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

Michael A. Bender[1], Samuel McCauley[1], Andrew McGregor[2], Shikha Singh[1], and Hoa T. Vu[2]

[1] Stony Brook University, Stony Brook, NY 11794-2424 USA.
{bender,smccauley,shiksingh}@cs.stonybrook.edu.
[2] University of Massachusetts, Amherst, MA 01003 USA.
{mcgregor,hvu}@cs.umass.edu.

Abstract. 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.

First, 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.

Next, we give online algorithms having smaller competitive ratios with resource augmentation. We demonstrate that performance can also be improved with a small amount of foresight. Lastly, we present algorithms tailored for "nearly sorted" inputs which are guaranteed to have sufficiently long optimal runs.

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*** [7, 11].

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 [5–7, 10, 12, 13, 15].

The most well-known external-memory sorting algorithm is multi-way merge sort [1, 9]. The multi-way merge sort is formalized in the ***disk-access machine***[3] (***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 $O\big((N/B)\log_{M/B}(N/B)\big)$ I/Os, where N is the number of elements to be sorted. This is the best possible [1].

[3] The external-memory (or I/O) model applies to any two levels of the memory hierarchy.

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. Naïvely, 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.

Replacement Selection. The classic algorithm for run generation is called ***replacement selection*** [8,11]. Starting from an initially full internal-memory (or buffer), replacement selection proceeds as follows:

1. Pick the smallest element[4] at least as large as each element in the current run.
2. If no such element exists, then the run ends.
3. ***Eject*** that element, and ***ingest*** the next, 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. The algorithm can then ingest from the input block and eject to the output block one element at a time, while the blocks can be filled/emptied in chunks of size B.

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 $2M$ [7]. In [11], 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 and outputs runs much longer than M. For example, when each element in the input appears at a distance at most M from its actual rank, a single run is generated.

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 where, each time a new run begins, the replacement-selection algorithm has a choice about whether to generate an ***up run*** or a ***down run***.

[4] Data structures such as heaps can identify the smallest elements in memory. But from the perspective of minimizing I/Os, this does not matter—computation is free in the DAM model.

Knuth [10] 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. [15] 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 [10]) and (2) produces significantly longer runs on inputs that have mixed up-down runs and reverse-sorted inputs.

Our Contributions. The results in this paper complement these prior works. In contrast to Knuth's negative result for random inputs [10], we show that strict up-down alternation is the best possible for worst-case inputs. Moreover, we give better competitive ratios with resource augmentation, which helps explain why heuristically choosing between up and down runs based on the elements currently in memory may yield better solutions.

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 in the output.[5] We establish the following:

1. *Analysis of alternating-up-down replacement selection.* We prove that alternating-up-down replacement selection is 2-competitive. Furthermore, we show that this is the best possible for deterministic online algorithms.
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 design a deterministic online algorithm that, given a buffer of size $4M$, matches or beats any optimal algorithm with 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 $3M$ 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. These results are reminiscent of previous literature studying sorting on inputs with "bounded disorder" [3] and adaptive sorting algorithms [4, 14, 16].
5. *PTAS for the offline problem.* We give a polynomial-time approximation scheme for the offline run-generation which guarantees a $(1+\varepsilon)$-approximation with a running time of $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^{1/\varepsilon} N \log N\right)$.

[5] Note that for a given input, minimizing the number of runs is equivalent to maximizing the average length of runs.
Due to space constraints, we defer some proofs to the full-version [2].

2 Up-Down Run Generation

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

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 those in Section 5 and Section 7, do require uniqueness.

An algorithm A ***reads*** an element of I when A transfers the element from the input sequence to the buffer. 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 that partition its output. Specifically, the number of runs in an output S, denoted $R(S)$, is the smallest number of mutually disjoint sequences $S_1, S_2, \dots, S_{R(S)}$ such that each S_i is a run and $S = S_1 \circ \dots \circ S_{R(S)}$ 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 call an online algorithm ***β-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*** consists of the contents of the buffer, concatenated with the remaining (not yet ingested) input elements. For the sake 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.

Notation. We use $(x \nearrow y)$ to denote the increasing sequence $x, x+1, x+2, \dots, y$ and $(x \searrow y)$ to denote the decreasing sequence $x, x-1, x-2, \dots, y$.

