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

Shikha Singh

Joint Work with :

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

Stony Brook University

UMASS AMHERST

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

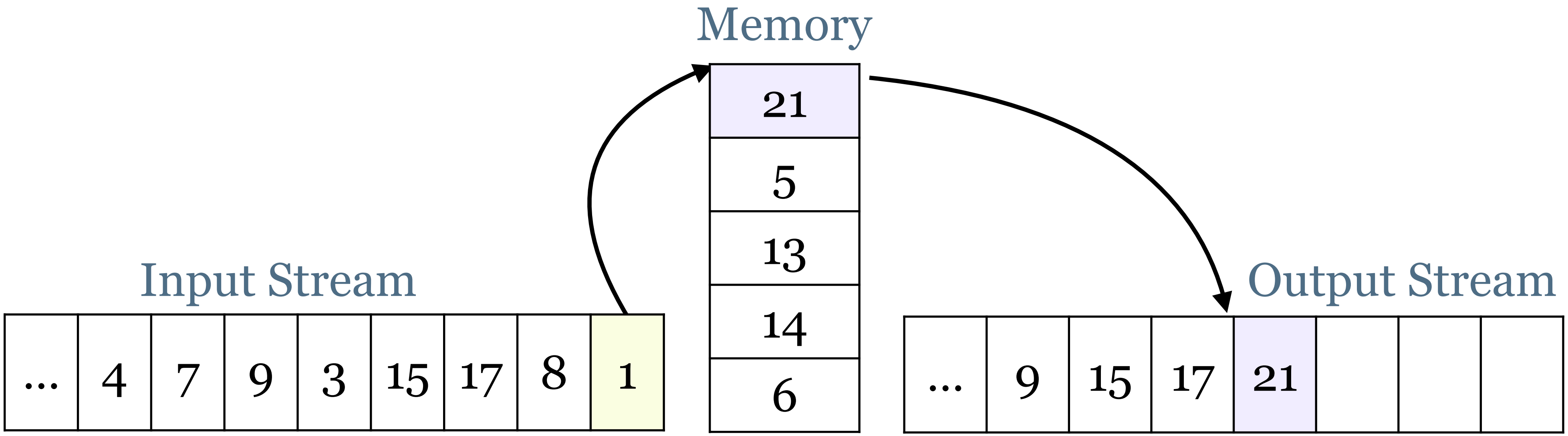
- Contiguous sequence of sorted elements in an array

5	9	11	2	4	7	6	13	25	30	3	5	7	11

- Number of runs:
    - Smallest number of runs that partition the array

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

- Run Generation is the first phase of external memory sorting



Memory

21
5
13
14
6

Input Stream

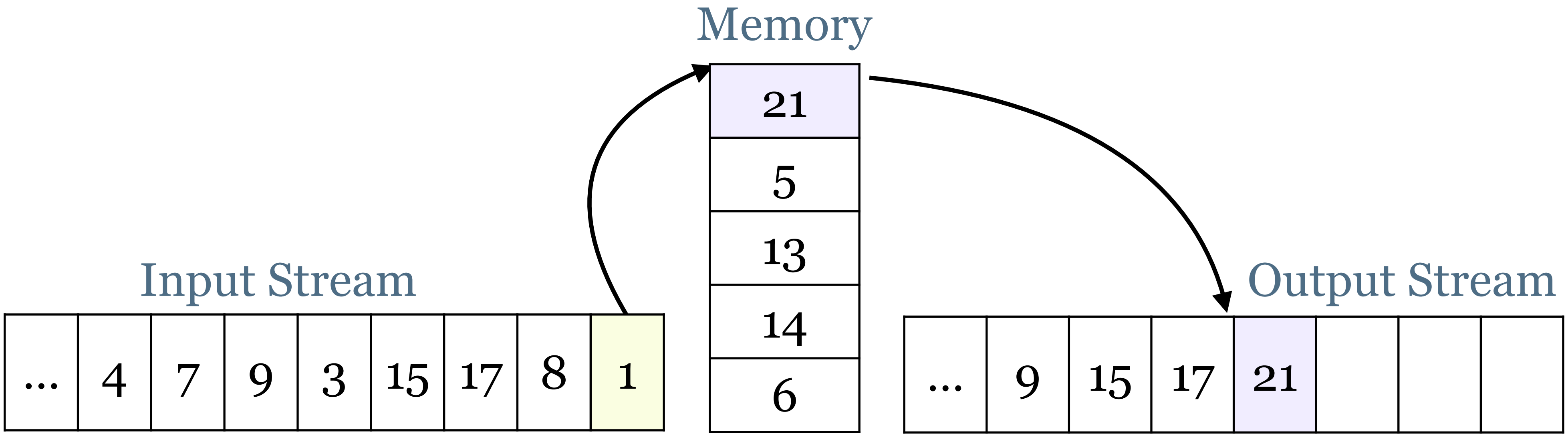
…	4	7	9	3	15	17	8	1

Output Stream

…	9	15	17	21			

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

- Scan input ingesting elements in memory
- Write out sorted runs to disk



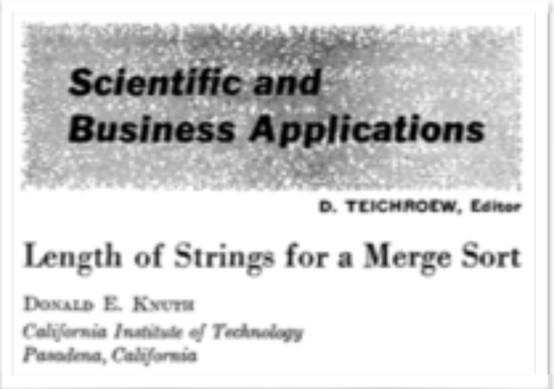
Objective: Minimize the number of runs or (equivalently)
Maximize average run length

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

Internal and Tape Sorting Using the Replacement-Selection Technique*

Martin A. Goetz
Applied Data Research, Inc., Princeton, N. J.

