Furthermore, the first request in $P(i)$ is different from all requests in $P(i-1)$. For each phase $P(i)$ let n_i be the number of new pages in $P(i)$, where a page in $P(i)$ is *new* if it is requested in $P(i)$ but not in $P(i-1)$.

We show that OPT has an amortized cost of at least $n_i/2$ in phase $P(i)$ while MARKING has expected costs of $n_i \cdot H_k$. From that we would get a competitive ratio of

$$c = \frac{\text{MARKING}(\sigma)}{\text{OPT}(\sigma)} \leq \frac{n_i H_k}{n_i/2} = 2H_k$$

Let us consider any two consecutive phases. The phases $P(i)$ and $P(i+1)$ contain exactly $k + n_{i+1}$ pairwise different pages. Therefore, OPT will have at least n_{i+1} page faults in this phase pair. Hence,

$$\text{OPT} \geq n_2 + n_4 + n_6 + \ldots = \sum_{i \text{ even}} n_i$$

If one considers the alternative phase pairs, we get

$$\text{OPT} \geq n_1 + n_3 + n_5 + \ldots = \sum_{i \text{ odd}} n_i$$

So altogether, the amortized costs are

$$\begin{aligned} 2\text{OPT} &\geq \sum_{i \text{ even}} n_i + \sum_{i \text{ odd}} n_i \\ &= \sum_i n_i \end{aligned}$$

In phase $P(i)$, MARKING generates exactly n_i page faults for the requests to new pages. However, there can also be page faults for pages that are not new in $P(i)$. Let us denote the number of old pages that are requested in phase $P(i)$ by o_i,

where page in $P(i)$ is called *old* if it was requested in phase $P(i-1)$. Since a phase ends whenever k different pages have been requested, $o_i + n_i = k$.

Now, let us consider the time point at which $j-1$ of the old pages have already been requested in $P(i)$ and now the jth old page is requested for the first time, where $1 \leq j \leq o_i$. Right before that request, the cache may already contain $n_i(j)$ new pages that have been requested so far, $n_i(j) \leq n_i$. There are $k-(j-1)$ old pages that have not been requested before the request to the jth old page. From these old pages, $n_i(j)$ many are not in the cache any more. Thus, the probability for a page fault is

$$\mathbb{P}[\text{page fault for } j\text{th old page}] = \frac{\binom{k-(j-1)-1}{n_i(j)-1}}{\binom{k-(j-1)}{n_i(j)}} = \frac{n_i(j)}{k-(j-1)} \leq \frac{n_i}{k-(j-1)}$$

Since $n_i \geq 1$, we have $o_i \leq k-1$, and therefore, $1/(k-o_i+1) \leq 1/2$. Thus, the expected cost for the old pages is

$$\begin{aligned} \sum_{j=1}^{o_i} \frac{n_i}{k-(j-1)} &= n_i \left(\frac{1}{k} + \frac{1}{k-1} + \frac{1}{k-2} + \ldots + \frac{1}{k-o_i+1} \right) \\ &\leq n_i \left(\frac{1}{k} + \ldots + \frac{1}{2} \right) = n_i(H_k - 1) \end{aligned}$$

Therefore, the expected cost of phase $P(i)$ is

$$\mathbb{E}[cost(P(i))] \leq n_i + n_i(H_k - 1) = n_i H_k$$

□

Moreover, the following theorem holds, which shows that MARKING is optimal up to a factor of 2.

Theorem 3.13 *Let A be a randomized online algorithm that is c-competitive against oblivious adversaries. Then $c \geq H_k$.*

Proof. Let A be an arbitrary online algorithm. The set of requested pages is $\{p_1, \ldots, p_{k+1}\}$. The oblivious adversary knows A but not the outcomes of its random choices. Nevertheless, it can compute after each request a vector of probabilities

$$Q = (q_1, \ldots, q_{k+1})$$

where q_i is the probability that page p_i is not in the cache of A. It holds that $\sum_{i=1}^{k+1} q_i = 1$.

We divide the input sequence σ into phases, where each phase consists of k subphases. The adversary marks the requested pages like in MARKING and subdivides σ into corresponding phases. With each newly marked page, a new subphase starts. In subphase j, we intend to create an expected cost of $1/(k+1-j)$ for A so that the cost of A in each phase is

$$\sum_{j=1}^{k} \frac{1}{k+1-j} = \frac{1}{k} + \frac{1}{k-1} + \ldots + 1 = H_k$$

The adversary is supposed to create a cost of just 1 in each phase. If so, it holds that $c = H_k/1 = H_k$.