Let $A = a_1, a_2, \dots, a_k$. We use $A \oplus x$ to denote the sequence $a_1+x, a_2+x, \dots a_k+x$. Similarly, we use $A \otimes x$ to denote the sequence $a_1x, a_2x, \dots, a_kx$.

Let A and B be sequences. We say that A ***covers*** B if for all e, $e \in B \Rightarrow e \in A$. 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 about run generation and present the tools used to analyze our algorithms. We show that it is never a good idea to end a run early or to "skip over" an element (keeping it in the buffer even when it could have been added to the the current run).

First, 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 $R(S') \leq R(S)$.

Lemma 1. *If I' is a subsequence of I, then* $\mathrm{OPT}(I') \leq \mathrm{OPT}(I)$.

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 the 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.

Lemma 3. *For any input I, there exists a proper output S such that $R(S) = \mathrm{OPT}(I)$.*

We use the following property of proper algorithms throughout the paper.

Property 1. Any proper algorithm satisfies the following two properties:
1. At each decision point, the elements of the buffer must have arrived while the previous run was being written.
2. A new element cannot be included in the current run if the last element output is larger (smaller) and the current run is increasing (decreasing).

The following observations and lemmas are used in the analysis of our algorithms.

Observation 1. *Consider algorithms A_1 and A_2 on input I. Suppose that at time step t_1 algorithm A_1 has written out all the elements that algorithm A_2 already wrote out by some previous time step t_2. Then, the unwritten-element sequence of algorithm A_1 at time step t_1 forms a subsequence of the unwritten-element sequence of algorithm A_2 at time step t_2.*

Lemma 4. *Consider a proper algorithm A. At some decision time step, A can write k runs $p_1 \circ \cdots \circ p_k$ or ℓ runs $q_1 \circ \cdots \circ q_\ell$ such that $|p_1 \circ \cdots \circ p_k| \geq |q_1 \circ \cdots \circ q_\ell|$.*

Then $p_1 \circ \cdots \circ p_k \circ p_{k+1}$, where p_{k+1} is either an up or down run, covers $q_1 \circ \cdots \circ q_\ell$; the unwritten-element sequence after A writes $p_1 \circ \cdots \circ p_{k+1}$ is a subsequence of the unwritten-element sequence after A writes $q_1 \circ \cdots \circ q_\ell$.

Proof. Since $|p_1 \circ \cdots \circ p_k| \geq |q_1 \circ \cdots \circ q_\ell|$, the set of elements that are in $q_1 \circ \cdots \circ q_\ell$ but not in $p_1 \circ \cdots \circ p_k$ have to be in the buffer when p_k ends. By Property 1, p_{k+1} will write all such elements.

The next theorem serves as a template for analyzing our algorithms. It lets us compare the output of our algorithm against that of the optimal in small *partitions*. We show that if in every partition i, an algorithm writes x_i runs that cover the first y_i runs of an optimal output (on the current unwritten-element sequence), and $x_i/y_i \leq \beta$, then the algorithm outputs no more than $\beta \cdot \text{OPT}$ runs.

Theorem 1. *Let A 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 $1 < i \leq k$, let I_i be the unwritten-element sequence after A writes S_{i-1}. Let $I_1 = I$, $I_{k+1} = \emptyset$ and $\alpha, \beta \geq 1$. 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 $R(S) \leq \beta \cdot \text{OPT}$. Similarly, if for all $i \leq k$, S_i covers the first y_i runs of S'_i, and $\mathbb{E}[x_i]/y_i \leq \alpha$, then $\mathbb{E}[R(S)] \leq \alpha \cdot \text{OPT}$.

Proof. Consider I'_i, the unwritten element sequence at the end of the first y runs of S'_{i-1} (we let $I'_1 = I$). We show that $\text{OPT}(I_i) \leq \text{OPT} - \sum_{j=1}^{i-1} y_i$ for all $1 \leq i \leq k$ using induction. Note that $\text{OPT}(I_1) = \text{OPT}$. Assume $\text{OPT}(I_i) \leq \text{OPT} - \sum_{j=1}^{i-1} y_i$. Since S_{i+1} covers the first y runs of S'_{i+1}, by Observation 1, I_{i+1} is a subsequence of I'_{i+1}. Then by Lemma 1, $\text{OPT}(I_{i+1}) \leq \text{OPT}(I'_{i+1})$.