1963



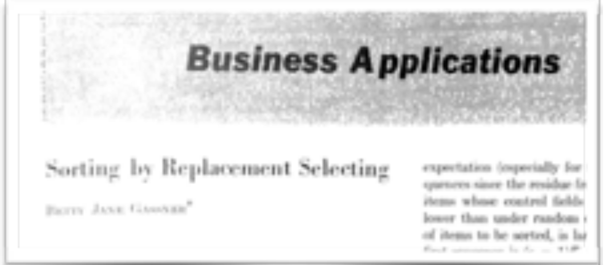
Scientific and Business Applications

D. TEICHROEW, Editor

Length of Strings for a Merge Sort

Donald E. Knuth
California Institute of Technology
Pasadena, California

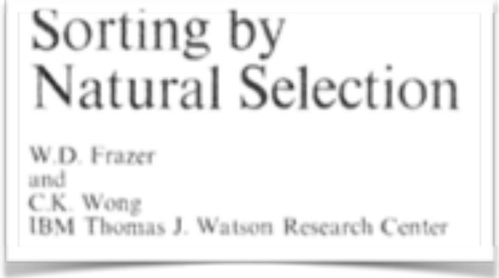
1963



Business Applications

Sorting by Replacement Selecting

1967



Sorting by Natural Selection

W.D. Frazer
and
C.K. Wong
IBM Thomas J. Watson Research Center

1972

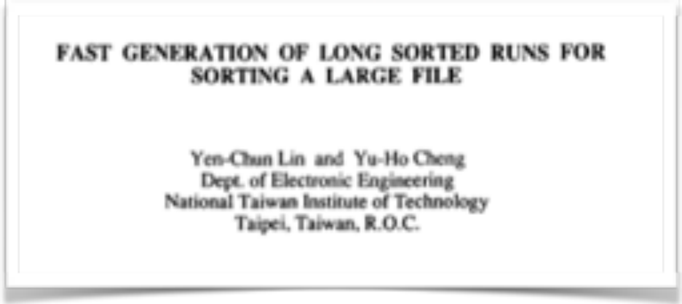
Perfectly Overlapped Generation of Long Runs for Sorting Large Files*

Yen-Chun Lin

1973

" If you remember the sixties, you weren't really there."

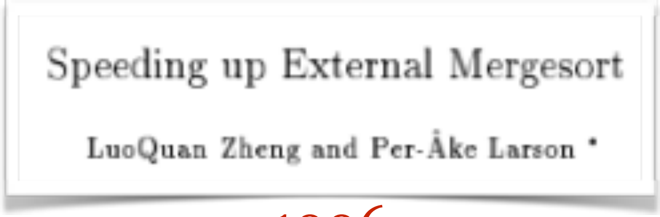
Run Generation Revisited: What Goes Up May or May Not Come Down



FAST GENERATION OF LONG SORTED RUNS FOR SORTING A LARGE FILE

Yen-Chun Lin and Yu-Ho Cheng
Dept. of Electronic Engineering
National Taiwan Institute of Technology
Taipei, Taiwan, R.O.C.

1991



Speeding up External Mergesort

LuoQuan Zheng and Per-Åke Larson *

1996

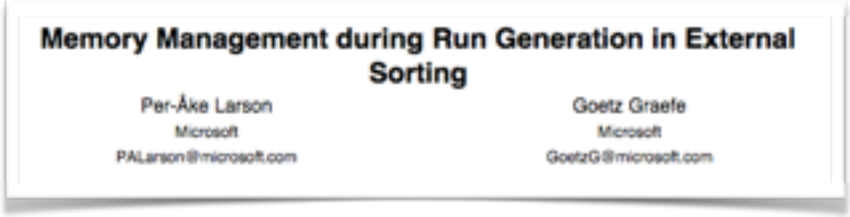
Perfectly overlapped generation of long runs on a transputer array for sorting

Yen-Chun Lin*, Horng-Yi Lai

Department of Electronic Engineering, National Taiwan Institute of Technology, P.O. Box 90-100, Taipei 106, Taiwan

Received 18 March 1996; revised 20 November 1996; accepted 9 December 1996

1997



Memory Management during Run Generation in External Sorting

Per-Åke Larson
Microsoft
PALarson@microsoft.com

Goetz Graefe
Microsoft
GoetzG@microsoft.com

1998

- Continued experimental studies to improve run length

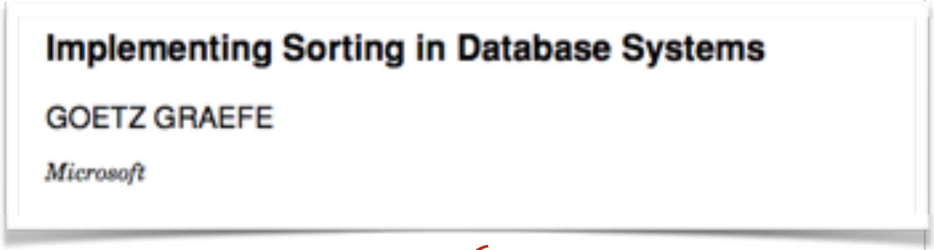
Run Generation Revisited:
What Goes Up May or May Not Come Down

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 15, NO. 4, JULY/AUGUST 2003

External Sorting: Run Formation Revisited

Per-Åke Larson, *Member, IEEE Computer Society*

2003



Implementing Sorting in Database Systems

GOETZ GRAEFE

Microsoft

2006

Two-way Replacement Selection

Xavier Martinez-Palau, David Dominguez-Sal, Josep Lluis Larriba-Pey
DAMA-UPC, Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Campus Nord-UPC, 08034 Barcelona
{xmartine,ddomings,larri}@ac.upc.edu

2010

External Sorting on Flash Memory
Via Natural Page Run Generation

YANG LIU, ZHEN HE, YI-PING PHOEBE CHEN AND THI NGUYEN

Department of Computer Science and Computer Engineering, La Trobe University, VIC 3086, Australia

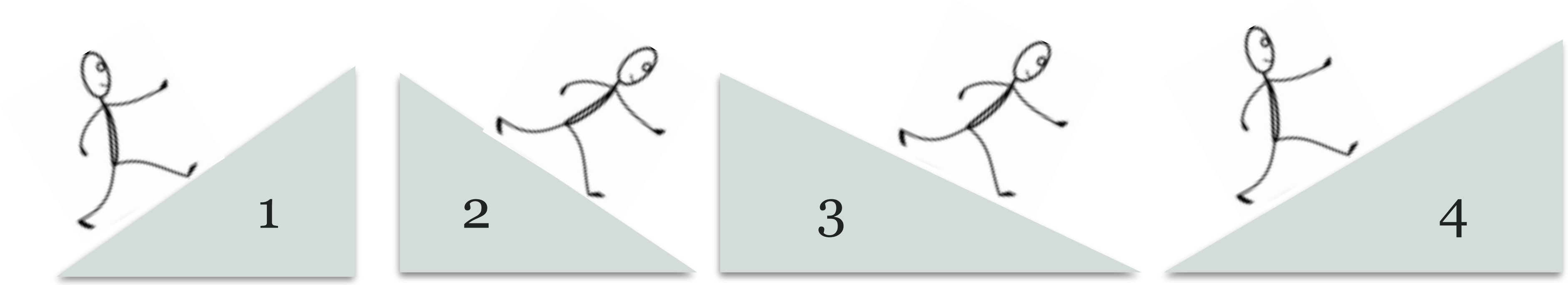
2011

- Classic Problem: Studied for over 60 years!