At the beginning of a subphase j there are j marked pages (which consist of the last marked page of the previous phase and those pages that were marked in the past $j-1$ subphases). Let M be the set of marked pages. We set

$$\gamma = \sum_{p_i \in M} q_i$$

and distinguish between two cases.

If $\gamma = 0$ (i.e., all marked pages are in A's cache), then we can select a page p_i from the $k+1-j$ unmarked pages with $q_i \geq 1/(k+1-j)$. This page is requested by the adversary and marked. This ends subphase j.

If $\gamma > 0$, then there is a $p_i \in M$ with $q_i = \epsilon > 0$. The adversary requests p_i and generates an expected cost of ϵ for A. As long as the expected cost for subphase j is less than $1/(k+1-j)$ and $\gamma > \epsilon$, the adversary requests the marked page $p_i \in M$ with the largest q_i. After the end of this loop, we request the unmarked page p_i with largest q_i and mark it, which

ends subphase j. Once the loop is left, the expected cost is either at least $1/(k+1-j)$ or it holds that $\gamma \leq \epsilon$. In the latter case, the expected cost is at least

$$\begin{aligned} \epsilon + \frac{1 - \sum_{p_i \in M} q_i}{k+1-j} &= \epsilon + \frac{1-\gamma}{k+1-j} \geq \epsilon + \frac{1-\epsilon}{k+1-j} \\ &\geq \frac{\epsilon}{k+1-j} + \frac{1-\epsilon}{k+1-j} = \frac{1}{k+1-j} \end{aligned}$$

so in any case we have an expected cost of at least $1/(k+1-j)$. At the end of the phase, the last marked page stays marked for the new phase (and marks the beginning of that phase).

At the beginning of a phase, the adversary replaces that page from the cache that is the last that is requested in that phase. In this way, it achieves a cost of 1 per phase. □

3.4 Self-organizing lists

In the self-organizing list problem we initial have an arbitrarily ordered list of data, and σ represents an arbitrary request sequence to these data. When a list element is requested, a cost occurs that depends on the position of the element in the list. The requested element may then be moved to an arbitrary position towards the front of the list at no additional cost. Moreover, it is possible to switch two neighboring elements at a cost of 1.

The following online strategies are possible:

- **Move-To-Front (MTF):** The requested element is moved to the front of the list.
- **Transpose:** The requested element is switched with its predecessor in the list.
- **Frequency-Count (FC):** Use a counter for each element in the list that is initialized with 0. Whenever the element is requested, its counter is increased by 1. After each request, the list is sorted in descending order according to the values of the counters.

The following properties can be shown for these rules:

Theorem 3.14 *MTF is 2-competitive.*

Theorem 3.15 *Transpose and FC are not c-competitive for any constant c.*

Furthermore, the following statement holds, which demonstrates that MTF is optimal within the class of deterministic algorithms.

Theorem 3.16 *Let A be any deterministic online algorithm for the self-organizing list. If A is c-competitive, then $c \geq 2$.*

Proof. Consider a list with n elements. The adversary generates a request sequence σ in which continuously the last element in A's list is requested. For $|\sigma| = m$, $A(\sigma) = mn$.

Now, consider the offline strategy SL that uses a static list in which the elements are ordered according to the number of times they are requested in σ. Let m be a multiple of n. The worst case for SL is the case that every element is requested equally often. Then for each $i = 1 \ldots m$, SL incurs m/n times a cost of i. Hence,

$$\mathrm{OPT}(\sigma) \leq \sum_{i=1}^{n} i \cdot \frac{m}{n} = \frac{n+1}{2} \cdot m$$

Therefore,

$$c = \frac{A(\sigma)}{\mathrm{OPT}(\sigma)} \geq \frac{mn}{m(n+1)/2} = \frac{2n}{n+1} = 2 - \frac{2}{n+1}$$

For large n the theorem follows. □

Hence, to get below 2, randomized algorithms are needed. Such an algorithm is given in Fig. 5.