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

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

We also have the same in expectation, that is,

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

4 Up-Down Replacement Selection

We analyze *alternating-up-down replacement selection*, which deterministically alternates between writing (maximal) up and down runs. Knuth [10] showed that when the input elements arrive in a random order, alternating-up-down replacement selection performs worse than replacement selection (all up runs). We show that for deterministic online algorithms, alternating-up-down replacement selection is 2-competitive and optimal for *any* (adversarial) input.

Lemma 5. *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 S_1 and S_2 have initial runs r_1 and r_2 respectively and 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'.*

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

Proof. We show that we can apply Theorem 1 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 5 (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.

Theorem 3. *Let A be any online deterministic algorithm with output S_I on input I. Then there are arbitrarily long inputs I such that $R(S_I) \geq 2\mathrm{OPT}(I)$.*

Furthermore, we show that no randomized algorithm can achieve a competitive ratio better than $3/2$.

Theorem 4. *Let A be any online, randomized algorithm. Then there are arbitrarily long inputs such that $\mathbb{E}[R(S_I)] \geq (3/2)\mathrm{OPT}(I)$.*

5 Run Generation with Resource Augmentation

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

- ***Extra buffer***: the algorithm's buffer is a constant factor larger (than the optimal).
- ***Extra visibility***: the algorithm has prescience—it can *see* a small number of incoming elements, but must read and write using the usual M-size buffer.

In this section, we assume that the input elements are unique, as duplicates nullify the power provided by augmentation. For example, c-visibility does not help if an input element is repeated c times.

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 next run.

We next look at some guarantees on the length of a run chosen by greedy (the ***greedy run***) and also on the run not chosen by greedy (the ***non-greedy run***).

Greedy is Good but not Great. First, we show that greedy is not optimal.

Lemma 6. *The greedy algorithm can be a factor of* $3/2$ *away from optimal.*

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 5. *Each greedy run except the last two has length* $\geq M + \lceil \lfloor M/2 \rfloor /2 \rceil$.

We 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 in the following lemma.

Lemma 7. *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* $|r_2| < 3M$*. This bound is tight; there is an input with* $|r_1| = 3M$ *and* $|r_2| = 3M - 1$.

Online Algorithms with Resource Augmentation. We 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.

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.

Theorem 6. *Let* A *be the algorithm with a* $4M$*-size buffer described above. On any input,* A *writes no more runs than an optimal algorithm with* M*-size buffer.*

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 pick. It is able to do so using its $4M$-size buffer because, by Lemma 7, 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 the choices in one run, by Theorem 1, it is able to match or beat OPT. $\square$

A natural question is whether resource augmentation boosts performance automatically, without using the greedy-run-simulation technique. The following lemma shows that this is not the case.

Lemma 8. *There exist inputs on which alternating up-down replacement selection with $4M$-size buffer does no better than it would with M-size buffer, that is, it produces twice the optimal number of 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.

The algorithm is only allowed to use the usual M-size buffer for reading and writing. Furthermore, the algorithm must read elements sequentially from input I, even if it sees future elements 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 6, we determine the direction leading to the longer (greedy) run using the $3M$-lookahead. However, unlike Theorem 6, an M-size buffer is too small to output this run. Instead, we show that it is possible to cover two runs of the optimal algorithm by writing three maximal runs—a greedy run, followed by two additional runs in the same direction and the opposite direction respectively.

Theorem 7. *Let* OPT *be the optimal number of runs given an M-size buffer on an input I with no duplicates. Then there exists an online algorithm A with an M-size buffer and $4M$-visibility such that A always outputs S satisfying $R(S) \leq (3/2)\mathrm{OPT}$.*

$7/4$-competitive using $2M$-size buffer. A $2M$-size buffer is insufficient to determine the direction leading to the longer (greedy) run. Instead, suppose an algorithm picks a direction randomly, and writes a maximal run using a M-size buffer. It then uses the additional M buffer slots to simulate the opposite run.

With probability $1/2$, the algorithm picks the greedy direction and can cover the first two runs of optimal (on the unwritten-element sequence) with three runs (as in Theorem 7). With probability $1/2$, the algorithm picks the wrong direction. Consequently, writing four (alternating) runs cover two runs of the optimal. In expectation, this achieves a competitive ratio of $1/2(3/2) + 1/2(4/2) = 7/4$.