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

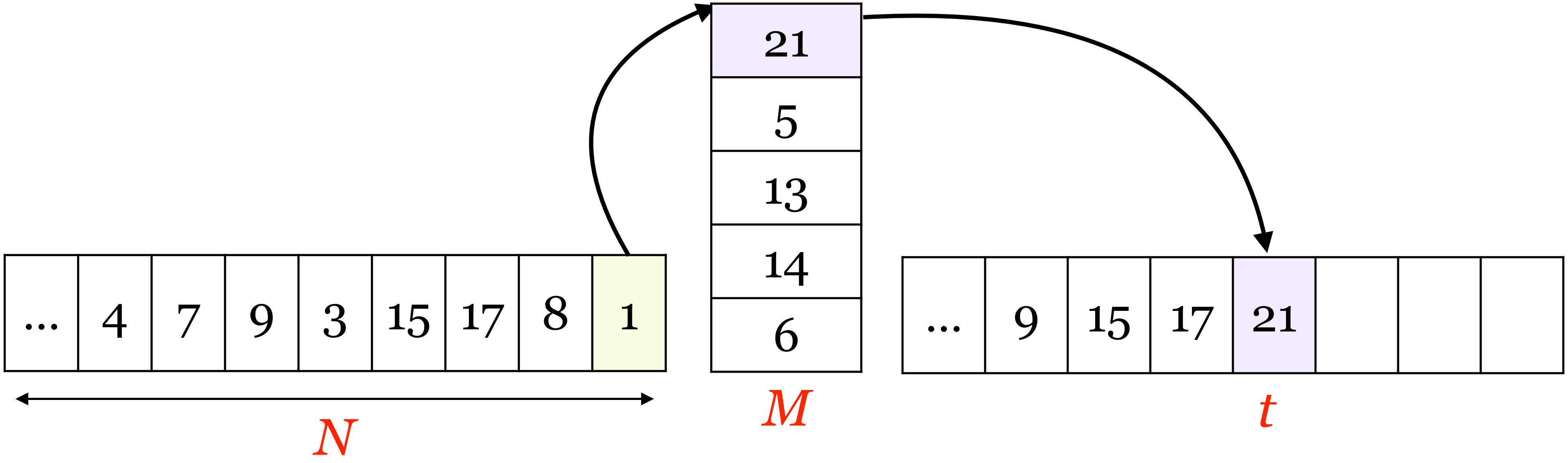
- Up Runs are monotonically increasing (sorted)
- Down Runs are monotonically decreasing (reverse sorted)

5	9	11	7	4	2	30	25	13	6	8	12	17	21



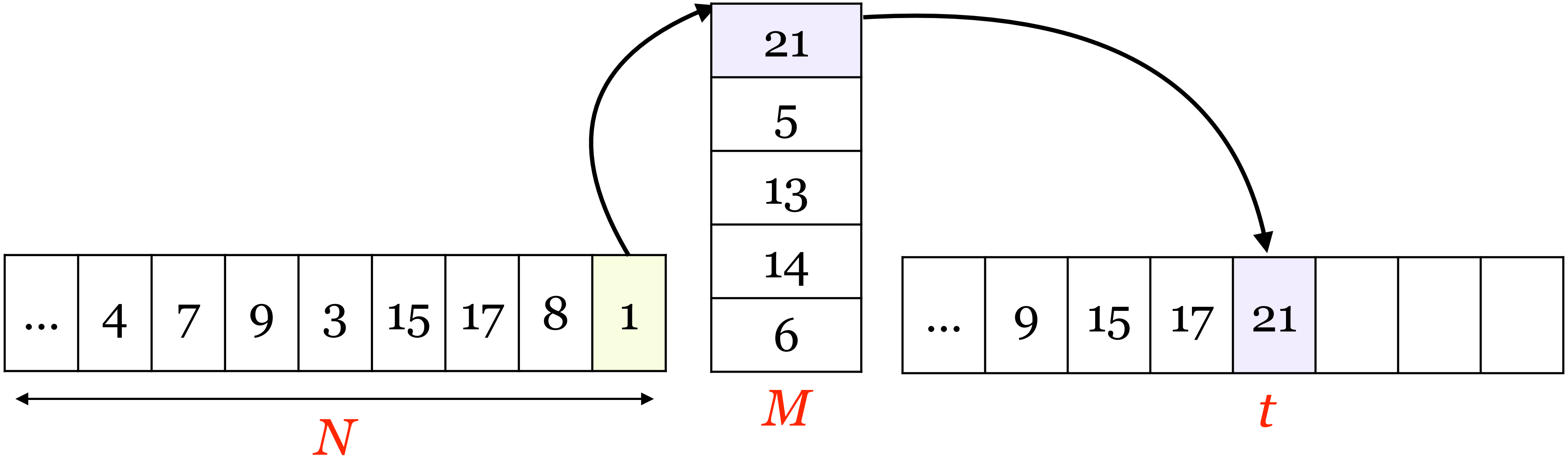
Run Generation: Problem Definition

- Input: Stream of N elements
- 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)



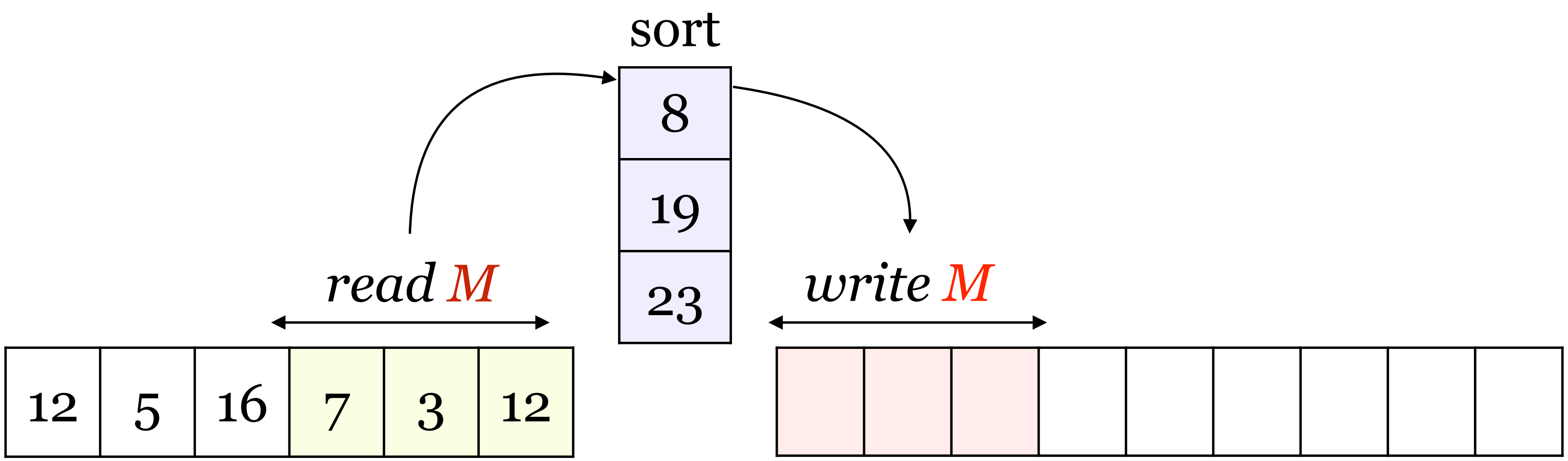
Run Generation: Problem Definition

- Algorithm decides what to eject based on
    - Contents of buffer, last element written
- Algorithm cannot arbitrarily access input or output
    - Read next-in-order from input, append to output
- Algorithm is at time step *t* if it has written *t* elements



