Run Generation Revisited:
What Goes Up May or May Not Come Down

Michael A. Bender† Samuel McCauley† Andrew McGregor‡ Shikha Singh†
Hoa T. Vu§

Abstract

In this paper, we revisit the classic problem of run generation. Run generation is the first phase of external-memory sorting, where the objective is to scan through the data, reorder elements using a small buffer of size $M$, and output runs (contiguously sorted chunks of elements) that are as long as possible.

We develop algorithms for minimizing the total number of runs (or equivalently, maximizing the average run length) when the runs are allowed to be sorted or reverse sorted. We study the problem in the online setting, both with and without resource augmentation, and in the offline setting.

• We analyze alternating-up-down replacement selection (runs alternate between sorted and reverse sorted), which was studied by Knuth as far back as 1963. We show that this simple policy is asymptotically optimal. Specifically, we show that alternating-up-down replacement selection is 2-competitive and no deterministic online algorithm can perform better.

• We give online algorithms having smaller competitive ratios with resource augmentation. Specifically, we exhibit a deterministic algorithm that, when given a buffer of size $4M$, is able to match or beat any optimal algorithm having a buffer of size $M$. Furthermore, we present a randomized online algorithm which is $7/4$-competitive when given a buffer twice that of the optimal.

• We demonstrate that performance can also be improved with a small amount of foresight. We give an algorithm, which is $3/2$-competitive, with foreknowledge of the next $3M$ elements of the input stream. For the extreme case where all future elements are known, we design a PTAS for computing the optimal strategy a run generation algorithm must follow.

• We present algorithms tailored for “nearly sorted” inputs which are guaranteed to have optimal solutions with sufficiently long runs.

---

†Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, USA. Email: {bender,smccauley, shiksingh}@cs.stonybrook.edu.

‡Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA. Email: {mcgregor,hvu}@cs.umass.edu.
1 Introduction

External-memory sorting algorithms are tailored for data sets too large to fit in main memory. Generally, these algorithms begin their sort by bringing chunks of data into main memory, sorting within memory, and writing back out to disk in sorted sequences, called runs [15, 19, 26, 34].

We revisit the classic problem of how to maximize the length of these runs, the run-generation problem. The run-generation problem has been studied in its various guises for over 50 years [14, 17–19, 25, 30, 31, 34]. The most well-known external-memory sorting algorithm is multi-way merge sort [1, 8, 15, 22, 28, 29, 40, 42, 44]. The multi-way merge sort is formalized in the disk-access machine (DAM) model of Aggarwal and Vitter [1]. If \( M \) is the size of RAM and data is transferred between main memory and disk in blocks of size \( B \), then an \( M/B \)-way merge sort has a complexity of \( O((N/B) \log_{M/B} (N/B)) \) I/Os, where \( N \) is the number of elements to be sorted. This is the best possible [1].

A top-down description of multi-way merge sort follows. Divide the input into \( M/B \) subproblems, recursively sort each subproblem, and merge them together in one final scan through the input. The base case is reached when each subproblem has size \( O(M) \), and therefore fits into RAM.

A bottom-up description of the algorithm starts with the base case, which is the run-generation phase. Naively, we can always generate runs of length \( M \): ingest \( M \) elements into memory, sort them, write them to disk, and then repeat.

The point of run generation is to produce runs longer than \( M \). After all, with typical values of \( N \) and \( M \), we rarely need more than one or two passes over the data after the initial run-generation phase. Longer runs can mean fewer passes over the data or less memory consumption during the merge phase of the sort. Because there are few scans to begin with, even if we only do one fewer scan, the cost of a merge sort is decreased by a significant percentage. Run generation has further advantages in databases even when a full sort is not required [22, 24].

**Replacement Selection.** The classic algorithm for run generation is called replacement selection [20, 26, 28]. We describe replacement selection below by assuming that the elements can be read into memory and written to disk one at a time.

To create an increasing run starting from an initially full internal memory, proceed as follows:
1. From main memory, select the smallest element \( 2 \) at least as large as every element in the current run.
2. If no such element exists, then the run ends; select the smallest element in the buffer.
3. Eject that element, and ingest the next element, so that the memory stays full.

Replacement selection can deal with input elements one at a time, even though the DAM model transfers input between RAM and disk \( B \) elements at a time. To see why, consider two additional blocks in memory, an “input block,” which stores elements recently read from disk, and an “output block,” which stores elements that have already been placed in a run and will be written back to disk. To ingest, take an element from the input block, and to eject an element, put the element in the output block. When the input block becomes empty, fill it from disk and when the output block fills up, flush it to disk. Similar to previous work, in this paper, we ignore these two blocks.

**Properties of Replacement Selection.** It has been known for decades that when the input appears in random order, then the expected length of a run is actually \( 2M \), not \( M \) [18, 19, 25]. In [26], Knuth gives memorable intuition about this result, conceptualizing the buffer as a snowplow traveling along a circular track.

Replacement selection performs particularly well on nearly sorted data (for many intuitive notions of “nearly”), and the runs generated are much larger than \( M \). For example, when each element in the input appears at a distance at most \( M \) from its actual rank, replacement selection produces a single run.

---

1 The external-memory model, also called the I/O model, applies to any two levels of the memory hierarchy.
2 Observe that data structures such as in-memory heaps can be used to identify the smallest elements in memory. However, from the perspective of minimizing I/Os, this does not matter—computation is free in the DAM model.
On the other hand, replacement selection performs poorly on reverse-sorted data. It produces runs of length $M$, which is the worst possible.

**Up-Down Replacement Selection.** From the perspective of the sorting algorithm, it matters little, or not at all, whether the initially generated runs are sorted or reverse sorted.

This observation has motivated researchers to think about run generation when the replacement-selection algorithm has a choice about whether to generate an up run or a down run, each time a new run begins.

Knuth [25] analyzes the performance of replacement selection that alternates deterministically between generating up runs and down runs. He shows that for randomly generated data, this alternative policy performs worse, generating runs of expected length $3M/2$, instead of $2M$.

Martinez-Palau et al. [34] revive this idea in an experimental study. Their two-way-replacement-selection algorithms heuristically choose between whether the run generation should go up or down. Their experiments find that two-way replacement selection (1) is slightly worse than replacement selection for random input (in accordance with Knuth [25]) and (2) produces significantly longer runs on inputs that have mixed up-down runs and reverse-sorted inputs.

**Our Contributions.** The results in our present paper complement these earlier results. In contrast to Knuth’s negative result for random inputs [25], we show that strict up-down alternation is best possible for worst-case inputs. Moreover, we give better competitive ratios with resource augmentation and lookahead, which helps explain why heuristically choosing between up and down runs based on what is currently in memory may lead to better solutions. Resource augmentation is a standard tool used in competitive analysis [9,11–13,38,39] to empower an online algorithm when comparing against an omniscient and all-powerful optimal algorithm.

Up-down run generation boils down to figuring out, each time a run ends, whether the next run should be an up run or a down run. The objective is to minimize the number of runs output. We establish the following:

1. **Analysis of alternating-up-down replacement selection.** We revisit (online) alternating-up-down replacement selection, which was earlier analyzed by Knuth [25]. We prove that alternating-up-down replacement selection is $2$-competitive and asymptotically optimal for deterministic algorithms. To put this result in context, it is known that up-only replacement selection is a constant factor better than up-down replacement selection for random inputs, but can be an unbounded factor worse than optimal for arbitrary inputs.

2. **Resource augmentation with extra buffer.** We analyze the effect of augmenting the buffer available to an online algorithm on its performance. We show that with a constant factor larger buffer, it is possible to perform better than twice optimal. Specifically, we exhibit a deterministic algorithm that, when given a buffer of size $4M$, matches or beats any optimal algorithm having a buffer of size $M$. We also design a randomized online algorithm which is $7/4$-competitive using a $2M$-size buffer.

3. **Resource augmentation with extra visibility.** We show that performance factors can also be improved, without augmenting the buffer, if an algorithm has limited foreknowledge of the input. In particular, we propose a deterministic algorithm which attains a competitive ratio of $3/2$, using its regular buffer of size $M$, with a lookahead of $3M$ incoming elements of the input (at each step).

4. **Better bounds for nearly sorted data.** We give algorithms that perform well on inputs that have some inherent sortedness. We show that the greedy offline algorithm is optimal for inputs on which the optimal runs are at least $5M$ elements long. We also give a $3/2$-competitive algorithm with $2M$-size buffer when the optimal runs are at least $3M$ long. These results are reminiscent of previous literature studying sorting on inputs with “bounded disorder” [10] and adaptive sorting algorithms [16,33,41].

5. **PTAS for the offline problem.** We give a polynomial-time approximation scheme for the offline run-generation problem. Specifically, our offline polynomial-time approximation algorithm guarantees a $(1 + \varepsilon)$-approximation to the optimal solution. We first give an algorithm with the running time of $O(2^{1/\varepsilon} N \log N)$ and then improve the running time to $O\left(\frac{1}{\varepsilon}N \log N\right)$.

\[ \text{Note that for a given input, minimizing the number of runs output is equivalent to maximizing the average length of runs output.} \]
Paper Outline. The paper is organized as follows. In Section 2, we formalize the up-down run generation problem and provide necessary notation. Section 3 contains important structural properties of run generation and key lemmas used in analyzing our algorithms. Analysis of alternating-up-down replacement selection and online lower bounds are in Section 4. Algorithms with resource augmentation, along with properties of the greedy algorithm, are presented in Section 5. The offline version of the problem is studied in Section 6. Improvements on well-sorted input are presented in Section 7. Section 8 summarizes related work and we conclude with open problems in Section 9. Due to space constraints, we defer some proofs to the appendix (Appendix A).

2 Up-Down Run Generation

In this section, we formalize the up-down run generation problem and introduce notation.

2.1 Problem Definition

An instance of the up-down run generation problem is a stream $I$ of $N$ elements. The elements of $I$ are presented to the algorithm one by one, in order. They can be stored in the memory of size $M$ available to the algorithm, which we henceforth refer to as the buffer. Each element occupies one slot of the buffer. In general, the model allows duplicate elements, although some results, particularly in Section 5 and Section 7, do require uniqueness.

We say that an algorithm $A$ reads an element of $I$ when $A$ transfers the element from the input sequence to the buffer. We say that an algorithm $A$ writes an element when $A$ ejects the element from its buffer and appends it to the output sequence $S$.

Every time an element is written, its slot in the buffer becomes free. Unless stated otherwise, the next element from the input takes up the freed slot. Thus the buffer is always full, except when the end of the input is reached and there are fewer than $M$ unwritten elements.

An algorithm can decide which element to eject from its buffer based on (a) the current contents of the buffer and (b) the last element written. The algorithm may also use $o(M)$ additional words to maintain its internal state (for example, it can store the direction of the current run). However, the algorithm cannot arbitrarily access $S$ or $I$—it can only append elements to $S$, and access the next in-order element of $I$. We say the algorithm is at time step $t$ if it has written exactly $t$ elements.

