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

Michael A. Bender, Samuel McCauley, Andrew McGregor, Shikha Singh, Hoa T. Vu

Introduction

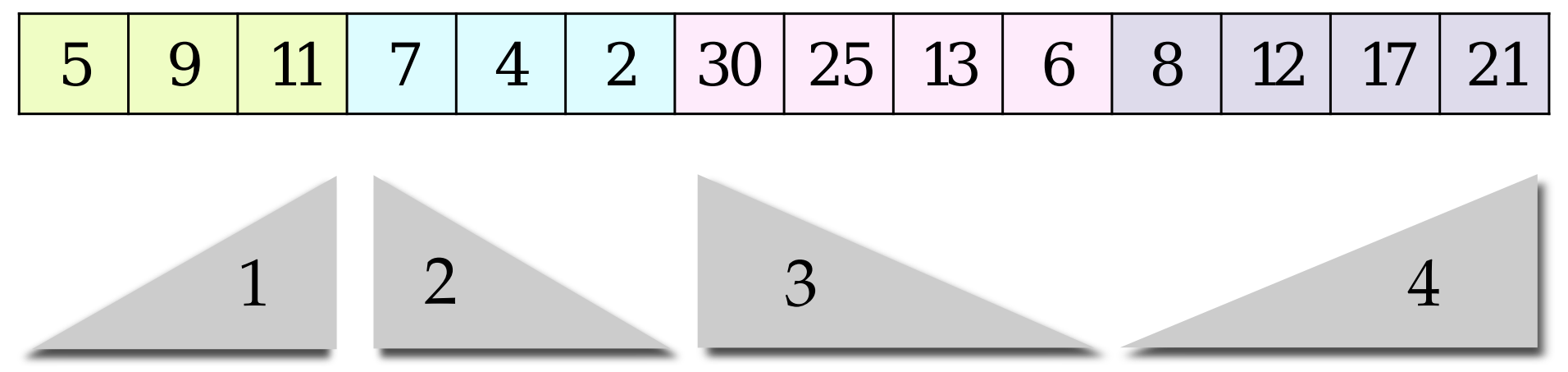
- Run generation is the first phase of external memory merge sort.
- The objective is to scan once through all the data and output runs (sorted chunks of elements) that are as long as possible.
- Longer runs lead to a faster merge phase.
- Generating runs before sorting is a common technique used, for example, in Python's Timsort.
- Classic, well-studied problem in the database community for over 50 years.
- Many heuristics have been proposed.
- We provide a theoretical foundation for run generation.
- We show that alternating between sorted and reverse sorted runs is asymptotically optimal online strategy, yielding at most twice the minimum number of runs.
- We improve performance ratios when the algorithm has extra resources or foreknowlege.

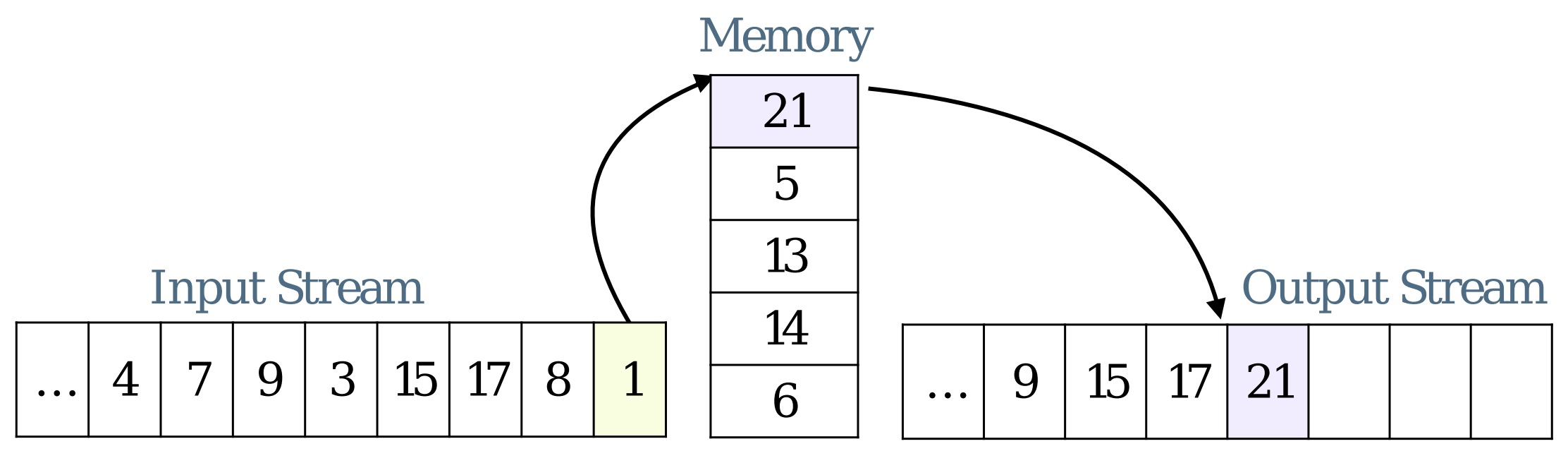
What is Run Generation?