Naive Run Generation: Base Case of External Memory Merge Sort

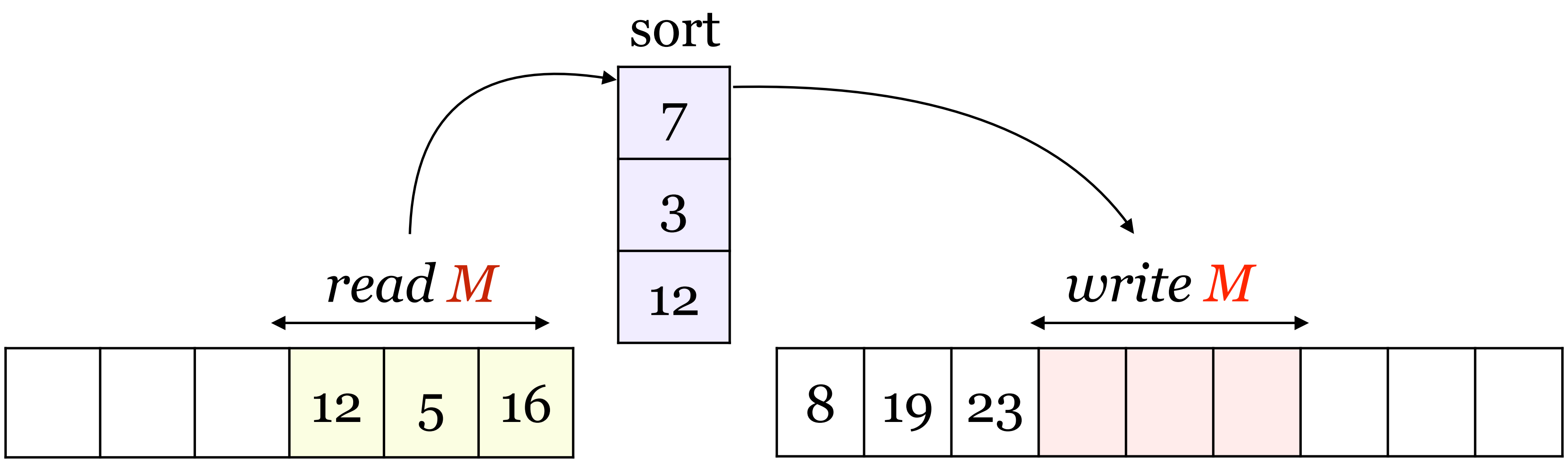
- Bring *M* elements to the buffer
- Sort them
- Write all of them to disk



Runs of length *M*

Naive Run Generation: Base Case of External Memory Merge Sort

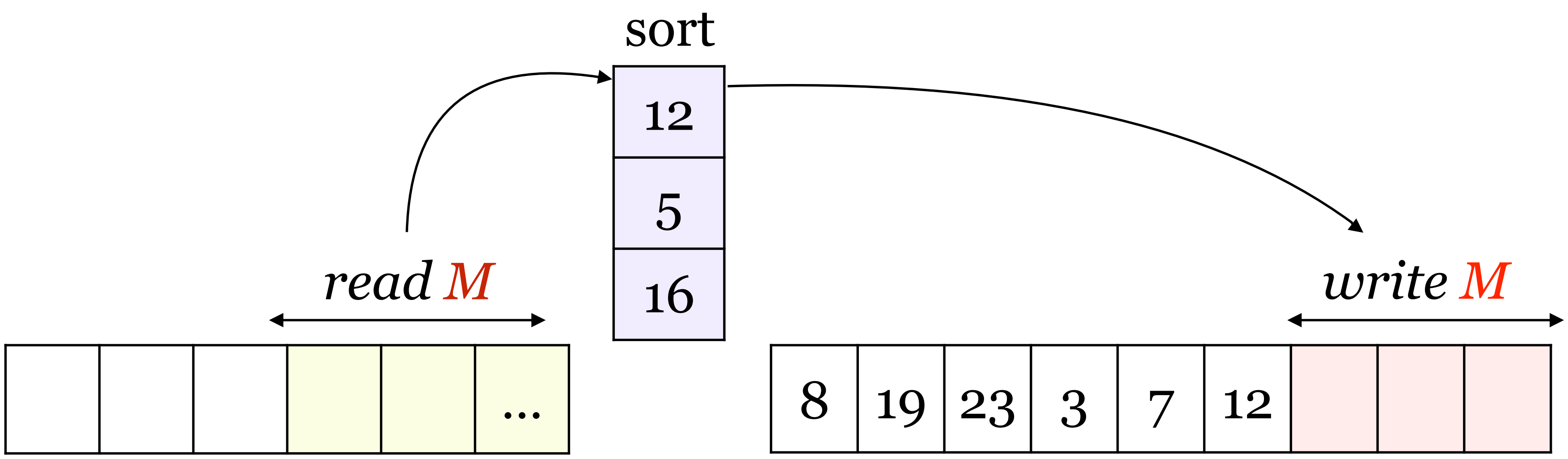
- Bring M elements to the buffer
- Sort them
- Write all of them to disk



Runs of length M

Naive Run Generation: Base Case of External Memory Merge Sort

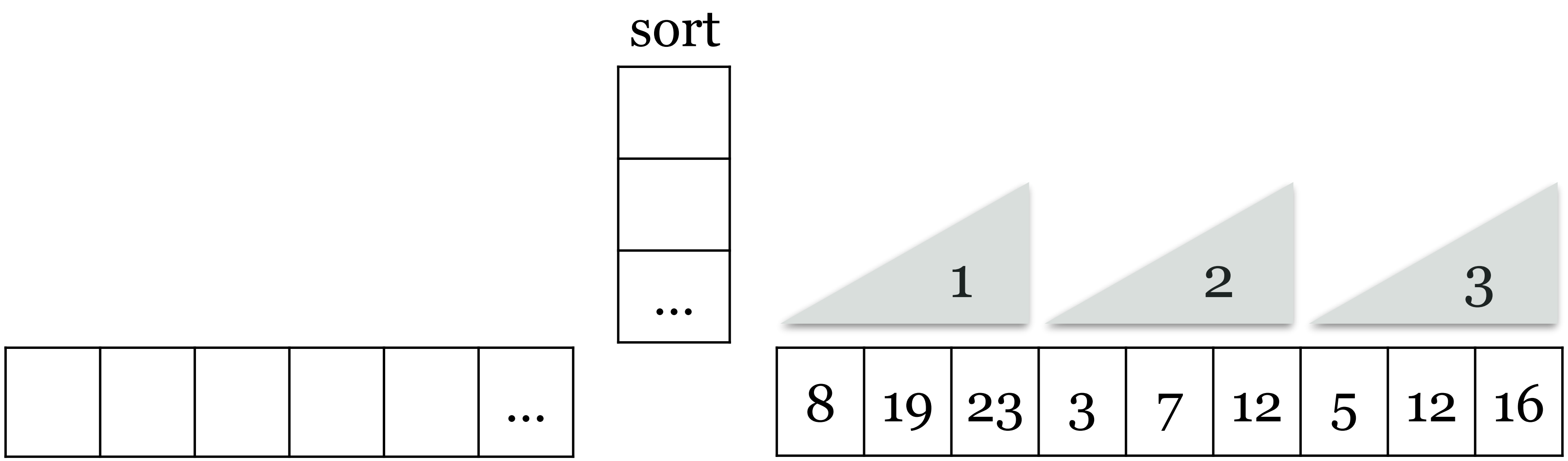
- Bring M elements to the buffer
- Sort them
- Write all of them to disk