A run is a sequence of sorted or reverse-sorted elements. The cost of the algorithm is the smallest number of runs we can use to partition its output. Specifically, the number of runs in an output $S$, denoted $R(S)$, is the smallest number of mutually disjoint sequences $S_1, S_2, \ldots, S_{R(S)}$ such that each $S_i$ is a run and $S = S_1 \circ \cdots \circ S_{R(S)}$ where $\circ$ indicates concatenation.

We let $\text{OPT}(I)$ be the minimum number of runs of any possible output sequence on input $I$, i.e., the number of runs generated by the optimal offline algorithm. If $I$ is clear from context, we denote this as OPT. Our goal is to give algorithms that perform well compared to OPT for every $I$. We say that an online algorithm is $\beta$-competitive if on any input, its output $S$ satisfies $R(S) \leq \beta\text{OPT}$.

At any time step, an algorithm’s unwritten-element sequence is comprised of the contents of the buffer, concatenated with the remaining (not yet ingested) input elements. For the purpose of this definition, we assume that the elements in the buffer are stored in their arrival order (their order in the input sequence $I$).

Time step $t$ is a decision point or decision time step for an algorithm $A$ if $t = 0$ or if $A$ finished writing a run at $t$. At a decision point, $A$ needs to decide whether the next run will be increasing or decreasing.

Reading in the next element of the input when there is a free slot in the buffer never hurts the performance of any algorithm. However, we allow the algorithm in the proof of Lemma 16 to maintain free slots in the buffer to simplify the analysis.
2.2 Notation

We employ the following notation. We use \((x \nearrow y)\) to denote the increasing sequence \(x, x+1, x+2, \ldots, y\) and \((x \searrow y)\) to denote the decreasing sequence \(x, x-1, x-2, \ldots, y\). We use \(\circ\) to denote concatenation: if \(A = a_1, a_2, \ldots, a_k\) and \(B = b_1, b_2, \ldots, b_\ell\) then \(A \circ B = a_1, a_2, \ldots, a_k, b_1, b_2, \ldots, b_\ell\).

Let \(A = a_1, a_2, \ldots, a_k\). We use \(A \oslash x\) to denote the sequence \(a_1 + x, a_2 + x, \ldots, a_k + x\). Similarly, we use \(A \odot x\) to denote the sequence \(a_1 x, a_2 x, \ldots, a_k x\).

Let \(A, B\) be sequences. We say \(A\) covers \(B\) if for all \(e \in B, e \in A\). A subsequence of a sequence \(A = a_1, \ldots, a_k\) is a sequence \(B = a_{n_1}, a_{n_2}, \ldots, a_{n_\ell}\) where \(1 \leq n_1 < n_2 < \ldots < n_\ell \leq k\).

3 Structural Properties

In this section, we identify structural properties of the problem and the tools used in the analysis of our algorithms, which will be important in the rest of the paper.

3.1 Maximal Runs

We show that in run generation, it is never a good idea to end a run early, and never a good idea to “skip over” an element (keeping it in buffer instead of writing it out as part of the current run).

To begin, we show that adding elements to an input sequence never decreases the number of runs. Note that if \(S'\) is a subsequence of \(S\), then \(R(S') \leq R(S)\) by definition.

**Lemma 1.** Consider two input streams \(I\) and \(I'\). If \(I'\) is a subsequence of \(I\), then \(\text{OPT}(I') \leq \text{OPT}(I)\).

**Proof.** Let \(A\) be an algorithm with input stream \(I\) and output \(S\). Suppose that \(A\) produces the optimal number of runs on \(I\), that is \(R(S) = \text{OPT}(I)\). Consider an algorithm \(A'\) on \(I'\). Algorithm \(A'\) performs the same operations as \(A\), but when it reaches an element that is not in \(I'\) (but is in \(I\)), it executes a no-op. These no-ops mean that the buffer of \(A'\) may not be completely full, since elements that \(A\) has in buffer do not exist in the buffer of \(A'\). Let \(S'\) be the output of \(A'\); \(S'\) is a subsequence of \(S\).

Then \(\text{OPT}(I') \leq R(S') \leq R(S) = \text{OPT}(I)\). \(\square\)

A maximal increasing run is a run generated using the following rules (a maximal decreasing run is defined similarly):
1. Start with the smallest element in the buffer and always write the smallest element that is larger than the last element written.
2. End the run only when no element in the buffer can continue the run, i.e., all elements in buffer are smaller than the last element written.

**Lemma 2.** At any decision time step, a maximal increasing (decreasing) run \(r\) covers every other (non-maximal) increasing (decreasing) run \(r'\).

A proper algorithm is an algorithm that always writes maximal runs. We say an output is proper if it is generated by a proper algorithm. We show that there always exists an optimal proper algorithm.

**Theorem 3.** For any input \(I\), there exists a proper algorithm \(A\) with output \(S\) such that \(R(S) = \text{OPT}(I)\).

**Proof.** We prove this by induction on the number of runs. If there is only one run, it must be maximal. Assume that all inputs \(I_t\) with \(\text{OPT}(I_t) = t\) have a maximal proper algorithm. Consider an input \(I_{t+1}\) with \(\text{OPT}(I_{t+1}) = t + 1\). Assume that an optimal algorithm on \(I_{t+1}\) is \(A_O\), and it is not proper; we will construct a proper \(A\) with the same number of runs. The first run \(A\) writes is maximal and has the same direction of
Theorem 7. Let \( S \) be an algorithm with output \( S \). Partition \( S \) into \( k \) contiguous subsequences \( S_1, S_2 \ldots S_k \). Let \( x_i \) be the number of runs in \( S_i \). For \( 1 \leq i \leq k \), let \( I_i \) be the unwritten-element sequence after \( A \) outputs \( S_{i-1} \); let \( I_1 = I \) and \( I_{k+1} = \emptyset \). Let \( \alpha, \beta \geq 1 \). For each \( I_i \), let \( S_i' \) be the output of an optimal algorithm on \( I_i \). If for all \( i \leq k \), \( S_i \) covers the first \( y_i \) runs of \( S_i' \), and \( x_i/y_i \leq \beta \), then \( R(S) \leq \beta \text{OPT} \). Similarly, if for all \( i \leq k \), \( S_i \) covers the first \( y_i \) runs of \( S_i' \), and \( E[x_i]/y_i \leq \alpha \), then \( E[R(S)] \leq \alpha \text{OPT} \).

Proof. Consider \( I'_i \), the unwritten-element sequence at the end of the first \( y_i \) runs of \( S'_{i-1} \) (we let \( I'_1 = I \)). We show that \( \text{OPT}(I_i) \leq \text{OPT} - \sum_{j=1}^{i-1} y_j \) for all \( 1 \leq i \leq k \) using induction. Note that \( \text{OPT}(I_1) = \text{OPT} \) (the base case). Induction hypothesis: assume \( \text{OPT}(I_i) \leq \text{OPT} - \sum_{j=1}^{i-1} y_j \). Since \( S_{i+1} \) covers the first \( y_i \) runs of \( S'_{i+1} \), by \text{Lemma 5} \( I_{i+1} \) is a subsequence of \( I'_{i+1} \). Then by \text{Lemma 5} \( \text{OPT}(I_{i+1}) \leq \text{OPT}(I'_{i+1}) \). By definition, for \( i > 1 \),

\[
\text{OPT}(I'_{i+1}) = \text{OPT}(I_i) - y_i \leq \text{OPT} - \sum_{j=1}^{i} y_j.
\]
Therefore, $\text{OPT}(I_{i+1}) \leq \text{OPT} - \sum_{j=1}^{i} y_j$. When $i = k$, we have $\text{OPT}(I_{k+1}) \leq \text{OPT} - \sum_{j=1}^{k} y_j$. But since $I_{k+1}$ contains no elements, $\text{OPT}(I_{k+1}) = 0$, and we have $\sum_{j=1}^{k} y_j \leq \text{OPT}$. Since $R(S) = \sum_{j=1}^{k} x_j$, and $\sum_{i=1}^{k} x_i \leq \beta \sum_{i=1}^{k} y_i$, we have the following:

$$R(S) = \frac{\sum_{i=1}^{k} x_i}{\text{OPT}} \cdot \text{OPT} \leq \frac{\sum_{i=1}^{k} x_i}{\sum_{i=1}^{k} y_i} \cdot \text{OPT} \leq \beta \text{OPT}.$$  

We also have the same in expectation, that is,

$$\mathbb{E}[R(S)] = \mathbb{E}\left[\sum_{i=1}^{n} x_i\right] \leq \alpha \sum_{i=1}^{n} y_i \leq \alpha \cdot \text{OPT}.$$  

\[\Box\]

## 4 Up-Down Replacement Selection

We begin by analyzing the alternating up-down replacement selection, which deterministically alternates between writing (maximal) up and down runs. Knuth [25] showed that when the input elements arrive in a random order (all permutations of the input are equally likely), alternating-up-down replacement selection performs worse than standard replacement selection (all up runs). Specifically, he showed that the expected length of runs generated by up-down-replacement selection is $1.5M$ on random input, compared to the expected length of $2M$ of replacement selection.

In this section, we show that for deterministic online algorithms, alternating-up-down replacement selection is, in fact, asymptotically optimal for any input. It generates at most twice the optimal number of runs in the worst case. This is the best possible—no deterministic algorithm can have a better competitive ratio.

### 4.1 Alternating-Up-Down Replacement Selection is 2-competitive

We begin by giving a structural lemma, analyzing identical runs on two inputs in which one input is a subsequence of the other.

**Lemma 8.** Consider two inputs $I_1$ and $I_2$, where $I_2$ is a subsequence of $I_1$. Let $S_1$ and $S_2$ be proper outputs of $I_1$ and $I_2$ such that:
1. $S_1$ and $S_2$ have initial runs $r_1$ and $r_2$ respectively,
2. $r_1$ and $r_2$ have the same direction

Let the unwritten-element sequence after $r_1$ and $r_2$ be $I'_1$ and $I'_2$ respectively. Then $I'_2$ is a subsequence of $I'_1$.

**Proof.** Assume that $r_1$ and $r_2$ are up runs (a similar analysis works for down runs). Let $r'_2$ be a run that is a subsequence of $r_1$, consisting of all elements of $r_1$ that are also in $I_2$. Then $r'_2$ can be produced by an algorithm $A'$ that mirrors the algorithm $A$ that generates $r_1$. When $A$ reads or writes an element in $I_2$, $A'$ reads or writes that element; when $A$ reads or writes an element not in $I_2$, $A'$ does nothing. Since $r_2$ is maximal, it covers $r'_2$ by Lemma 2.  

**Theorem 9.** Alternating up-down replacement selection is 2-competitive.

**Proof.** We show that we can apply Theorem 7 to this algorithm with $\beta = 2$.

In any partition that is not the last one of the output, the alternating algorithm writes a maximal up run $r_u$ and then writes a maximal down run $r_d$. We must show that $r_u \circ r_d$ covers any run $r_O$ written by a proper optimal algorithm on $I_r$, the unwritten element sequence at the beginning of the partition.
If \( r_O \) is an up run, then \( r_O = r_u \) and thus is covered by \( r_u \circ r_d \). If \( r_O \) is a down run, consider \( I' \), the unwritten-element sequence after \( r_u \) is written; \( I' \) is a subsequence of \( I_r \). By Lemma 8 (with \( I_1 = I_r \) and \( I_2 = I' \)), \( r_u \circ r_d \) covers \( r_O \).