Problem Definition

- Input stream arrives over time; can be stored temporarily in a buffer of size M
- Buffer gets full -> write an element to output stream, next element is read into the slot freed
- Buffer is always full (except when <M elements remain)
- Algorithm decides what to eject based on contents of buffer, last element written
- Algorithm can only read (in order) from input and append to output

Goal: Output the minimum number of runs





Up or Down? Up or Down?

Runs are contiguous sorted partitions of the output

- Up Runs: sorted in increasing order
- Down Runs: sorted in decreasing order

Maximal Runs

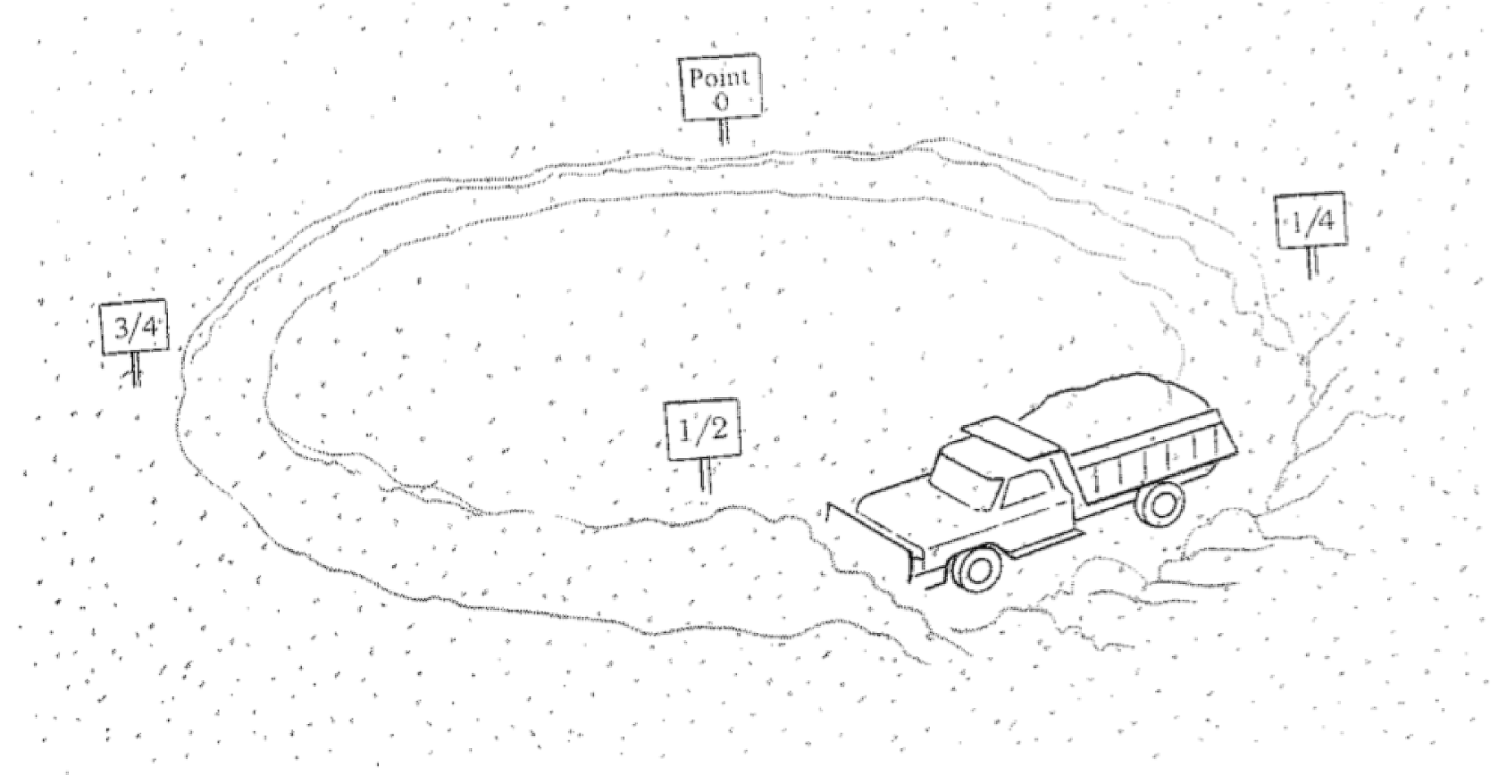
- Algorithm never skips over elements
- Algorithm never ends a run until forced to