Runs of length M

Naive Run Generation: Base Case of External Memory Merge Sort

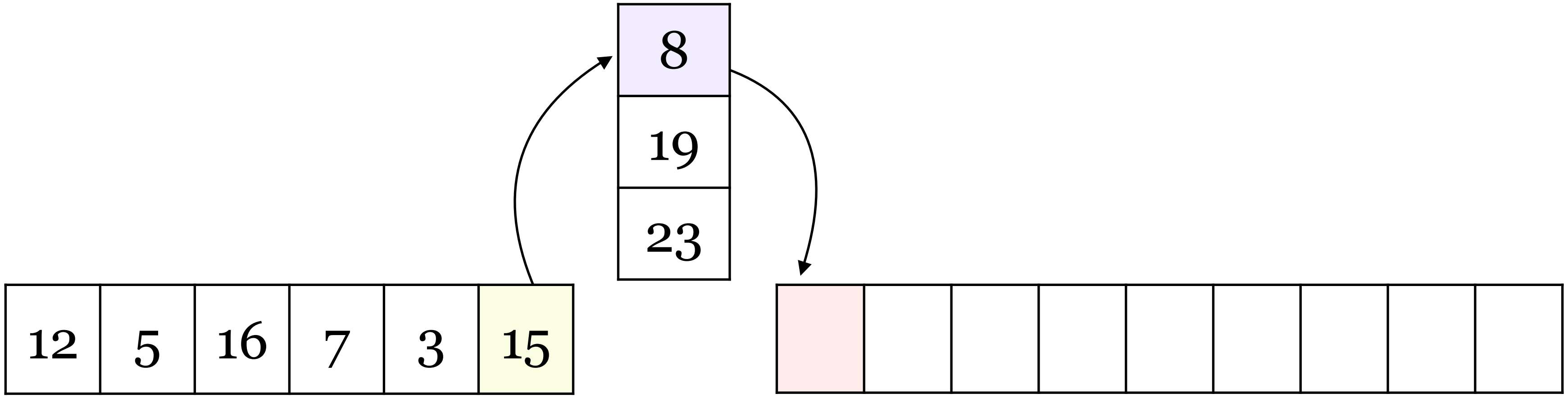
- Bring *M* elements to the buffer
- Sort them
- Write all of them to disk



Runs of length *M*

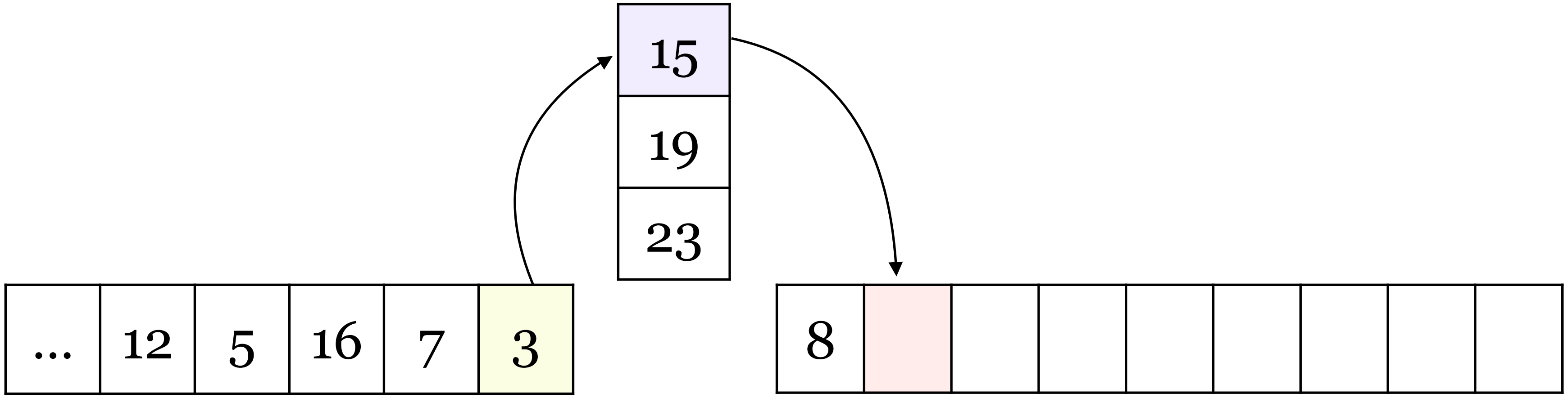
Classic Algorithm: Replacement Selection

- Replacement Selection [Goetz 63]:
  - Starting from a full buffer, output smallest element
  - Write smallest element in buffer $\geq$ the last output
  - If no such element, start a new run and continue



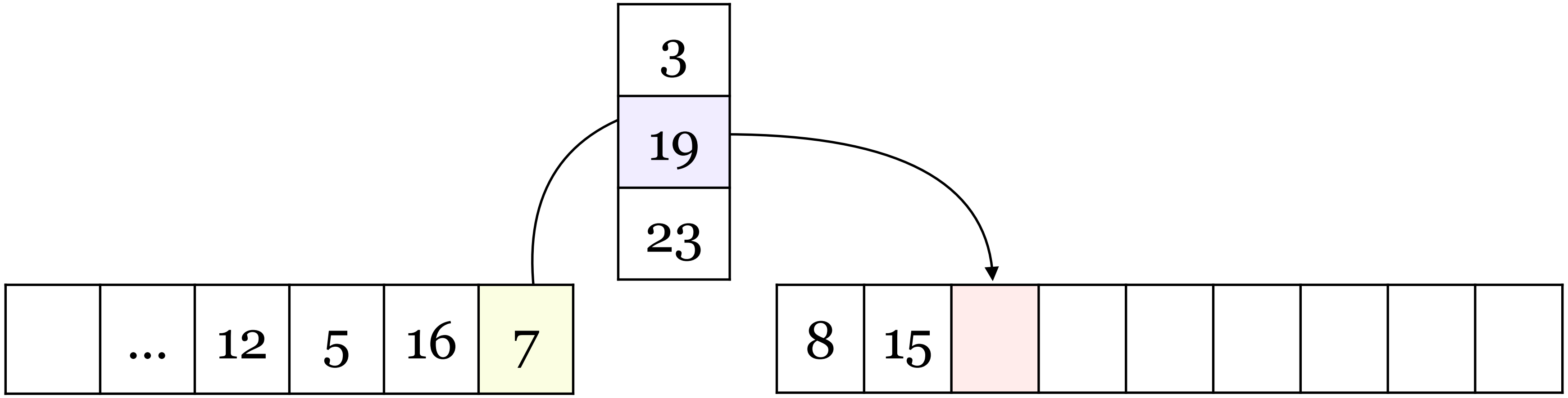
Classic Algorithm: Replacement Selection