In the last partition, the algorithm can write at most two runs while any optimal output must contain at least one run. Hence \( x_i/y_i \leq 2 \) in all partitions as required.

4.2 Lower Bounds on Online Algorithms for Up-Down Run Generation

Now, we show that no deterministic online algorithm can hope to perform better than alternating-up-down replacement selection. Then, we partially answer the question of whether randomization helps overcome this impossibility result. Specifically, we show that no randomized algorithm can achieve a competitive ratio better than \( 3/2 \). We provide the main ideas of the proofs here and defer the details to Appendix A.

**Theorem 10.** Let \( A \) be any online deterministic algorithm with output \( S_I \) on input \( I \). Then there are arbitrarily long \( I \) such that \( R(S_I) \geq 2 \text{OPT}(I) \).

**Proof Sketch.** Given any \( M \) elements in the buffer, every time \( A \) commits to a run direction (up/down), the adversary sets the incoming elements such that they do not help the current run. Thus, \( A \) is forced to have runs of length at most \( M \) while \( \text{OPT} \) (since it has knowledge of the future) can do better.

We also give a lower bound for randomized algorithms using similar ideas; however, in this case we do not have a matching upper bound. We use Yao’s minimax principle to prove this bound. That is, we generate a randomized input and show that any deterministic algorithm cannot perform better than \( 3/2 \) times \( \text{OPT} \) on that input against an oblivious adversary.

**Theorem 11.** Let \( A \) be any online, randomized algorithm. Then there are arbitrarily long input sequences such that \( \mathbb{E}[R(S_I)] \geq (3/2) \text{OPT}(I) \).

5 Run Generation with Resource Augmentation

In this section, we use resource augmentation to circumvent the impossibility result on the performance of deterministic online algorithms. We consider two kinds of augmentation:

- **Extra Buffer:** The algorithm’s buffer is actually a constant factor larger, that is, it can use its large buffer to read elements from the input, rearrange them, and write to the output.
- **Extra Visibility:** The algorithm’s buffer is restricted to be of size \( M \) but it has prescience—the algorithm can see some elements in the immediate future (say, the next \( 3M \) elements), without the ability to write them early.

We present algorithms that, under the above conditions, achieve a competitive ratio better than 2 when compared against an optimal offline algorithm with a buffer of size \( M \).

Resource augmentation is a common tool used in competitive analysis [9, 11–13, 38, 39]. It gives the online algorithm power to make better decisions and exclude worst case inputs, allowing us to compare the performance, more realistically, against an all-powerful offline optimal algorithm.

The results in this section require the elements of the input to be unique. Duplicate elements can nullify the extra ability to see or write future (non-repeated) elements which is provided by visibility and buffer-augmentation respectively. For example, consider the input,

\[
I = (99, 101, 100, \ldots, 100, \ldots). \underbrace{\ldots}_{cM-2 \text{ times}}
\]

On input \( I \), any algorithm with \( cM \)-size buffer or visibility is as powerless as the one without any augmentation.
Note that the assumption of distinct elements in run generation is not new. Knuth’s analysis of the average run lengths [25] also requires uniqueness.

We begin by analyzing the greedy algorithm for run generation. Greedy is a proper algorithm which looks into the future at each decision point, determines the length of the next up and down run and writes the longer run.

Greedy is not an online algorithm. However, it is central to our resource augmentation results. The idea of resource augmentation, in part, is that the algorithm can use the extra buffer or visibility to determine, at each decision point, which direction (up or down) leads to the longer run.

Greedy looks into the future at each decision point, determines the length of the next up and down run and writes the longer run.

Next, we show that all the runs written by the greedy algorithm (except the last two) are guaranteed to be too long in the following lemma.

**Example 12.** Consider the input $I = I_1 \circ (I_1 \oplus 10M) \circ (I_1 \oplus 20M) \circ \cdots \circ (I_1 \oplus 10cM)$, where

$I_1 = (4M + 4 \nearrow 5M + 3) \circ (M + 2) \circ (5M + 4 \nearrow 6M + 3)$

$\circ (2M + 1 \nearrow 3M - 1) \circ (4M + 3 \searrow 3M + 4) \circ (2M \searrow M + 3) \circ (M + 1 \searrow 1)$.

On input $I$ above, writing down runs repeatedly produces $2c$ runs; two for each $I \oplus i10M$. On the other hand, the output of greedy is $S_1 \circ (S_1 \oplus 10M) \circ \cdots \circ (S_1 \oplus c10M)$, where $S_1 = (4M + 4 \nearrow 6M + 3) \circ (M + 2) \circ (2M + 1 \nearrow 3M - 1) \circ (3M + 4 \nearrow 4M + 3) \circ (2M \searrow M + 3) \circ (M + 1 \searrow 1)$ which contains $3c$ runs.

Next, we show that all the runs written by the greedy algorithm (except the last two) are guaranteed to have length at least $5M/4$. In contrast, up-down replacement selection can have have runs of length $M$ in the worst case.

**Theorem 13.** Each greedy run, except the last two runs, has length at least $M + \lceil \lfloor M/2 \rfloor/2 \rceil$.

We now bound how far into the future an algorithm must see to be able to determine which direction greedy would pick at a particular decision point. Intuitively, an algorithm should never have to choose between a very long up run and a very long down run. We formalize this idea about the non-greedy run not being too long in the following lemma.

**Lemma 14.** Given an input $I$ with no duplicate elements. Let the two possible initial increasing and decreasing runs be $r_1$ and $r_2$. Then $|r_1| < 3M$ or $|r_2| < 3M$.

The next example shows that the above bound is tight.

**Example 15.** Consider the input $I = I_1 \circ I_2 \circ I_3$, where

$I_1 = (1 \nearrow (M - 1)) \otimes M, \ I_2 = (M^2 \searrow M^2 - M + 1)$

$I_3 = (M - 1 \searrow 1) \circ (M^2 + 2 \nearrow M^2 + M + 1)$.

Then,

$r_1 = ((1 \nearrow (M - 1)) \otimes M) \circ (M^2 - M + 1 \nearrow M^2 + M + 1)$

$r_2 = (M^2 \searrow M^2 - M + 1) \circ ((M - 1 \searrow 1) \otimes M) \circ (M - 1 \searrow 1)$.

Thus, we have $|r_1| = 3M$ and $|r_2| = 3M - 1$.

The following lemma sheds some light on the choices made by an optimal algorithm with respect to that of greedy. It says, roughly, that if at any decision point, an optimal algorithm chooses to write the non-greedy run, and then writes the next run in the opposite direction, it performs no better than an optimal algorithm which chooses the greedy run in the first place.
Lemma 16. At any decision time step consider two possible next maximal runs \( r_1 \) and \( r_2 \). If \( |r_1| \geq |r_2| \), then one of the following is the prefix of an optimal output on the unwritten-element sequence:

1. \( r_1 \circ r_3 \) where \( r_3 \) is a maximal run after \( r_1 \) and it can be either up or down.
2. \( r_2 \circ r_4 \) where \( r_4 \) is maximal run after \( r_2 \) with the same direction of \( r_2 \).

5.2 Online Algorithms with Resource Augmentation

We now present several online algorithms which use resource augmentation (buffer or visibility) to determine an up-down replacement selection strategy, beating the competitive ratio of 2. For a concise summary of results, see Figure 1.

Matching OPT using \( 4M \)-size Buffer. We present an algorithm with \( 4M \)-size buffer that writes no more runs than an optimal algorithm with an \( M \)-size buffer. Later on, we prove that \((4M-2)\)-size is necessary even to be \( 3/2 \)-competitive; thus this augmentation result is optimal up to a constant.

Consider the following deterministic algorithm with a \( 4M \)-size buffer. The algorithm reads elements until its buffer is full. It then uses the contents of its buffer to determine, for an algorithm with buffer size \( M \), if the maximal up run or the maximal down run would be longer. If the maximal up run is longer, the algorithm uses its full buffer (of size \( 4M \)) to write a maximal up run; otherwise it writes a maximal down run. The algorithm stops when there is no element left to write.

Theorem 17. Let \( A \) be the algorithm with a \( 4M \)-size buffer described above. On any input \( I \), \( A \) never writes more runs than an optimal algorithm with buffer size \( M \).

Proof Sketch. At each decision point, \( A \) determines the direction that a greedy algorithm on the same unwritten-element sequence, but with a buffer of size \( M \), would have picked. It is able to do so using its \( 4M \)-size buffer because, by Lemma 14, we know the length of the non-greedy run is bounded by \( 3M \). Note that it does not need to write any elements during this step. In each partition, \( A \) writes a maximal run \( r \) in the greedy direction and thus covers the greedy run by Lemma 2. Furthermore, \( r \) covers the non-greedy run as well since all of the elements of this run must already be in \( A \)'s initial buffer and hence get written out. An optimal algorithm (with \( M \)-size buffer), on the unwritten-element-sequence, has to choose between the greedy and the non-greedy run. Since \( A \) covers both choices of the optimal in one run, by Theorem 7, it is able to match or beat OPT.

A natural question is whether resource augmentation boosts performance automatically, without using the run-simulation technique. However, the following example shows that our \( 2 \)-competitive algorithm, even when allowed to have \( 4M \)-size buffer, may still be as bad when using \( M \)-size buffer.

Example 18. Consider the input, \((8M \downarrow 1) \circ (16M \downarrow 8M + 1) \circ \cdots \circ (8cM \downarrow 8(c - 1)M + 1)\). The alternating algorithm from Section 4.1 which alternates maximal up and maximal down runs will write \( 2c \) runs given a \( 4M \)-size buffer. In contrast, the optimal number of runs with an \( M \)-size buffer has \( c \) runs.

3/2-competitive using \( 4M \)-visibility. When we say that an algorithm has \( X \)-visibility (\( X \geq M \)) or \( (X - M) \)-lookahead, it means that the algorithm has knowledge of the next \( X \) elements of its unwritten element sequence, and can use this knowledge when deciding what to write.

However, only the usual \( M \)-size buffer is used for reading and writing. Furthermore, the algorithm must continue to read elements into its buffer sequentially from \( I \), even if it sees elements further down the stream it would like to read or rearrange instead.

We present a deterministic algorithm which uses \( 4M \)-visibility to achieve a competitive ratio of 3/2. At each decision point, similar to the algorithm in Theorem 17 we can use \( 3M \)-lookahead to determine the direction leading to the longer (greedy) run. However, unlike Theorem 17 we cannot use a large buffer to
write future elements. Instead, we do the following—write a maximal greedy run, followed by two additional maximal runs in the same direction and opposite direction respectively.

We show that, at each decision point, the above algorithm is able to cover two runs of optimal (on the unwritten-element-sequence) using three runs. Lemma 16 and Lemma 6 are key in this analysis (see Appendix A for details). Thus, we have the following.