Crux of Run Generation

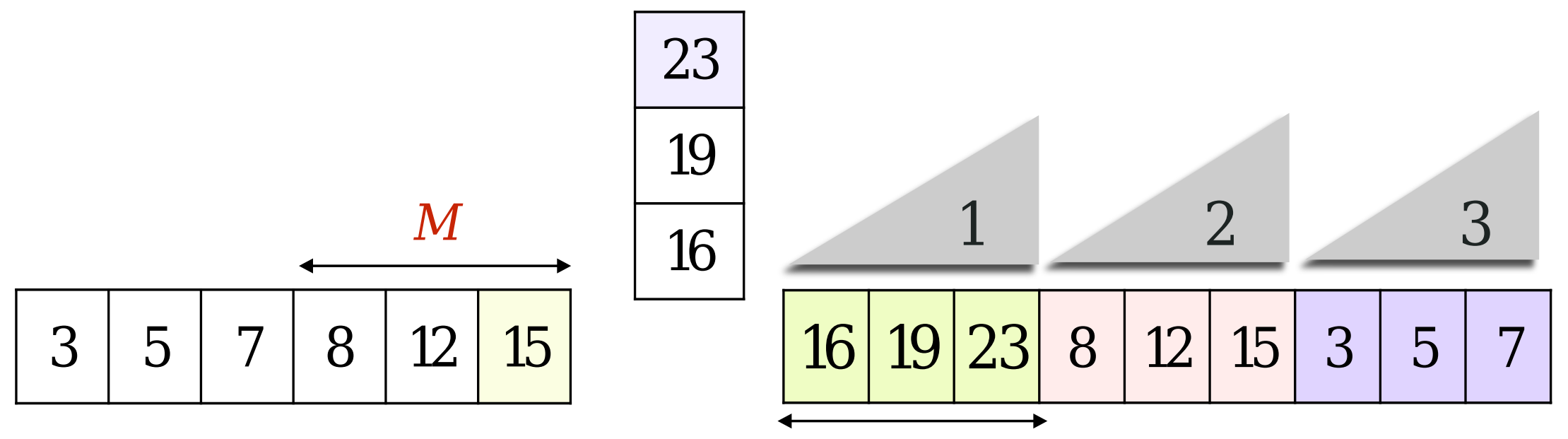
- Wlog, an algorithm must write maximal runs
- Only decision: Write an up run or a down run?

Prior Work

- Replacement Selection [Goetz 1963]: Classic algorithm; writes repeated maximal up runs
- Performance on random data: expected run length twice the size of memory; Knuth's proof by snow plow