- Replacement Selection [Goetz 63]:
  - Starting from a full buffer, output smallest element
  - Write smallest element in buffer $\geq$ the last output
  - If no such element, start a new run and continue



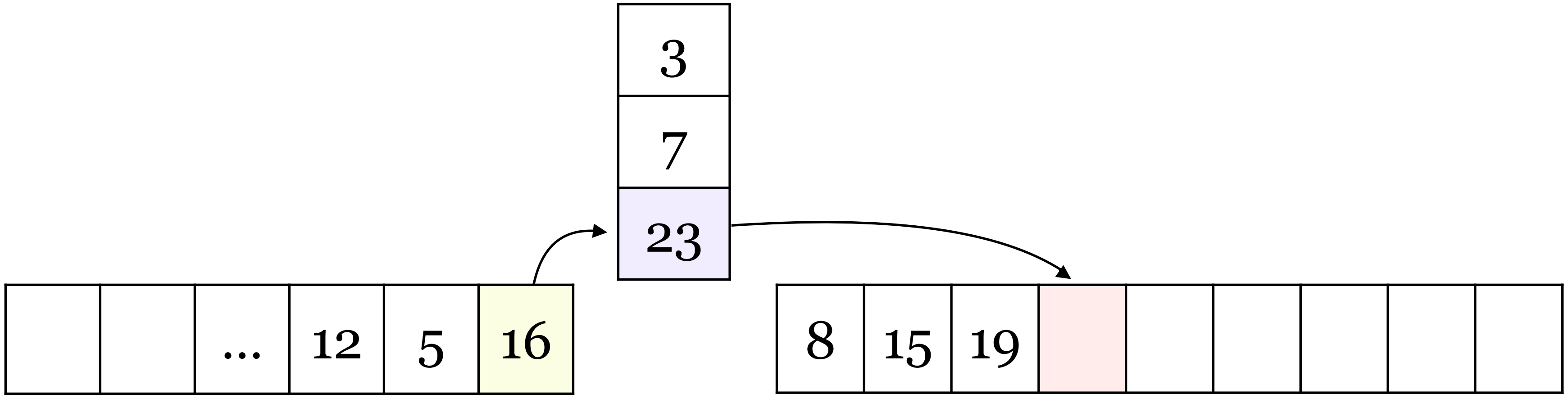
Classic Algorithm: Replacement Selection

- Replacement Selection [Goetz 63]:
  - Starting from a full buffer, output smallest element
  - Write smallest element in buffer $\geq$ the last output
  - If no such element, start a new run and continue



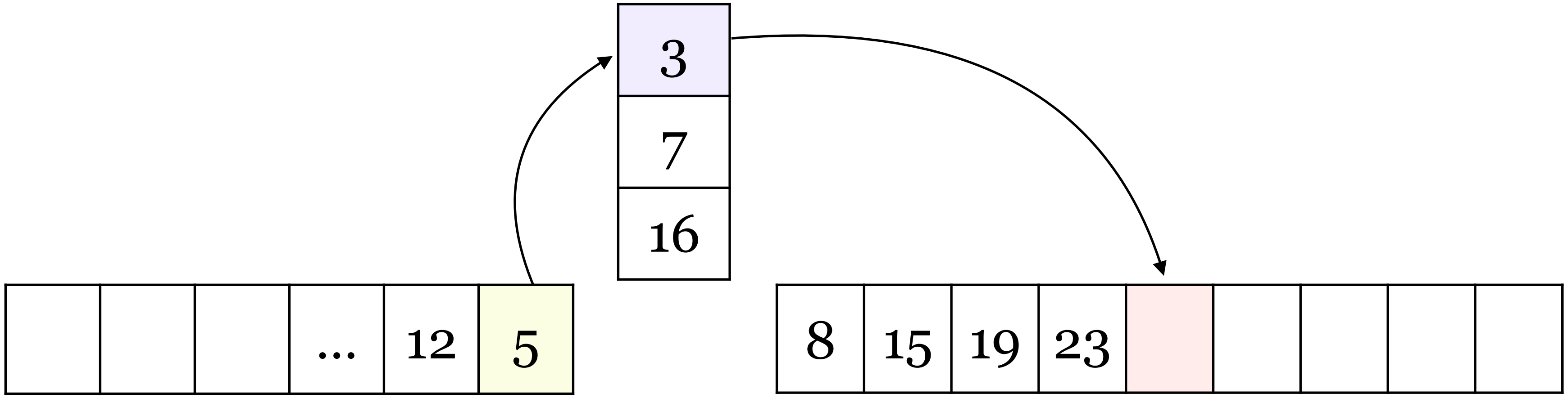
Classic Algorithm: Replacement Selection

- Replacement Selection [Goetz 63]:
  - Starting from a full buffer, output smallest element
  - Write smallest element in buffer $\geq$ the last output
  - If no such element, start a new run and continue



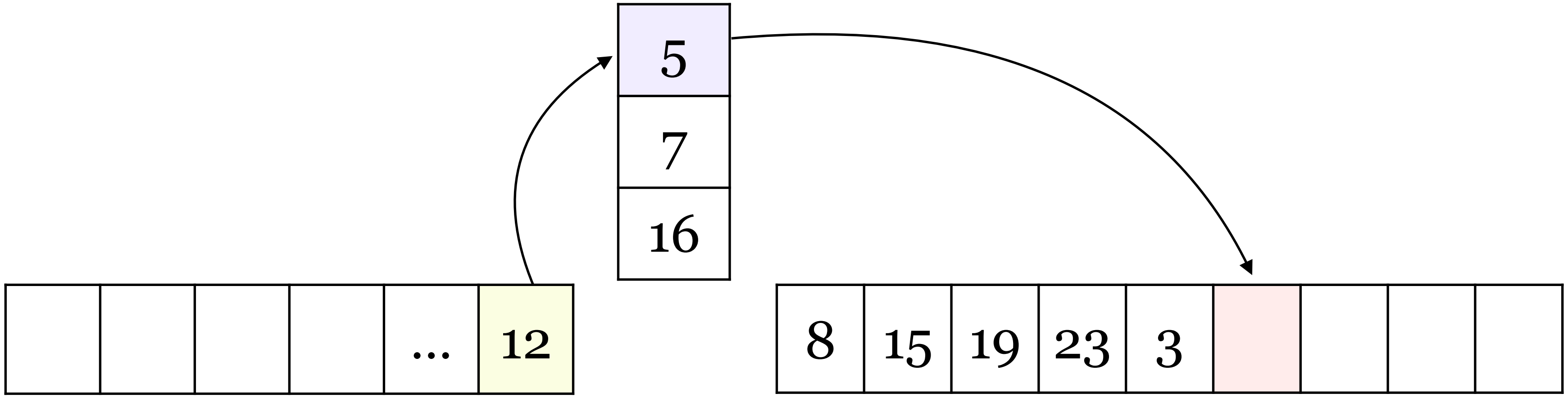
Classic Algorithm: Replacement Selection