**Theorem 19.** Let $OPT$ be the optimal number of runs on input $I$ given an $M$-size buffer, where $I$ has no duplicate elements. Then there exists an online algorithm $A$ with an $M$-size buffer and $4M$-visibility such that $A$ always outputs $S$ satisfying $R(S) \leq (3/2)OPT$.

**7/4-competitive using 2M-size buffer.** We have seen that it is possible to achieve a competitive ratio of $3/2$ using a standard $M$-size buffer as long as the algorithm is able to determine the direction leading to the longer (greedy) run (see Theorem 19). Now we only have a $2M$-size buffer. The algorithm will pick a direction randomly, and write a maximal run in that direction using its regular $M$ buffer. It use the additional $M$-size buffer to simulate a run in the opposite direction (and thus figure out which one is longer).

With probability $1/2$, the algorithm is lucky and picks the greedy direction. In this case, we can cover the first two runs of optimal (on the unwritten-element sequence) with three runs as in Theorem 19. With probability $1/2$, the algorithm picks the wrong direction and we spend four (alternating) runs to cover two runs of optimal. Thus, in expectation we achieve a competitive ratio of $1/2(3/2) + 1/2(4/2) = 7/4$.

**Theorem 20.** Let $OPT$ be the optimal number of runs on input $I$ given an $M$-size buffer, where $I$ has no duplicate elements. Then there exists an online algorithm $A$ with a $2M$-size buffer such that $A$ always outputs $S$ satisfying $\mathbb{E}[R(S)] \leq (7/4)OPT$ and $R(S) \leq 2OPT$.

### 5.3 Lower Bound for Resource Augmentation

We show that with less than $(4M - 2)$-augmentation, no deterministic online algorithm can be $3/2$-competitive on all inputs. Thus, an algorithm with $(4M - 2)$-size buffer cannot be optimal, so Theorem 17 is nearly tight. Similarly, Theorem 19 is nearly tight, since $4M - 2$-size buffer implies $4M - 2$-visibility.

**Theorem 21.** With buffer size less than $(4M - 2)$, for any deterministic online algorithms $A$, there exists an input $I$ such that if $S$ is the output of $A$ on $I$, then $R(S) \geq (3/2)OPT$.

### 6 Offline Algorithms for Run Generation

We give offline algorithms for run generation. The offline problem is the following—given the entire input, compute (using a standard polynomial computation time algorithm) the optimal strategy which when executed by a run generation algorithm (with a buffer of size $M$) produces the minimum possible number of runs.

For any $\varepsilon$, we provide an offline polynomial time approximation algorithm that gives a $(1 + \varepsilon)$-approximation to the optimal solution. This is called a polynomial-time approximation scheme, or PTAS. The

| Buffer size | Lookahead | Competitive ratio | Comments |
|-------------|-----------|-------------------|----------|
| $M$         | -         | 2                 | Deterministic |
| $2M$        | -         | 1.75              | Randomized |
| $M$         | $3M$      | 1.5               | Deterministic |
| $4M$        | -         | 1                 | Deterministic |

Figure 1: Summary of online algorithms on run generation on any input
running time of our first attempt is $O(2^{1/\varepsilon}N \log N)$. We then improve the running time to $O(\varphi^{1/\varepsilon}N \log N)$ where $\varphi = (1 + \sqrt{5})/2 \approx 1.618$ is the well-known golden ratio.

**Simple PTAS.** Our first attempt breaks the output into sequences with a small number of runs, and uses brute force to find which set of runs writes the most elements. We show that for any $\varepsilon$, we can achieve a $1 + \varepsilon$ approximation in polynomial time using this strategy.

**Theorem 22.** There exists an offline algorithm $A$ that always writes an $S$ satisfying $R(S) \leq (1 + \varepsilon) \cdot \text{OPT}$. The running time of $A$ is $O(2^{1/\varepsilon}N \log N)$.

**Improved PTAS.** We reduce the running time by bounding the number of choices we need to consider in a brute-force search. We do this using Lemma 16.

At each decision point, an algorithm chooses between starting an increasing run $r_1$ and a decreasing run $r_2$. If $|r_1| \geq |r_2|$, then by Lemma 16, we are able to discard $r_2$ followed by an increasing run.

Let $F_d$ be the number of run sequences we need to consider if $d$ runs remain to be written (for example, naïve PTAS has $F_d = 2^d$). First, the algorithm must handle all run sequences beginning with $r_1$; this is the same as an instance of $F_{d-1}$. Then the algorithm handles all run sequences beginning with $r_2$ followed by a decreasing run; this is an instance of $F_{d-2}$. Thus $F_d = F_{d-1} + F_{d-2}$; by examination, $F_1 = 1$ and $F_2 = 2$. This is the Fibonacci sequence, which gives us the $\varphi^{1/\varepsilon}$ factor in the running time.

**Theorem 23.** There exists an offline algorithm $A$ that writes $S$ such that $R(S) \leq (1 + \varepsilon) \cdot \text{OPT}$. The running time of $A$ is $O(\varphi^{1/\varepsilon}N \log N)$ where $\varphi$ is the golden ratio $(1 + \sqrt{5})/2$.

### 7 Run Generation on Nearly Sorted Input

This section presents results proving that up-down replacement selection performs better when the input has inherent sortedness (or “bounded-disorder” [14]). Replacement selection produces longer runs on nearly sorted data. In particular, if every input element is $M$ away from its target position, then a single run is produced. Similarly, we give algorithms which perform well on inputs, where the optimal runs are also long.

In particular, we say that an input is $c$-nearly-sorted if there exists a proper optimal algorithm whose outputs consists of runs of length at least $cM$.

**3/2-competitive using $2M$-size Buffer.** We provide a randomized online algorithm that, on inputs which are $3$-nearly-sorted, achieves a competitive ratio of $3/2$, while using an augmented-buffer of size $2M$.

A sketch of the algorithm follows. At each decision point, the algorithm picks a run direction at random. It starts a maximal run in that direction, but uses its extra $M$-buffer to simulate the run in the opposite direction. By Lemma 14, the algorithm can tell if it picked the same run as greedy (with $M$-buffer), similar to Theorem 20. If the algorithm got lucky and picked the greedy run, it repeats the process.

If the algorithm picked the non-greedy run, it uses some careful bookkeeping to write elements and simulate the run in the opposite direction. In doing so, the algorithm winds up at the same point in the input it would have reached, had it written the greedy run in the first place, but with an additional cost of one run.

**Theorem 24.** There exists a randomized online algorithm $A$ using $M$ space in addition to its buffer such that, on any $3$-nearly-sorted input $I$ that has no duplicates, $A$ is a $3/2$-approximation in expectation. Furthermore, $A$ is at worst a $2$-approximation regardless of its random choices.

**Exact Offline Algorithm on Nearly Sorted Input.** We show that the greedy (offline) algorithm is a linear time optimal algorithm on inputs which are $5$-nearly-sorted. We first prove the following lemma.

**Lemma 25.** If a proper algorithm produces runs of length at least $5M$ on a given input with no duplicates, then it is optimal.
Thus, we get our required linear time exact offline algorithm.

**Theorem 26.** The greedy offline algorithm, i.e., picking the longer run at each decision point, is optimal on a 5-nearly-sorted input that contain no duplicates. The running time of the algorithm is $O(N)$.

### 8 Additional Related Work

**Replacement Selection.** The classic algorithm for run generation is replacement selection [20]. While replacement selection considers only up runs, Knuth [25] analyzed alternating up-down replacement selection in 1963. He showed that for uniformly random input, alternating up-down replacement selection produces runs of expected length $3M/2$, compared to $2M$ of the standard replacement selection [18, 19, 26].

Recently, Martinez-Palau et al. [34] introduce Two-way replacement selection (2WRS), reviving the idea of up-down replacement selection. The 2WRS algorithm maintains two heaps in memory for up and down runs and heuristically decides in which heap each element must be placed. Their simulations show that 2WRS performs significantly better on inputs with mixed up-down, alternating up-down, and random sequences.

Replacement selection with a fixed-sized reservoir appears in [17, 40]. Larson [28] introduced batched replacement selection, a cache-conscious replacement selection which works for variable-length records. Koltsidas, Müller, and Viglas [27] study replacement selection for sorting hierarchical data.

Improvements for the merge phase of external sorting have been considered in [10, 15, 37, 43, 44], but this is beyond the scope of this paper.

**Reordering Buffer Management.** Run generation problem is reminiscent of the buffer reordering problem (also known as the sorting buffer problem), introduced by Räcke et al. [36]. It consists of a sequence of $n$ elements that arrive over time, each having a certain color. A buffer, that can store up to $k$ elements, is used to rearrange them. When the buffer becomes full, an element must be output. A cost is incurred every time an element is output that has a color different from the previous element in the output sequence. The goal is to design a scheduling strategy for the order in which elements must be output, so as to minimize the total number of color changes. The buffer reordering problem models a number of important problems in manufacturing processes and network routing and has been extensively studied, both in the online and offline case [3, 7, 9, 12, 23]. The offline version of the buffer reordering problem is NP hard [9], while the complexity of our problem remains unresolved.

**Patience Sort and Longest Increasing Subsequence.** An old sorting technique used to sort decks of playing cards, Patience Sort [33] has two phases—the creation of sorted piles or runs, and the merging of these runs. The elements arrive one at a time and each one can be added to an existing run or starts a new run of its own. Unlike this paper, a legal run only consists of elements decreasing in value, and patience sort can form any number of parallel runs. The goal is to minimize the number of runs. The greedy strategy of placing an element to the left-most legal run is optimal. Moreover, the minimum number of such runs is the length of the longest increasing subsequence of the input [2]. Patience sort has been studied in the streaming model [21].

Similar to Replacement Selection, Patience Sort is able to leverage partially sorted input data. Chandramouli and Goldstein [10] present improvements to patience sort, and combine it with replacement selection to achieve practical speed up.

**Adaptive Sorting Algorithms.** Python’s inbuilt sorting algorithm, Timsort [41] works by finding contiguous runs of increasing or decreasing value during the run generation phase. External memory sorting for well-ordered or “partially sorted” data has been studied by Liu et al. [32]. They minimize the I/O cost of run generation phase by finding “naturally occurring runs”. See [16] for a survey on adaptive sorting algorithms.
9 Conclusion and Open Problems

In this paper, we present an in-depth analysis of algorithms for run generation. We establish that considering both up and down runs can substantially reduce the number of runs in an external sort. The notion of up-down replacement selection has received relatively little attention since Knuth’s negative result \cite{knuth}, until its promise was acknowledged by the experimental work of Martinez-Palau et al. \cite{martinez-palau}.

The results in our paper complement the findings of Knuth \cite{knuth} and Martinez-Palau et al. \cite{martinez-palau}. In particular, strict up-down alternation being the best possible strategy explains why heuristics for up-down run-generation can lead to better performance in some cases. Moreover, our constant-factor competitive ratios with resource augmentation and lookahead may guide followup heuristics and practical speed-ups.

We conclude with open problems.