However, Replacement selection does poorly on reverse sorted



Runs of length M on reverse sorted input

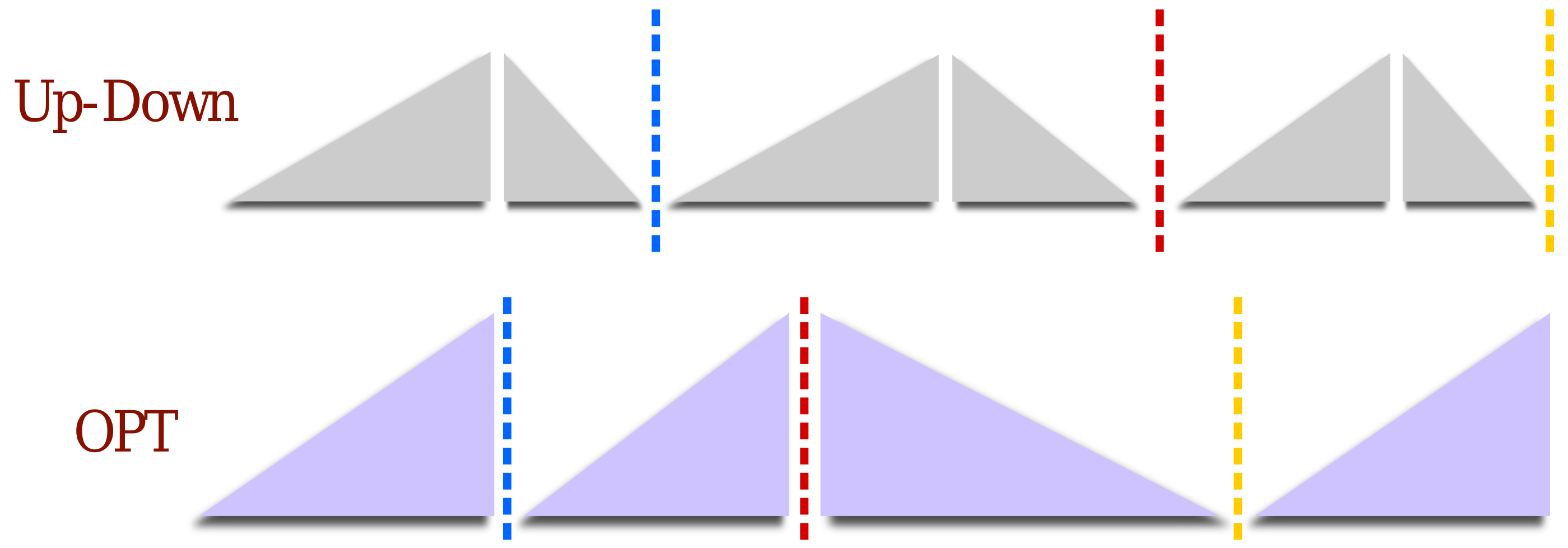
- Most recently, Martinez-Palau et al. [VLDB 2010]: Heuristically choose between starting an up or down run

Alternating Up-Down: Best Possible

- Deterministically alternating between up and down runs performes worse than Replacement Selection on random input
- Expected run length is 1.5M compared to 2M [Knuth 1963]

Our Result

- We show Alternating Up-Down is 2-competitive on any input
- Tight Lower Bound: No online deterministic algorithm can do better



Two runs of Up-Down cover at least one run of OPT

Random up-down?

- No randomized online algorithm can be better than 1.5-competitive

Summary of Results

Competitive Ratio	Buffer Size	Visibility	Online
2	M	M	Yes
1.5	M	4M	Yes
1.75	2M	2M	Yes
1	4M	4M	Yes
(1+ε)	M	N	No
1.5*	2M	2M	Yes
1*	M	N	No

** On "nearly sorted" input*

Contact Information

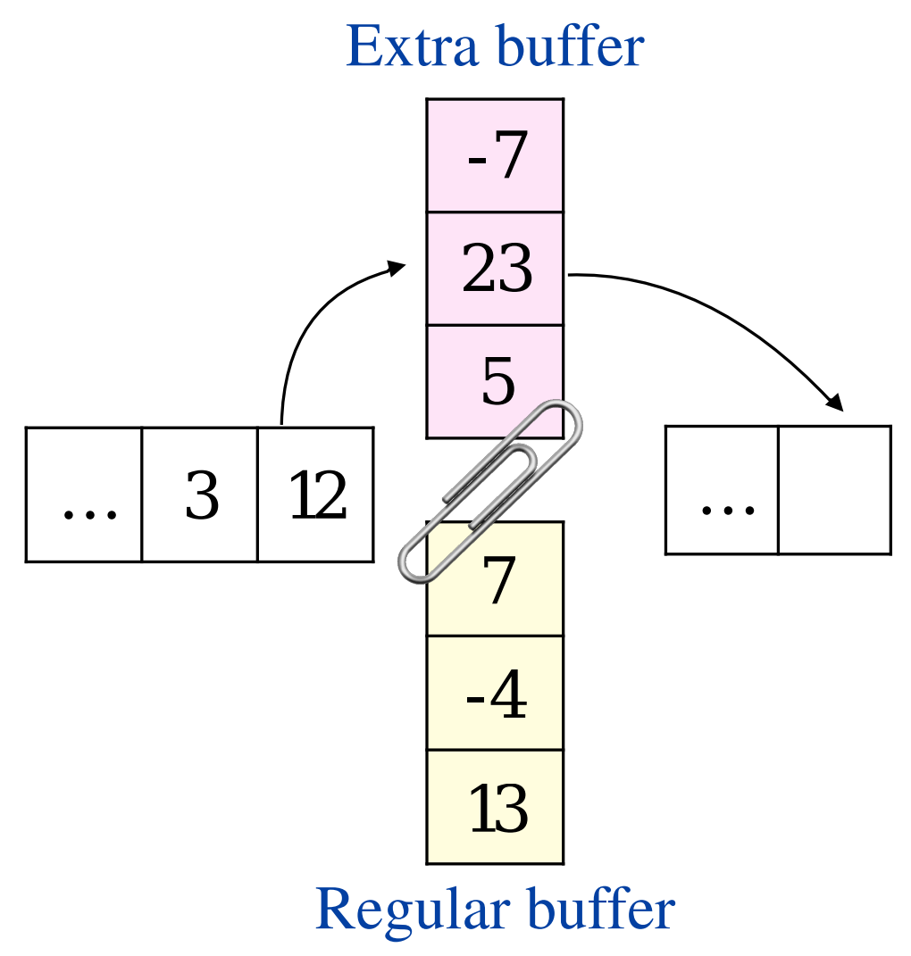
Samuel McCauley: smccauley@cs.stonybrook.edu
Shikha Singh: shiksingh@cs.stonybrook.edu
Hoa T. Vu: hvu@cs.umass.edu