Theorem 8. *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 $\mathbb{E}[R(S)] \leq (7/4)\mathrm{OPT}$ and $R(S) \leq 2\mathrm{OPT}$.*

Lower Bound for Resource Augmentation. With less than $(4M-2)$-augmentation, no deterministic online algorithm can be $3/2$-competitive on all inputs. Thus, Theorem 6 and Theorem 7 are nearly tight.

Theorem 9. *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* $R(S) \geq (3/2)\mathrm{OPT}$.

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 polynomial-computation-time algorithm) the optimal strategy which, when executed by a run generation algorithm (with a buffer of size M), produces the minimum number of runs.

For any ε, a $(1 + \varepsilon)$-approximation can be achieved by a brute force search on partitions of the output containing small number of runs. We improve the running time of this simple PTAS by pruning the suboptimal paths in this search.

Theorem 10. *There exists an offline algorithm* A *with output* S *such that* $R(S) \leq (1 + \varepsilon)\mathrm{OPT}$*. The running time of* A *is* $O(\varphi^{1/\varepsilon} N \log N)$ *where* $\varphi = (1 + \sqrt{5})/2$.

7 Run Generation on Nearly Sorted Input

We show that up-down replacement selection performs better on inputs with inherent sortedness. In particular, we say that an input is ***c-nearly-sorted*** if there exists a proper optimal algorithm which outputs runs of length at least cM.

Theorem 11. *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.*

Theorem 12. *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 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 [10], until the experimental work of Martinez-Palau et al. [15].

The results in our paper complement the findings of Knuth [10] and Martinez-Palau et al. [15]. In particular, strict up-down alternation being the best possible strategy explains why heuristics for up-down run-generation lead to better performance. Moreover, our constant-factor competitive ratios with resource augmentation 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? 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 be 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 all require maintaining too many buffer states to run in polynomial time.

9 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, CCF 0953754, IIS 1251110, CCF 1320719, and by Google Research and Sandia National Laboratories.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Communications of the ACM 31(9), 1116–1127 (Sep 1988)
2. Bender, M.A., McCauley, S., McGregor, A., Singh, S., Vu, H.T.: Run generation revisited: What goes up may or may not come down. arXiv preprint arXiv:1504.06501 (2015)
3. Chandramouli, B., Goldstein, J.: Patience is a virtue: Revisiting merge and sort on modern processors. In: Proc. Int'l Conference on Management of Data. pp. 731–742 (2014)
4. Estivill-Castro, V., Wood, D.: A survey of adaptive sorting algorithms. ACM Computing Surveys 24(4), 441–476 (1992)
5. Frazer, W., Wong, C.: Sorting by natural selection. Communications of the ACM 15(10), 910–913 (1972)
6. Friend, E.H.: Sorting on electronic computer systems. Journal of the ACM 3(3), 134–168 (1956)
7. Gassner, B.J.: Sorting by replacement selecting. Communications of the ACM 10(2), 89–93 (1967)
8. Goetz, M.A.: Internal and tape sorting using the replacement-selection technique. Communications of the ACM 6(5), 201–206 (1963)
9. Graefe, G.: Implementing sorting in database systems. ACM Computing Surveys 38(3), 10 (2006)
10. Knuth, D.E.: Length of strings for a merge sort. Communications of the ACM 6(11), 685–688 (1963)
11. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3 (1998)
12. Lin, Y.C.: Perfectly overlapped generation of long runs for sorting large files. Journal of Parallel and Distributed Computing 19(2), 136–142 (1993)
13. Lin, Y.C., Lai, H.Y.: Perfectly overlapped generation of long runs on a transputer array for sorting. Microprocessors and Microsystems 20(9), 529–539 (1997)
14. Mallows, C.L.: Patience sorting. Bulletin of Inst. of Math. Appl. 5(4), 375–376 (1963)
15. Martinez-Palau, X., Dominguez-Sal, D., Larriba-Pey, J.L.: Two-way replacement selection. In: Proc. of the VLDB Endowment. vol. 3, pp. 871–881 (2010)
16. Wikipedia: Timsort (2004), http://en.wikipedia.org/wiki/Timsort