Theorem 3.17 *BIT is 1.75-competitive against oblivious adversaries.*

Algorithm BIT:

Store for each list element x a bit $b(x)$. Initially, these bits are set to 0 or 1 independently and uniformly at random. For each request to x, $b(x)$ is complemented. If $b(x)$ switches to 1, then x is moved to the front of the list, and otherwise the list position of x stays as it is.

Figure 5: The BIT Algorithm.

Proof. Consider an arbitrary request sequence σ. At each time of σ and for each x, $b(x)$ has the same probability to be 0 or 1 because

$$b(x) = (\text{initial value} + \text{number of requests to } x) \bmod 2$$

We use an amortized analysis for the number of inversions. For an *inversion* (x, y), x is in front of y in OPT's list but y is in front of x in BIT's list. We distinguish between inversions at which $b(x) = 0$ (type 1) and $b(x) = 1$ (type 2). As a potential function, we use

$$\Phi = (\text{number of type 1 inversions}) + 2 \cdot (\text{number of type 2 inversions})$$

We denote the online cost of BIT with $\mathrm{BIT}(\sigma(t))$ and define the expected amortized cost as

$$\mathbb{E}[\mathrm{BIT}(\sigma(t))] + \mathbb{E}[\Phi(t) - \Phi(t-1)] = \mathbb{E}[\mathrm{BIT}(\sigma(t)] + \mathbb{E}[\Delta\Phi(t)]$$

Our goal is to show that for each time t,

$$\mathbb{E}[\mathrm{BIT}(\sigma(t))] + \mathbb{E}[\Phi(t) - \Phi(t-1)] \leq 1.75\mathrm{OPT}(\sigma(t))$$

Then BIT is 1.75-competitive, because when summing over all t we get:

$$\begin{aligned} \sum_{t=1}^{m} (\mathbb{E}[\mathrm{BIT}(\sigma(t))] + \mathbb{E}[\Phi(t) - \Phi(t-1)]) &\leq c \sum_{t=1}^{m} \mathrm{OPT}(\sigma(t)) \\ \Leftrightarrow \quad \mathbb{E}[\mathrm{BIT}(\sigma)] + \mathbb{E}[\Phi(m)] - \mathbb{E}[\Phi(0)] &\leq c\mathrm{OPT}(\sigma) \\ \Leftrightarrow \quad \mathbb{E}[\mathrm{BIT}(\sigma)] &\leq c\mathrm{OPT}(\sigma) - \mathbb{E}[\Phi(m)] \end{aligned}$$

since $\Phi(0) = 0$. Since in addition to that, $\Phi(t) \geq 0$ for all t, $\mathbb{E}[\mathrm{BIT}(\sigma)] \leq c\mathrm{OPT}(\sigma)$.

In our analysis, OPT serves the requests before BIT. We consider now the cases in which an inversion is removed or added. Let k be the position in OPT's list directly before the currently requested list element x and i be the position directly before the new position of x (see Fig. 6).

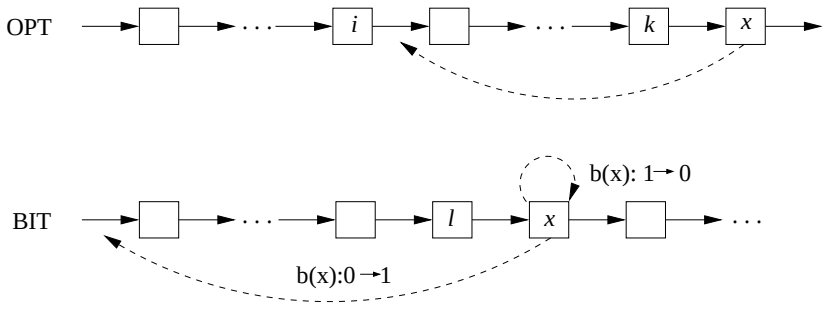

Figure 6: Processing of x in OPT and BIT.

Due to OPT's movement of x to the front, at most $k - i$ new inversions can be created (namely, for those pairs (x, y) with y at the positions $i + 1, \ldots, k$). Thus,

$$\begin{aligned} \mathrm{OPT}(\sigma(t)) &= k + 1 \quad \text{and} \\ \mathbb{E}[\Delta_{\mathrm{OPT}}\Phi] &\leq (k - i)\,(1 \cdot \mathbb{P}[\text{type 1}] + 2 \cdot \mathbb{P}[\text{type 2}]) = (k - i)\left(\frac{1}{2}(1 + 2)\right) = 1.5(k - i) \end{aligned}$$

Let x be at position $\ell + 1$ in BIT's list (see Fig. 6). According to our cost model,

$$\mathrm{BIT}(\sigma(t)) = \ell + 1$$

For the potential change due to BIT's processing of the request we distinguish between two cases.
First case: $\ell \leq i$.