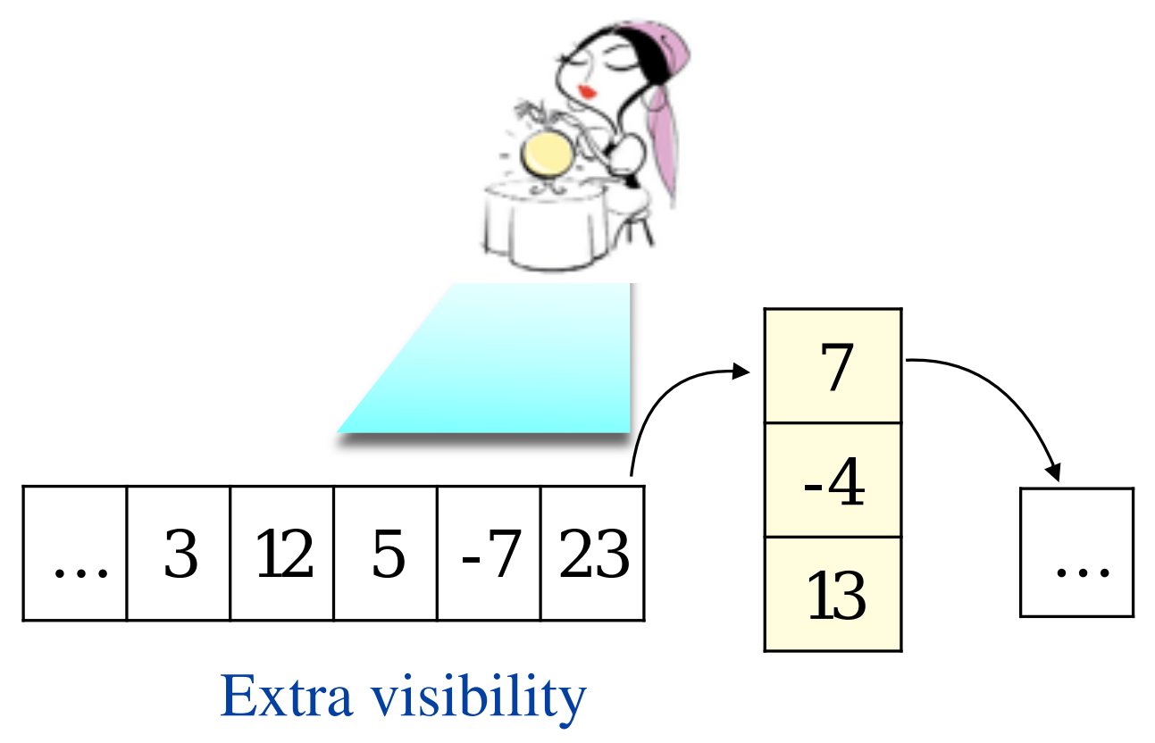
Stony Brook University
UMASS AMHERST

Resource Augmentation Results

Types of Resource Augmentation

- Extra Buffer: Algorithm can read into and write from the additional buffer
- Extra Visibility: Algorithm can only view a fixed number of future elements





Main Idea of Resource Augmentation

- Simulate Greedy: every time pick the direction that leads to a longer run

Greed is Good

- If the greedy run is at least 3M long, then non-greedy run is shorter than 3M

Our Results

- We give an algorithm that can match OPT when provided 4M-buffer
- We give an algorithm which is 1.5-competitive when provided 4M-visibility