Can randomization help circumvent the lower bound of $2$ on the competitive ratio of online algorithms (without resource augmentation)? We know that no randomized online algorithm can have a competitive ratio better than $3/2$, but there is still a gap. What is the performance of the greedy offline algorithm compared to optimal? We show that greedy can as bad as $3/2$ times optimal. Is there a matching upper bound? Can we design a polynomial, exact, algorithm for the offline run-generation problem? We find it intriguing that our attempts at an exact dynamic program requires maintaining too many buffer states to run in polynomial time.

10 Acknowledgments

We gratefully acknowledge Goetz Graefe and Harumi Kuno for introducing us to this problem and for their advice. This research was supported by NSF grants CCF 1114809, CCF 1217708, IIS 1247726, IIS 1251137, CNS 1408695, CCF 1439084, and by Sandia National Laboratories.

References

[1] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[2] David Aldous and Persi Diaconis. Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem. *Bulletin of the American Mathematical Society*, 36(4):413–432, 1999.

[3] Yuichi Asahiro, Kenichi Kawahara, and Eiji Miyano. NP-hardness of the sorting buffer problem on the uniform metric. *Discrete Applied Mathematics*, 160(10):1453–1464, 2012.

[4] Noa Avigdor-Elgrabli and Yuval Rabani. A constant factor approximation algorithm for reordering buffer management. In *Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 973–984, 2013.

[5] Noa Avigdor-Elgrabli and Yuval Rabani. An improved competitive algorithm for reordering buffer management. In *Proc. 54th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 1–10, 2013.

[6] Noa Avigdor-Elgrabli and Yuval Rabani. An optimal randomized online algorithm for reordering buffer management. *arXiv preprint arXiv:1303.3386*, 2013.

[7] Reuven Bar-Yehuda and Jonathan Laserson. Exploiting locality: approximating sorting buffers. *Journal of Discrete Algorithms*, 5(4):729–738, 2007.
[8] Dina Bitton and David J DeWitt. Duplicate record elimination in large data files. *ACM Transactions on database systems*, 8(2):255–265, 1983.

[9] Ho-Leung Chan, Nicole Megow, René Sitters, and Rob van Stee. A note on sorting buffers offline. *Theoretical Computer Science*, 423:11–18, 2012.

[10] Badrish Chandramouli and Jonathan Goldstein. Patience is a virtue: Revisiting merge and sort on modern processors. In *Proc. 2014 ACM SIGMOD Int’l Conference on Management of Data*, pages 731–742, 2014.

[11] Chandra Chekuri, Ashish Goel, Sanjeev Khanna, and Amit Kumar. Multi-processor scheduling to minimize flow time with $\varepsilon$ resource augmentation. In *Proc. of the 36th annual ACM Symposium on Theory of Computing (STOC)*, pages 363–372. ACM, 2004.

[12] Matthias Englert and Matthias Westermann. Reordering buffer management for non-uniform cost models. In *Proc. 32nd Int’l Colloquium on Automata, Languages and Programming (ICALP)*, pages 627–638, 2005.

[13] Leah Epstein and Rob Van Stee. Online bin packing with resource augmentation. *Discrete Optimization*, 4(3):322–333, 2007.

[14] Terje O Espelid. On replacement selection and dinsmore’s improvement. *BIT Numerical Mathematics*, 16(2):133–142, 1976.

[15] Vladimir Estivill-Castro and Derick Wood. Foundations for faster external sorting. *Foundation of Software Technology and Theoretical Computer Science*, 880:414–425, 1994.

[16] Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys (CSUR)*, 24(4):441–476, 1992.

[17] WD Frazer and CK Wong. Sorting by natural selection. *Communications of the ACM*, 15(10):910–913, 1972.

[18] Edward H Friend. Sorting on electronic computer systems. *Journal of the ACM*, 3(3):134–168, 1956.

[19] Betty Jane Gassner. Sorting by replacement selecting. *Communications of the ACM*, 10(2):89–93, 1967.

[20] Martin A Goetz. Internal and tape sorting using the replacement-selection technique. *Communications of the ACM*, 6(5):201–206, 1963.

[21] Parikshit Gopalan, TS Jayram, Robert Krauthgamer, and Ravi Kumar. Estimating the sortedness of a data stream. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 318–327, 2007.

[22] Goetz Graefe. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, 38(3):10, 2006.

[23] Sungjin Im and Benjamin Moseley. New approximations for reordering buffer management. In *Proc. 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1093–1111, 2014.

[24] Tom Keller, Goetz Graefe, and David Maier. Efficient assembly for complex objects. In *Proc. 2014 ACM SIGMOD Int’l Conference on Management of Data*, pages 148–157, 1991.

[25] Donald Ervin Knuth. Length of strings for a merge sort. *Communications of the ACM*, 6(11):685–688, 1963.
[26] Donald Ervin Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Pearson Education, 1998.

[27] Ioannis Koltsidas, Heiko Müller, and Stratis D Viglas. Sorting hierarchical data in external memory for archiving. *Proceedings of the VLDB Endowment*, 1(1):1205–1216, 2008.

[28] Per-Åke Larson. External sorting: Run formation revisited. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):961–972, 2003.

[29] Per-Åke Larson and Goetz Graefe. Memory management during run generation in external sorting. In *Proc. 1998 ACM SIGMOD Int’l Conference on Management of Data*, volume 27, pages 472–483, 1998.

[30] Yen-Chun Lin. Perfectly overlapped generation of long runs for sorting large files. *Journal of Parallel and Distributed Computing*, 19(2):136–142, 1993.

[31] Yen-Chun Lin and Horng-Yi Lai. Perfectly overlapped generation of long runs on a transputer array for sorting. *Microprocessors and Microsystems*, 20(9):529–539, 1997.

[32] Yang Liu, Zhen He, Yi-Ping Phoebe Chen, and Thi Nguyen. External sorting on flash memory via natural page run generation. *The Computer Journal*, 54(11):1882–1990, 2011.

[33] Colin L Mallows. Patience sorting. *Bulletin of Inst. of Math. Appl.*, 5(4):375–376, 1963.

[34] Xavier Martinez-Palau, David Dominguez-Sal, and Josep Lluis Larriba-Pey. Two-way replacement selection. In *Proc. of the VLDB Endowment*, volume 3, pages 871–881, 2010.

[35] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[36] Harald Räcke, Christian Sohler, and Matthias Westermann. Online scheduling for sorting buffers. In *Proc. 10th European Symposium on Algorithms (ESA)*, pages 820–832, 2002.

[37] Betty Salzberg. Merging sorted runs using large main memory. *Acta Informatica*, 27(3):195–215, 1989.

[38] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[39] Hongyang Sun and Rui Fan. Improved semi-online makespan scheduling with a reordering buffer. *Information Processing Letters*, 113(12):434–439, 2013.

[40] TC Ting and YW Wang. Multiway replacement selection sort with dynamic reservoir. *The Computer Journal*, 20(4):298–301, 1977.

[41] Wikipedia. Timsort, 2004. http://en.wikipedia.org/wiki/Timsort.

[42] Weiye Zhang and Per-Åke Larson. Dynamic memory adjustment for external mergesort. In *Proc. of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 376–385. Morgan Kaufmann Publishers Inc., 1997.

[43] Weiye Zhang and Per-Åke Larson. Buffering and read-ahead strategies for external mergesort. In *Proc. 24rd Int’l Conference on Very Large Data Bases (VLDB)*, pages 523–533, 1998.

[44] LuoQuan Zheng and Per-Åke Larson. Speeding up external mergesort. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):322–332, 1996.
A Appendix: Omitted proofs

Proof of Lemma[2] Without loss of generality, assume \( r \) and \( r' \) are increasing runs. Consider any time step \( t \) when elements from both \( r \) and \( r' \) are being written. Let \( B_t \) and \( B'_t \) be the buffer of \( r \) and \( r' \) at time step \( t \) respectively; let \( C_t \) be the set of elements in \( B_t \) that are eventually written to \( r \), and \( C'_t \) be the set of elements in \( B'_t \) that are eventually written to \( r' \). We prove inductively that \( C'_t \subseteq C_t \). This implies that \( r \) covers \( r' \). The base case is true as \( r \) and \( r' \) start with the same buffer. Since we have \( C'_t \subseteq C_t \), we must show that (a) the element \( z \) written to \( r \) is not in \( C'_t \) and (b) the element \( e \) read into \( C'_t \) must also be in \( C_t \).

Consider the elements \( z' \) and \( z \) written by \( r' \) and \( r \), respectively, at time \( t + 1 \). We must have \( z' \geq z \); thus either \( z = z' \), or \( z \) is never written to \( r' \) (either way it is not in \( C'_t \)).

Since \( e \) is in \( C'_t \), it is eventually written by \( r' \); thus \( e \geq z' \). Thus \( e \geq z \), but that means \( e \) is eventually written by \( r \). Since \( e \) was just read it is in \( B_t + 1 \); thus \( e \in C_t \). \( \square \)

Observation A. If \( A \) has just written an element \( e \), and is writing a down (up) run, then \( A \) cannot write any element larger (smaller) than \( e \) in the same run. Similarly, if \( A \) has just written \( e \), then \( A \) cannot write both an element larger than \( e \) and an element smaller than \( e \) in the same run.

Proof of Theorem[10] Let \( I_t \), the first \( M \) elements of the input, be \( I_1 = 1, 2, \ldots, M \). We divide the rest of the input into segments of size \( M \). Let the \((t+1)\)st such segment be \( I_{t+1} \). Then,

\[
I_{t+1} = (1 + tM \uparrow M + tM) \text{ or } (-(1 + tM) \downarrow -(M + tM)).
\]

Call this a positive segment and a negative segment respectively. At time \( M(t - 1) + 1 \) we decide whether \( I_{t+1} \) is a positive or a negative segment based on \( A \).

Specifically, we choose \( I_{t+1} \) using either the direction of the run \( A \) is writing, or the value of the most recent element written. If \( A \) is writing a down run, \( I_{t+1} \) is a positive segment; if \( A \) is writing an up run, \( I_{t+1} \) is a negative segment. It may be that \( A \) has only written one element of a run (so \( A \) could turn this into either an up run or a down run). If this element was the smallest element in the buffer of \( A \), \( I_{t+1} \) is a negative segment. Otherwise, \( I_{t+1} \) is a positive segment.

First we show that \( A \) must write at least one new run for each \( I_t \); thus \( R(S) \geq t \). At least one run is required for \( A \) to write \( I_1 \), so for the remainder of the proof we assume \( t > 1 \). Consider time \( M(t - 2) + 1 \), when \( I_t \) begins. We assume that \( I_t \) is a positive segment—a mirroring argument works when \( I_t \) is a negative segment. Furthermore, note that the elements of \( I_t \) are the largest in the instance so far.

There are two cases: \( A \) is currently writing a down run, or the initial element of a new run.

Case 1: Algorithm \( A \) is currently writing a down run. Then the elements of \( I_t \) must be larger than any element in \( A \)'s down run. Thus \( A \) must use another run to write the elements of \( I_t \) by Observation[A].

Case 2: Algorithm \( A \) is writing the initial element of a new run. By construction, the element written is not the smallest element in \( A \)'s buffer, but is smaller than all elements in \( I_t \). Then \( A \) must spend one run to write the smallest element in its buffer, and another to write \( I_t \). Thus, \( I_t \) causes \( A \) to write a run in addition to its current run by Observation[A].