(a) If $b(x) = 1$ before the access, then $b(x)$ changes to 0, which causes the inversions (x, y) to become cheaper (see the definition of Φ). The position of x will not be changed, so there cannot be any new inversions. Hence, $\Delta_{\mathrm{BIT}_a}\Phi \leq 0$.

(b) If $b(x) = 0$, then at most ℓ new inversions (y, x) can occur, that are valued according to the bit values of the list elements y jumped over by x. Therefore, we get

$$\mathbb{E}[\Delta_{\mathrm{BIT}_b}\Phi] \leq \ell \left(\frac{1}{2}(1+2)\right) = 1.5\ell$$

Since (a) and (b) occur with probability $1/2$, it holds:

$$\begin{aligned}
\mathrm{BIT}(\sigma(t)) + \mathbb{E}[\Delta\Phi] &= \mathrm{BIT}(\sigma(t)) + \mathbb{E}[\Delta_{\mathrm{OPT}}\Phi] + \mathbb{E}[0.5\Delta_{\mathrm{BIT}_a}\Phi + 0.5\Delta_{\mathrm{BIT}_b}\Phi] \\
&\leq \ell + 1 + 1.5(k-i) + 0.5 \cdot 0 + 0.5 \cdot 1.5\ell \\
&= \ell + 1 + 0.75\ell + 1.5(k-i) \\
&= 1.75\ell + 1.5(k-i) + 1 \\
&\leq 1.75i + 1.75(k-i) + 1 \\
&\leq 1.75(k+1) \\
&= 1.75\mathrm{OPT}(\sigma(t))
\end{aligned}$$

Second case: $i \leq \ell$.

(a) If $b(x) = 1$, then the position of x does not change. Since x is now at position $i+1$ in OPT, there are $\ell - i$ inversions (x, y) whose type changes from 2 to 1. From that we obtain that $\Delta_{\mathrm{BIT}_a}\Phi \leq -(\ell - i)$.

(b) If $b(x) = 0$, then x is moved to the front of the list, which removes $\ell - i$ inversions (x, y). So Φ reduces by at least $\ell - i$. Moreover, at most i new inversions (y, x) are created, that add an expected value of 1.5 to Φ. Therefore, $\mathbb{E}[\Delta_{\mathrm{BIT}_b}\Phi] \leq 1.5i - (\ell - i)$.

Since (a) and (b) occur with the same probability, we get:

$$\begin{aligned}
\mathrm{BIT}(\sigma(t)) + \mathbb{E}[\Delta\Phi] &= \mathrm{BIT}(\sigma(t)) + \mathbb{E}[\Delta_{\mathrm{OPT}}\Phi] + \mathbb{E}[0.5\Delta_{\mathrm{BIT}_a}\Phi + 0.5\Delta_{\mathrm{BIT}_b}\Phi] \\
&\leq \ell + 1 + 1.5(k-i) - 0.5(\ell - i) + 0.5(1.5i - \ell + i) \\
&= \ell + 1 + 1.5k - 1.5i - 0.5\ell + 0.5i + 0.75i - 0.5\ell + 0.5i \\
&= 1 + 1.5k + 0.25i \\
&\leq 1 + 1.75k \\
&\leq 1.75(k+1) \\
&= 1.75\mathrm{OPT}(\sigma(t))
\end{aligned}$$

□

One may wonder whether further improvements are possible. Indeed, BIT turned out not to be the best possible online algorithm, but Teia [3] has shown that any randomized online algorithm for the self-organizing list problem has a competitive ratio of at least 1.5.

References

[1] S. Albers. Online- und Approximationsalgorithmen. Institut für Informatik, Universität Freiburg. Siehe auch http://www.informatik.uni-freiburg.de/∼ipr/ipr_teaching/ss_06/alg06.html, 1994.

[2] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11:73–91, 1994.

[3] B. Teia. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47:5–9, 1993.