- Replacement Selection [Goetz 63]:
  - Starting from a full buffer, output smallest element
  - Write smallest element in buffer $\geq$ the last output
  - If no such element, start a new run and continue



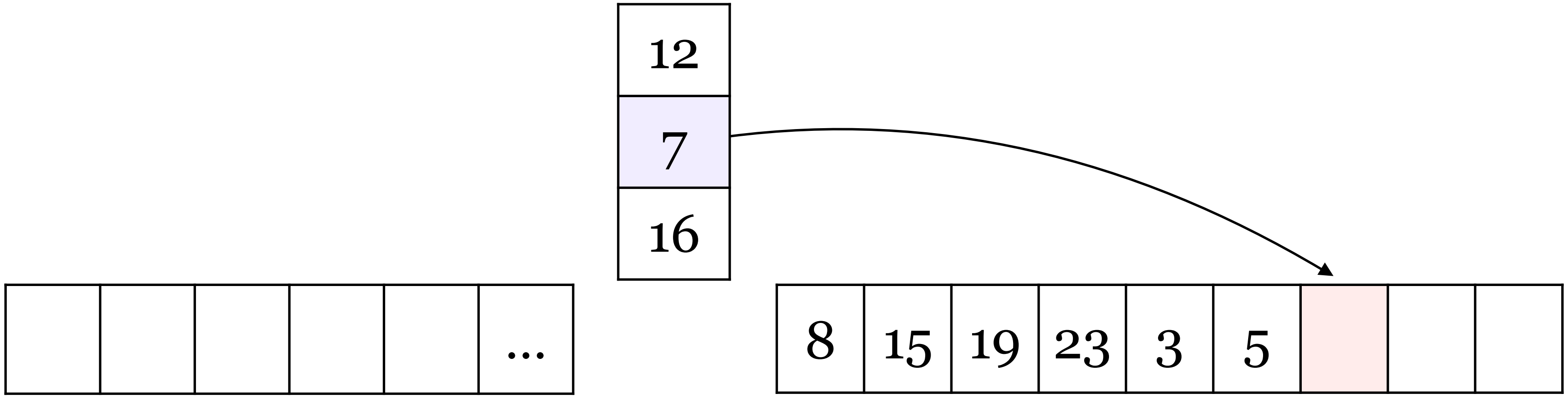
Classic Algorithm: Replacement Selection

- Replacement Selection [Goetz 63]:
  - Starting from a full buffer, output smallest element
  - Write smallest element in buffer $\geq$ the last output
  - If no such element, start a new run and continue



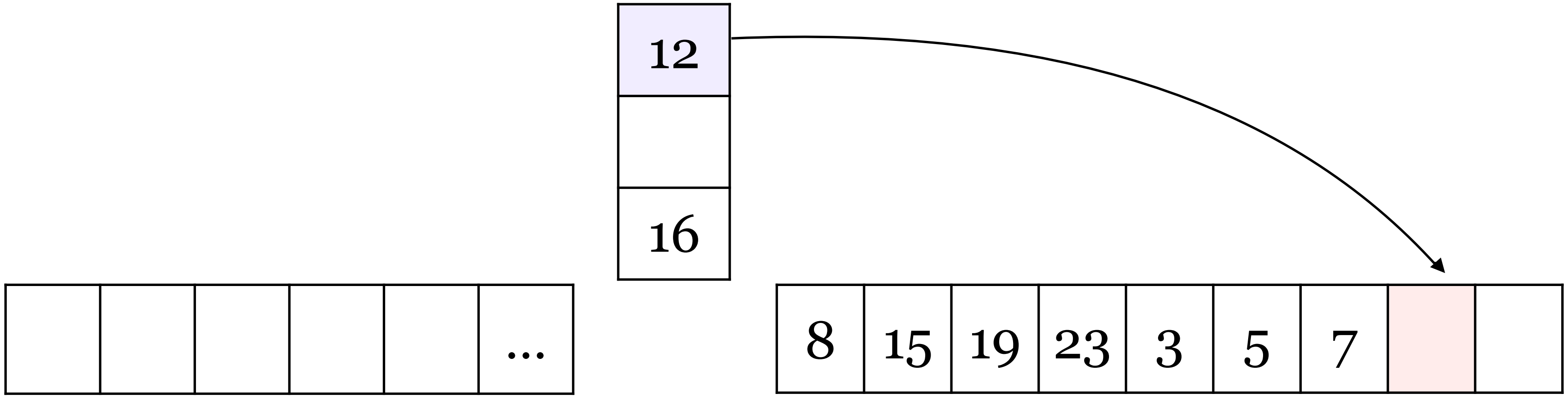
Classic Algorithm: Replacement Selection

- Replacement Selection [Goetz 63]:
  - Starting from a full buffer, output smallest element
  - Write smallest element in buffer $\geq$ the last output
  - If no such element, start a new run and continue



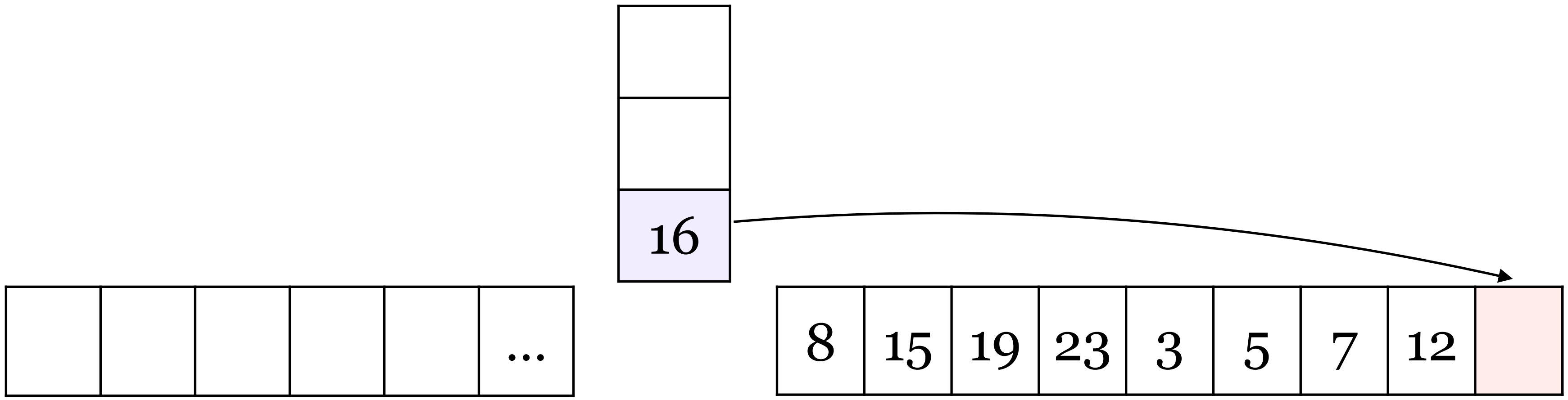
Classic Algorithm: Replacement Selection