On the other hand, an offline algorithm can write \( I_{2i} \) and \( I_{2i+1} \) in one run. Assume that \( I_{2i} \) is a positive segment—a mirroring argument works when \( I_{2i} \) is a negative segment. If \( I_{2i+1} \) is positive, both can be written using an up run. If \( I_{2i+1} \) is negative, both can be written using a down run. Thus OPT is no more than \([t/2]\). \( \square \)

Proof of Theorem[11] Our lower bound uses the same basic principles as Theorem[10]. We first show the lower bound with some repeated elements, then show how to perturb the elements to avoid repetitions. We generate a randomized input and show that any deterministic algorithm cannot perform better than \( 3/2 \) times OPT on that input. The theorem is then proven by Yao’s minimax principle. Yao’s minimax principle states
that the best expected performance of a randomized algorithm is at least as large as the expected performance of the best deterministic algorithm over a (known) distribution of inputs. (See, e.g., [35] for details.)

As in Theorem 10, we divide the input into segments of size $4M$. Call these segments $I_t$ for $t = 1, 2, \ldots, \lceil N/4M \rceil$. Note that this input is randomized: for each $t$, we pick one of two inputs, each with probability $1/2$. We choose either

$I_t^1 = (1 \uparrow M)$ \quad $I_t^1 = (1 \uparrow M)$

$I_t^2 = (2M, \downarrow M + 1)$ \quad $I_t^2 = (-2M, \uparrow -M - 1)$

$I_t^3 = (3M, \downarrow 2M + 1)$ \quad $I_t^3 = (-3M, \uparrow -2M - 1)$

$I_t^4 = (4M, \downarrow 3M + 1)$ \quad $I_t^4 = (-4M, \uparrow -3M - 1)$

(a positive segment) \quad (a negative segment)

Let $I_t = I_t^1 \odot I_t^2 \odot I_t^3 \odot I_t^4$. A positive or negative segment is chosen randomly for each $I_t$ with probability 1/2.

The optimal algorithm spends no more than one run per $I_t$, using an up run for a positive segment or a down run for a negative segment.

We show that any deterministic algorithm requires at least one new run to write $I_{t-1}^4$ and $I_t^4$ for $t > 1$. Further analysis shows that with probability $1/2$, any deterministic algorithm requires at least one run to write the remainder of $I_1^4$, $I_2^4$, and $I_3^4$. Note that one run is also required to write $I_1^4$; summing, this gives a total expected cost of $(3/2)OPT$.

Consider a segment $I_t$; $t > 1$. Once all of $I_{t-1}^4$ has been read into its buffer, at least one element of $I_{t-1}^4$ has been written. Once all of $I_t^4$ has been read into its buffer, at least one element of $I_{t-1}^4$ has been written. Finally, once all of $I_t^4$ has been read into its buffer, at least one element of $I_t^4$ has been written. Applying Observation A at least one new run is required to write these three elements.

Now we show that with probability 1/2, an additional run is required to write $I_t^2$. Let $x$ be the first element written by $I_t^1$ (thus, the cost of writing $x$ itself was handled in the above case—we show when an additional run is required). Note that the algorithm must choose an $x$ before it sees any element of $I_t^2$ (so it does now know if $I_t^1$ is positive or negative).

Let $x \neq 1$ and $I_t$ be a positive segment. By Observation A, an additional run is required to write both 1 and any element of $I_t^2$. If 1 is not written, all of $I_t^2$ cannot be stored in the buffer—but then, $I_t^2$ and $I_t^3$ cannot be written using one run. Similarly, let $x = 1$ and $I_t$ be a negative segment. By Observation A, an additional run is required to write both $M$ and any element of $I_t^2$; otherwise $I_t^2$ and $I_t^3$ require an additional run to be written.

Thus any deterministic algorithm cannot perform better than a $3/2$-approximation. Applying Yao’s min-max principle proves the theorem.

Now we perturb the input to avoid duplicate elements. We multiply each element by $\lceil N/4M \rceil$, and add $t$ to each element of $I_t$. In other words, we use a new segment

$I'_t = (I_t \otimes \lceil N/4M \rceil) \oplus t$.

Our arguments above only depended on the relative ordering of the elements, which is preserved by this perturbation. For example, assume $I_t$ and $I_{t-1}$ are both positive segments. Then all elements of $I_t^2$ are less than all elements of $I_{t-1}^2$, and all elements of $I_t^3$ are greater than all elements of $I_{t-1}^4$.

**Proof of Theorem 13** We will build $S$ constructively. At each time $t$ where greedy chooses an up or down run, we show that one of its choices leads to a run of length at least $5M/4$. Since the greedy algorithm always picks the longer run at each decision time step, the run with length less than $5M/4$ can never be part of its output.

Consider any time step $t$ where $t < N - 5M/4$. If $t$ is larger than this value, the final run will have length $n - t$. Let the contents of buffer at time $t$ be $\sigma_1, \ldots, \sigma_M$. Let $I'$ be the sequence of $\lfloor M/2 \rfloor$ elements of $I$ arriving after $t$. 

17
Consider the run starting at $\sigma_M$ and continuing downwards; call this down run $r_1$. Let $r_2$ be the up run starting at $\sigma_1$ and continuing upwards. Any element of $I'$ less than $\sigma_{[M/2]}$ will be (eventually) written out to $r_1$; any that are greater than $\sigma_{[M/2]}$ will be written out to $r_2$. Every number must fall into one of these categories, so there must be at least \( \lceil M/2 \rceil \) numbers added to the larger run. Each run at least includes the elements already in the buffer, so the larger run has length at least $M + \lceil M/2 \rceil$.

The last $5M/4$ elements can be handled by greedy in at most 2 runs (since each trivially has length at least $M$). Thus the above applies to all but the last two runs of greedy.

**Proof of Lemma**. Let $S_1$ be an output that writes $r_1$ initially and $S_2$ be an output that writes $r_2$ initially. Without loss of generality, suppose that $r_1$ is increasing and $r_2$ is decreasing. Let $r_1 = r_1(1), r_1(2), \ldots, r_1(k)$ and $r_2 = r_2(1), r_2(2), \ldots, r_2(\ell)$. The idea of the proof is to split these runs into two phases (a) elements of $r_1$ are smaller than the corresponding elements of $r_2$ and (b) when the elements of $r_1$ are larger than or equal to those of $r_2$. During each of these two phases, we use the fact that incoming elements written by $S_1$ have to be in the buffer of $S_2$ (and vice versa) to bound their length. We assume that both runs write exactly one element for each element they read in; this cannot affect the length of the runs.

Let $B_0$ be the original buffer, i.e., the first $M$ elements of $I$. Let $i$ be the transition point between the two phases mentioned above; in other words, $r_1(i+1) \geq r_2(i+1)$ but $r_1(i) < r_2(i)$.

Divide $r_1$ into $s_1$ and $t_1$, where $s_1$ is the first $i$ elements of $r_1$, and $t_1$ is the remainder of $r_1$. We further divide $s_1$ into $s_1^B$, the elements of $s_1$ that are in $B_0$, and $s_1^N$, the elements of $s_1$ that are not in $B_0$. Let $t_1^B$ be the elements of $t_1$ that are in $B_0$. Let $f_1$ be the set of elements in $r_1$ that are read in after $x_{i+1}$ is written. Let $u_1$ be the set of elements not in $r_1$ that are read in before $x_{i+1}$ is written. We define the corresponding sets for $r_2$ as well: $s_2^B, s_2^N, t_2^B, f_2, r_2, u_2$.

We can bound the size of several of these sets by $M$. Note that $s_1$ cannot have more than $M$ elements, since all must be stored in the buffer while $s_2$ is being written. Thus $|s_1^N| + |s_1^B| \leq M$. Similarly, $|s_2^N| + |s_2^B| \leq M$. We also must have $|u_1| \leq M$ and $|u_2| \leq M$. Finally, consider $s_{2}^N \cup t_{1}^B$. Any element in $s_{2}^N$ must be read before time step $i$. Since $s_1$ is disjoint from $s_2$ (by definition of $i$), all elements of $s_{2}^N$ must be in $S_1$’s buffer at time step $i$. All elements of $t_{1}^B$ must also be in $S_1$’s buffer at time step $i$, so $|s_{2}^N| + |t_{1}^B| \leq M$.

Starting from time step $i+1$, any new element $e$ that is read in cannot be in both $r_1$ and $r_2$. This means that all elements of $f_1$ must be in the buffer of $S_2$ until $r_2$ ends, and all elements of $f_2$ must be in the buffer of $S_1$ until $r_1$ ends. On the other hand, all elements of $u_2$ must eventually be a part of $r_1$, and similarly for $u_1$ and $r_2$.

To begin, we show a weaker version of the lemma for runs of length $4M$. We have $|r_1| \leq (|s_1^N| + |s_1^B|) + (|u_2|) + (|s_2^N| + |t_1^B|) + |f_1| \leq 3M + |f_1|$. Then if $|r_1| \geq 4M$, then $|f_1| \geq M$. Since all elements of $f_1$ must be stored in the buffer of $S_2$ until $r_2$ ends, $r_2$ must end when the $M$th element of $f_1$ is read in. Then we must have $|r_2| < |r_1|$.

We have $|f_1| \geq M - |u_2|$; otherwise, $|r_1| \leq (|s_1^N| + |s_1^B|) + |s_2^N| + |t_1^B| + |f_1| < 3M + |f_1|$. Consider the first $M - |u_2| - 1$ elements read in after $i$ that are eventually written to $r_1$ (this is a prefix of $f_1$), call them $f_1'$. Since $|f_1| \geq M - |u_2|$, there must be another element $e \in f_1$ that is read after all elements of $f_1'$. Note $e \in r_1$. Let $t$ be the time when $e$ arrives.

At $t$, the buffer of $S_2$ must contain all elements of $u_2$, as well as all elements of $f_1'$ and $e$. The buffer of $S_2$ is then full of elements that cannot be written in $r_2$. Hence, $S_2$ is forced to start a new run at time $t < |r_1|$, so $|r_2| < |r_1|$. Then we must have $|f_2| + |u_1| < M$. None of the elements in these sets are in $r_1$, and must be stored in $S_1$’s buffer until $r_1$ ends (which is after $r_2$ ends). Finally, we have,

$$|r_2| \leq (|s_2^N| + |s_2^B|) + (|s_1^N| + |t_2^B|) + (|u_1| + |f_2|) < 3M,$$

as required.
Proof of Theorem 17. Algorithm $A$ will simulate the maximal up run, $r_1$, and maximal down run, $r_2$, to see which is longer, but it does not actually need to write any elements during this simulation. By Lemma 14 if we find that one run has length at least $3M$, it must be the longer run.

We now describe exactly how to simulate a run $r$ using $4M$ space without writing any elements. Algorithm $A$ simulates the run one step at a time. We describe the actions and buffer of an algorithm with $M$-size buffer writing $r$ as the simulated algorithm. Without loss of generality, assume $r$ is an up run.

Assume that all elements are stored in the buffer in the order they arrive. Thus, after all elements are stored in the buffer, algorithm $A$ will simulate the maximal up run, $r$. Consider two algorithms $I$ and $B$, in which $I$ are also written to $r$ large as $r$ and $B$ contains all elements of $r$.

Specifically, at time step $t$, $A$ finds the smallest element $e$ in the first $M + t$ elements of the buffer that is larger than $\ell$. This is the next element of $r$. Thus in the next time step, $A$ updates $\ell \leftarrow e$, and repeats. If no such $e$ can be found, no element in $M + t$ (and thus no element in the buffer) can continue the run, so the run ends at time $t$.

The last time $A$ can update $\ell$ is when the simulated algorithm has seen all elements in the buffer; in other words, $t = 3M$. By Lemma 14, this is sufficient to determine which run is longer.

The algorithm now knows which run is longer; without loss of generality, assume $|r_1| \geq |r_2|$. Then the algorithm writes a maximal run $r$ using its $4M$-size buffer in the direction of $r_1$. Run $r$ is guaranteed to contain all elements of $r_1$ by Lemma 14. Since $r_2$ has length less than $3M$ by Lemma 14 all of its elements must already be in the $4M$-size buffer. Thus they are written during $r$ because a maximal run always writes its buffer contents. The first initial run of a proper optimal algorithm on the unwritten-element sequence has to be either $r_1$ or $r_2$. Since $r$ covers both $r_1$ and $r_2$, by Theorem 7 with $\beta = 1$, $A$ never writes more runs than an optimal algorithm with an $M$-size buffer.

Lemma A. Consider two algorithms $A_1$ and $A_2$ that have the same remaining input $I$ when they both start writing a new maximal run, called $r_1$ and $r_2$. Let their buffers at this point be $B_1$ and $B_2$ (that may not be full) respectively, and assume $\max(B_1 \setminus B_2) \leq \min(B_2 \setminus B_1)$. If $r_1$ and $r_2$ are increasing then all elements in $I$ written to $r_2$ are also written to $r_1$. Similarly, if $r_1$ and $r_2$ are decreasing, all elements in $I$ written to $r_1$ are also written to $r_2$.

Proof. It suffices to prove the first case where $r_1$ and $r_2$ are both increasing as the other case can be proven similarly. After $r_1(t)$ and $r_2(t)$ were written, let $C_1(t)$ and $C_2(t)$ be the set of elements in the buffers of $A_1$ and $A_2$ that will be written in $r_1$ and $r_2$ at some point in the future, i.e., the set of elements that are at least as large as $r_1(t)$ or $r_2(t)$ respectively.

It is easy to prove by induction that the invariant

$$\max(C_1(t) \setminus C_2(t)) \leq \min(C_2(t) \setminus C_1(t))$$

always holds. We note that this invariant implies $r_1(t + 1) \leq r_2(t + 1)$. Therefore, if this invariant is true for all $t < \min(|r_1|, |r_2|)$, any incoming element $e \in I$ satisfies $e \in r_2 \Rightarrow e \in r_1$ as required. We now prove the invariant:

The base case is true since $C_1(0) = B_1$, $C_2(0) = B_2$. Suppose the invariant holds for $t$, then $r_1(t + 1) \leq r_2(t + 1)$ and a new element $e$ is read in.

Case 1: if $e \in C_2(t + 1) \Rightarrow e \geq r_2(t + 1) \geq r_1(t + 1) \Rightarrow e \in C_1(t + 1)$.

Case 2: if $e \in C_1(t + 1)$ and $e \notin C_2(t + 1)$, then $e < r_2(t) = \min(C_2(t)) \leq \min(C_2(t + 1))$. 

19
Proof of Lemma 16. Without loss of generality, assume \( r_1 \) and \( r_2 \) are initial maximal increasing and decreasing runs respectively and \( |r_1| \geq |r_2| \). Suppose \( r_2 \circ r_3 \), where \( r_3 \) is an increasing, is prefix of an optimal output \( S_{\text{OPT}}(I) \).

Consider the case \( |r_1| = |r_2| \). Let their buffers at the end of these two runs be \( B_1, B_2 \) and let \( j \) be the smallest index such that \( r_1(j + 1) > r_2(j + 1) \). Consider any new element \( e \in I \) that is read in before \( r_1(j + 1) \) is written. Obviously, we have:

\[
\begin{align*}
e \in B_1 & \Rightarrow e \leq r_1(j), \\
e \in B_2 & \Rightarrow e \geq r_2(j).
\end{align*}
\]

Consider any new element \( e \) which is read in after \( r_1(j + 1), r_2(j + 1) \) were written. It is easy to see the followings:

\[
\begin{align*}
e \in B_1 \setminus B_2 & \Rightarrow e \leq r_1(j + 1), \\
e \in B_2 \setminus B_1 & \Rightarrow e \geq r_2(j + 1).
\end{align*}
\]

Therefore, \( \max(B_1 \setminus B_2) \leq \max(r_1(j), r_2(j + 1)) \leq \min(r_2(j), r_1(j + 1)) \leq \min(B_2 \setminus B_1) \). The situation can be visualized in Figure 2 as follows. If an incoming element cannot be written in the current run, it lies below or above (depending on whether the run is increasing or decreasing) the last element written. The regions are marked with their associated sets described above.

Consider \( r_1 \circ r_4 \), where \( r_4 \) is a maximal increasing run. Every elements in \( r_2 \) will be written in either \( r_1 \) or \( r_4 \) by Lemma 6. If \( e \in r_3 \), then we consider the cases where \( e \in B_2 \) or \( e \in r_3 \setminus B_2 \). If \( e \in B_2 \), then \( e \) is either in \( r_1 \) or in \( B_1 \) which means \( e \) is either in \( r_1 \) or \( r_4 \). If \( e \in r_3 \setminus B_2 \), \( e \in r_4 \) by Lemma A using the fact that \( \max(B_1 \setminus B_2) \leq \min(B_2 \setminus B_1) \). Thus, \( r_1 \circ r_4 \) covers \( r_2 \circ r_3 \). As a result, \( r_1 \) is also a prefix of an optimal output \( S_{\text{OPT}}(I) \) by Observation 3.

If \( |r_1| > |r_2| \) and \( |r_2| = k \). Then the simplest argument goes as follows. Instead of arguing based on \( r_1 \) directly, we consider \( r_f \) that is increasing but may not be maximal. Consider an algorithm \( A_f \) that writes \( r_f(1) = r_1(1), \ldots, r_f(k) = r_4(k) \). Then, without reading any new element in, it finishes its first
run $r_f$ by writing out all elements in its buffer that are larger than $r_1(k)$ (the set of these elements is $H$ in Figure 3). After this extra step, let the buffer be $B_f$. Use the exact same argument as above, we have that $\max(B_f \setminus B_2) \leq \max(r_f(j), r_2(j+1)) \leq \min(r_2(j), r_f(j+1)) \leq \min(B_2 \setminus B_f)$. Using the same argument as in the first case, we have that $r_f$ followed by a maximal increasing run will cover $r_2$ followed by a maximal increasing run. Hence, $r_f$ is a prefix of an optimal output. Since $r_1$ covers $r_f$, it is also a prefix of an optimal output by Observation B.

Proof of Theorem 19. In any partition that is not the last one, let $I_r$ be the unwritten-element sequence and let $r_1, r_2$ be the two possible maximal initial runs where $r_1$ is increasing and $r_2$ is decreasing. Without loss of generality, suppose $|r_1| \geq |r_2|$. We use the simulation technique of Theorem 17 to determine which run is longer.

The algorithm stops when there is no element left to write. We break our analysis into cases based on what runs are in an optimal output. In each case, we show that Theorem 7 proves a competitive ratio of $\beta = 3/2$.

If $r_2$ is a prefix of a proper optimal output $S_{\text{OPT}}(I_r)$, let $r_5$ be the maximal run after $r_2$ in $S_{\text{OPT}}(I_r)$. After writing $r_3$, the algorithm already writes out all elements in $r_2$ by Lemma 6. Let the unwritten-element sequence after writing $r_3$ be $I_3$ and let the unwritten-element sequence after writing $r_2$ be $I_2$. By Lemma 6, $I_3$ is a subsequence of $I_2$. According to Lemma 16, $r_5$ has to be decreasing in order to possibly have fewer runs than writing $r_1$ initially. Hence, applying Lemma 8 to $r_4$ and $r_5$, we know that at the end of $r_4$, the algorithm has written all elements of $r_2$ and $r_5$. Thus, $r_1 \circ r_3 \circ r_4$ covers $r_2 \circ r_5$.

If $r_1 \circ r_3$ is a prefix of $S_{\text{OPT}}(I_R)$, then we are done as $r_1 \circ r_3 \circ r_4$ trivially covers $r_1 \circ r_3$.

If $r_1$ is a prefix of $S_{\text{OPT}}(I_R)$ but $r_1 \circ r_3$ is not a prefix of $S_{\text{OPT}}(I_R)$. Then, let $r_3'$ be the opposite maximal run to $r_3$, i.e., $r_1, r_3'$ are the first two runs of $S_{\text{OPT}}(I_R)$. We have $I_3$ is a subsequence of $I_1$. Hence, applying Lemma 8 to $r_3'$ on input $I_1$ and $r_4$ on input $I_3$, we have that at the end of $r_4$, the algorithm has written out all elements in $r_1 \circ r_3'$. Thus, $r_1 \circ r_3 \circ r_4$ covers $r_1 \circ r_3'$.

In the last partition, since $A$ outputs at most 3 runs, it can only achieve a ratio worse than $3/2$ if the optimal algorithm wrote out a single run. But then that run is longer, and $A$ would choose it. Therefore, we have $R(S) \leq (3/2) \cdot \text{OPT}$.

Proof of Theorem 20. In each partition, let the unwritten-element sequence be $I_r$ and the optimal proper output of $I_r$ be $S_{\text{OPT}}(I_r)$. The algorithm randomly picks the direction of the next run and writes a maximal runs in that direction using $M$-size buffer. It uses the extra $M$ buffer slots to simulate the buffer state of the maximal run in the other direction to check if the run it chose is at least as long as the other run. If the algorithm picked the run that is at least as long as the other run, it then writes a maximal run in the same direction followed by another maximal run in the opposite direction. The algorithm stops when there is no more element to write. In the proof of Theorem 19, we showed that these three runs will cover the first two runs of $S_{\text{OPT}}(I_r)$.

If the algorithm picked the shorter run, it then writes three more maximal runs with alternating directions. We know that the first two runs with alternating directions cover the first run of $S_{\text{OPT}}(I_r)$ as argued in the proof of Theorem 5, hence, the next two runs with alternating directions cover the second run of $S_{\text{OPT}}(I_r)$.

In the last partition, if $\text{OPT}(I_r) = 2$, the analysis is the same. If $\text{OPT}(I_r) = 1$, then the optimal output must be the longer maximal run. The algorithm, if picked the shorter run, then will cover the longer run when it writes the next maximal run in the opposite direction as showed in the proof of Theorem 9. Therefore, we have $\mathbb{E}[x]/y \leq 1/2(4/2) + 1/2(3/2) = 7/4$.

Applying Theorem 7 with $\alpha = 1.75$ and $\beta = 2$, we have: $\mathbb{E}[R(S)] \leq (7/4)\text{OPT}$ and $R(S) = 2\text{OPT}$.

Proof of Theorem 21. Suppose an algorithm has $(4M - 3)$-size buffer. Consider the input $I_1 \circ e \circ I_2$ where $I_1 = (1 \Rightarrow M - 1) \circ (2M - 1 \leftarrow M) \circ (3M \Rightarrow 4M - 2) \circ (-M \leftarrow -2M + 2)$. 

21
If \( S \) first writes \(-2M + 2 \), then let \( e = -2M + 1 \).

- Case 1: if \( S \) writes \( e = -2M + 1 \) next, then let \( I_2 = (0 \searrow -(M-1)) \). Thus, \( S \) has to spend at least two runs while an optimal output is one run: \((4M - 2 \searrow 3M) \circ (2M - 1 \swarrow -2M + 1)\).
- Case 2: if \( S \) writes \(-2M + 3 \) next, let \( I_2 = (-2M \searrow -10M) \circ (2M \nearrow 3M - 1) \). Then \( S \) has to spend at least 3 runs while an optimal output has 2 runs: \((4M - 2 \searrow 3M) \circ (2M - 2 \swarrow -10M) \circ (2M \nearrow 3M - 1) \).

Similarly, if \( S \) first writes \( 4M - 2 \), then let \( e = 4M - 1 \).

- Case 1: if \( S \) writes \( e = 4M - 1 \) next, then let \( I_2 = (2M \nearrow 3M - 1) \).
- Case 2: if \( S \) writes \( 4M - 3 \) next, let \( I_2 = (4M \nearrow 10M) \circ (0 \searrow -M + 1) \).

If \( S \) first writes \( e' \notin \{-2M + 2, 4M - 2\} \), then let \( e = -2M + 1, I_2 = (0 \searrow -(M-1)) \). Thus, \( S \) has to spend at least two runs while an optimal output has the following output with one run: \((4M - 2 \searrow 3M) \circ (2M - 1 \searrow -2M + 1) \).

**Proof of Theorem 22.** We apply Theorem 7 with \( x = ([1/\varepsilon] + 1) \) and \( y = [1/\varepsilon] \). In any partition except the last one, the algorithm chooses the combination of \([1/\varepsilon]\) maximal runs \( r_1 \circ \cdots \circ r_{[1/\varepsilon]} \) whose output is longest (ties are broken arbitrarily) and writes out one extra run \( r_{([1/\varepsilon]+1)} \). By Lemma 6, \( r_1 \circ \cdots \circ r_{[1/\varepsilon]+1} \) covers the first \([1/\varepsilon]\) runs of a proper optimal output of the unwritten-element sequence \( I_r \) in \((1 + [1/\varepsilon])\) runs. In the last partition, the algorithm chooses a combination of runs with the smallest number of runs.

Therefore, we obtain an \( \beta = 1 + 1/[1/\varepsilon] \leq 1 + 1/\varepsilon \) approximation.

There are \( 2([1/\varepsilon]+1) \) combinations to consider (each run can be up or down). The length of a run can be calculated in \( O(N_i) \) time by simulating it directly. \( N_i \) is the length of the longest output, namely, \( |r_1 \circ \cdots \circ r_{[1/\varepsilon]+1}| \). Since \( N_i \) items are then written out, the total running time is \( O(\sum_{t=1}^{t} N_i 2^{[1/\varepsilon]}) = O(N2^{1/\varepsilon}) \). Searching for the shortest way to write out the remaining elements (once \( I_r = \emptyset \)) takes \( O(N2^{1/\varepsilon}) \) time, which does not affect the running time.

**Proof of Theorem 23.** In each partition, we restrict the search for the combination of \((1/[\varepsilon])\) consecutive runs that writes the longest sequence as described above. By Lemma 16, if \( d \) runs remain to be written out, we must examine one subcase with \( d - 1 \) runs remaining, and one with \( d - 2 \) runs remaining. Thus, the number of combinations we need to consider is \( F_d = F_{d-1} + F_{d-2} \). Therefore, the running time of this step is \( O(F_{[1/\varepsilon]} N_i \log N_i) \). Thus, we have

\[
O(F_{[1/\varepsilon]} N_i \log N_i) = O(\varphi^{[1/\varepsilon]} N_i \log N_i).
\]

This is because \( F_{[1/\varepsilon]} = (\varphi^{[1/\varepsilon]} - \varphi_{[1/\varepsilon]}/\sqrt{5} \leq \varphi^{[1/\varepsilon]} \).

**Proof of Theorem 24.** At each decision time step, \( A \) flips a coin to pick a direction for the next run \( r \). It begins writing an up or down run according to the coin flip.

Meanwhile, \( A \) uses \( M \) additional space to simulate \( r' \), the run in the opposite direction. In particular, it simulates the contents of the buffer at each time step, as well as the last element written. Note that \( A \) does not need to keep track of the most recent element read when simulating \( r' \), as it is always the last element in the buffer.

By Lemma 14, the run with the incorrect direction has length less than \( 3M \) and the run with the correct direction has length of \( 3M \) or more. Thus, \( A \) can tell if it picked the correct direction. With probability \( 1/2 \), \( A \) writes the longer run. Therefore, it knows it made the correct direction and repeats, flipping another coin.

Now consider the case where \( A \) picks the wrong direction. When \( r \) ends (at time \( t \)), \( r' \) is continuing. Then \( A \) must act *exactly* as if it had written \( r' \). Specifically, we cannot simply cover \( r' \) and use an argument akin to Theorem 7 as then the unwritten-element sequence may not be 3-nearly-sorted.

To simulate \( r' \), \( A \) has two tasks: (a) \( A \) must write all elements that were written by \( r' \) that were not written by \( r \), and (b) \( A \) must “undo” writing any element that was written during \( r \) that is not in \( r' \), in case
these elements are required to make a subsequent run have length $3M$. Divide the buffer into two halves: $B_r$ is the buffer after writing $r$, and $B_{r'}$ is the buffer being simulated when $r$ ends.

The first task is to ensure that $A$ writes all elements written by $r'$ that were not written by $r$ in the direction of $r'$. These elements must be in $B_r$ since they were not written by $r$; and they must not be in $B_{r'}$ since they were written by $r'$. Thus, $A$ can simply write out each element in $B_r$ that is not in $B_{r'}$ and continue writing $r'$ from that time step.

The second task is to ensure that all elements written during $r$ that were not written during $r'$ cannot affect future run lengths. These elements must be in $B_{r'}$ but not in $B_r$. We mark these elements as special ghost elements. We can do this with $O(1)$ additional space by moving them to the front of the buffer and keeping track of how many of them there are. During subsequent runs, these are considered to be a part of $A$’s buffer. However, when $A$ would normally want to write one of these elements out, it instead simply deletes it from its buffer without writing any element. That said, $A$ still counts these deletions towards the size of the run. Note that our buffer never overflows, as $A$ continues to write (or delete) one element per time step.

When this simulation is finished, the contents of $A$’s buffer are exactly what they would have been had it written out $r'$ in the first place—however some are ghost elements, and will be deleted instead of written. Then $A$ repeats, flipping another coin.

For each run in the optimal output, either: $A$ writes that run exactly for cost 1 (with probability 1/2), or $A$ writes another run, and makes up for its mistake by simulating the correct run exactly, for cost 2 (with probability 1/2). Thus $A$ has expected cost $(3/2)\text{OPT}$. In the worst case, $A$ guesses incorrectly each time for a total cost of $2\text{OPT}$.

**Proof of Lemma** $25$. Suppose we are at a decision time step. Without loss of generality, assume this time step to be 0. Let the next two possible maximal runs be $r_1$ and $r_2$ that are up and down respectively. Without loss of generality, suppose $|r_1| \geq 5M$. Let $r_3$ be the maximal decreasing run that follows $r_2$. By Lemma $16$, either writing $r_1$ or writing $r_2 \circ r_3$ is optimal on the unwritten-element sequence. Call the two outputs $S_1$ and $S_2$ respectively.

Let $s_1^B, s_2^N, t_1^B, f_1, u_1, s_2^B, s_2^N, t_2^B, f_2, u_2$ be the same sets described in the proof of Lemma $14$. Let $r_1 = (x_1, \ldots, x_k), r_2 = (y_1, \ldots, y_k), r_3 = (y_{k+1}, \ldots, y_q)$. Let the buffers of $S_1$ and $S_2$ after time step $t + \ell$ be $B_{1,\ell}, B_{2,\ell}$. Let $\ell \geq \ell$ be the smallest such that $x_{j+1} \geq y_{j+1}$.

Similar to the proof of Lemma $14$ we let $r_{1,2} = \{x_{j+1}, \ldots, x_j\}$ and $t_{1,2} = \{x_{j+1}, \ldots, x_k\}$. Let $s_{1,2}^N$ be the set of elements in $r_{1,2}$ but not in $B_{1,\ell}$. Let $s_{1,2}^B$ be the set of elements in $r_{1,2}$ and also in $B_{1,\ell} \setminus r_{1,2}$. Let $f_{1,2}$ be the set of elements in $t_{1,2}$ and also in $B_{1,\ell} \setminus r_{1,2}$. Let $u_{1,2}$ be the set of elements not in $r_1$ and read in before $x_{j+1}$ is written. Let $f_{1,2}$ be the set of elements in $r_1$ and read in after $x_{j+1}$ is written.

We define $s_3 = \{y_{k+1}, \ldots, y_q\}$ and $t_3 = \{y_{k+1}, \ldots, y_q\}$. Let $s_3^B$ be the set of elements in $s_3$ but not in $B_{2,\ell}$. Let $f_3$ be the set of elements in $s_3 \cap B_{2,\ell}$. Let $u_3$ be the set of elements that are not in $r_3$ and read in before $y_{j+1}$ is written. Let $f_3$ be the set of elements in $r_3$ and read in after $y_{j+1}$ is written.

Since the buffer of $S_2$ must keep all elements in $s_{1,2}^N, s_{1,2}^B$ at time step $j + 1$, we have $|s_{1,2}^N| + |s_{1,2}^B| \leq M$. We have,

$$|r_1| = |r_2| + (|u_3| + |s_3^N|) + (|s_{1,2}^N| + |s_{1,2}^B|) + |f_{1,2}|$$

$$\leq (2M + |u_1| + |f_2|) + (M - |u_1| - |f_2|) + M + |f_{1,2}|$$

$$= 4M + |f_{1,2}|.$$

Since $|f_{1,2}| \geq M$ because of our assumption $|r_1| \geq 5M$, $r_3$ has to end before $r_1$ using the same argument as in the proof of Lemma $14$. Since $|r_1| \geq |r_2 \circ r_3|$ follows by any maximal run will cover $r_2 \circ r_3$. Therefore, $r_1$ is an optimal prefix of the unwritten-element sequence. At every time step, the maximal run of length $5M$ or more is always a prefix of an optimal output on the unwritten-element sequence as required.

\[\Box\]