- Replacement Selection [Goetz 63]:
  - Starting from a full buffer, output smallest element
  - Write smallest element in buffer $\geq$ the last output
  - If no such element, start a new run and continue



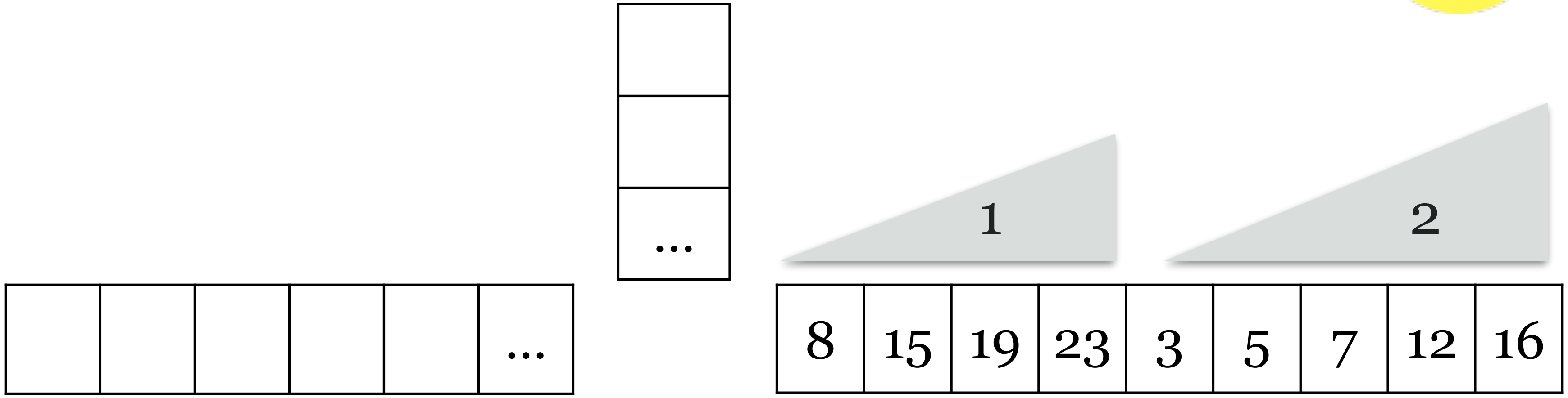
Classic Algorithm: Replacement Selection

- Replacement Selection [Goetz 63]:
  - Starting from a full buffer, output smallest element
  - Write smallest element in buffer $\geq$ the last output
  - If no such element, start a new run and continue



Classic Algorithm: Replacement Selection

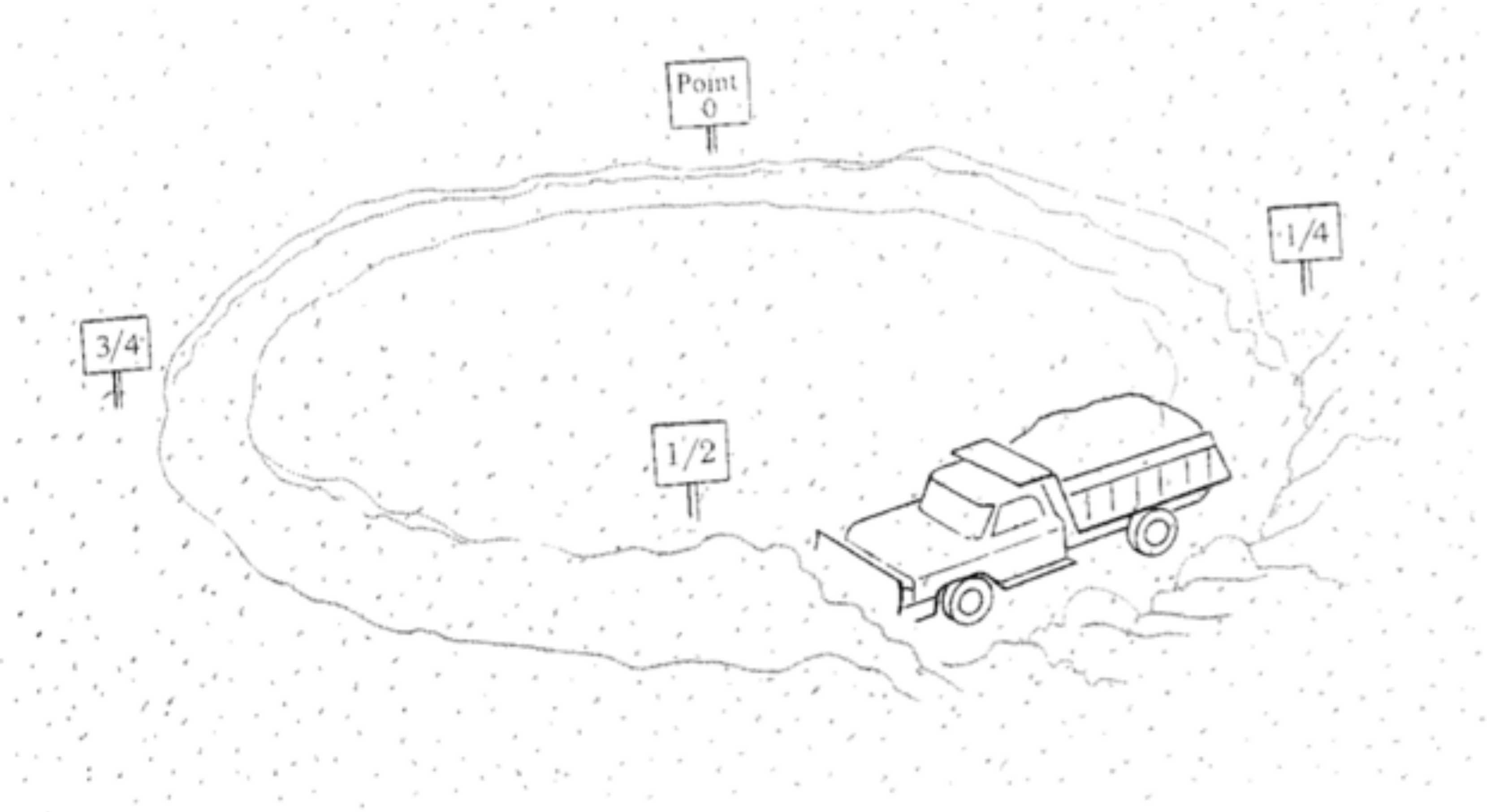
- Fewer runs on nearly sorted input
    - If every element is within M of its rank - one run



Runs of length > M

Performance of Replacement Selection

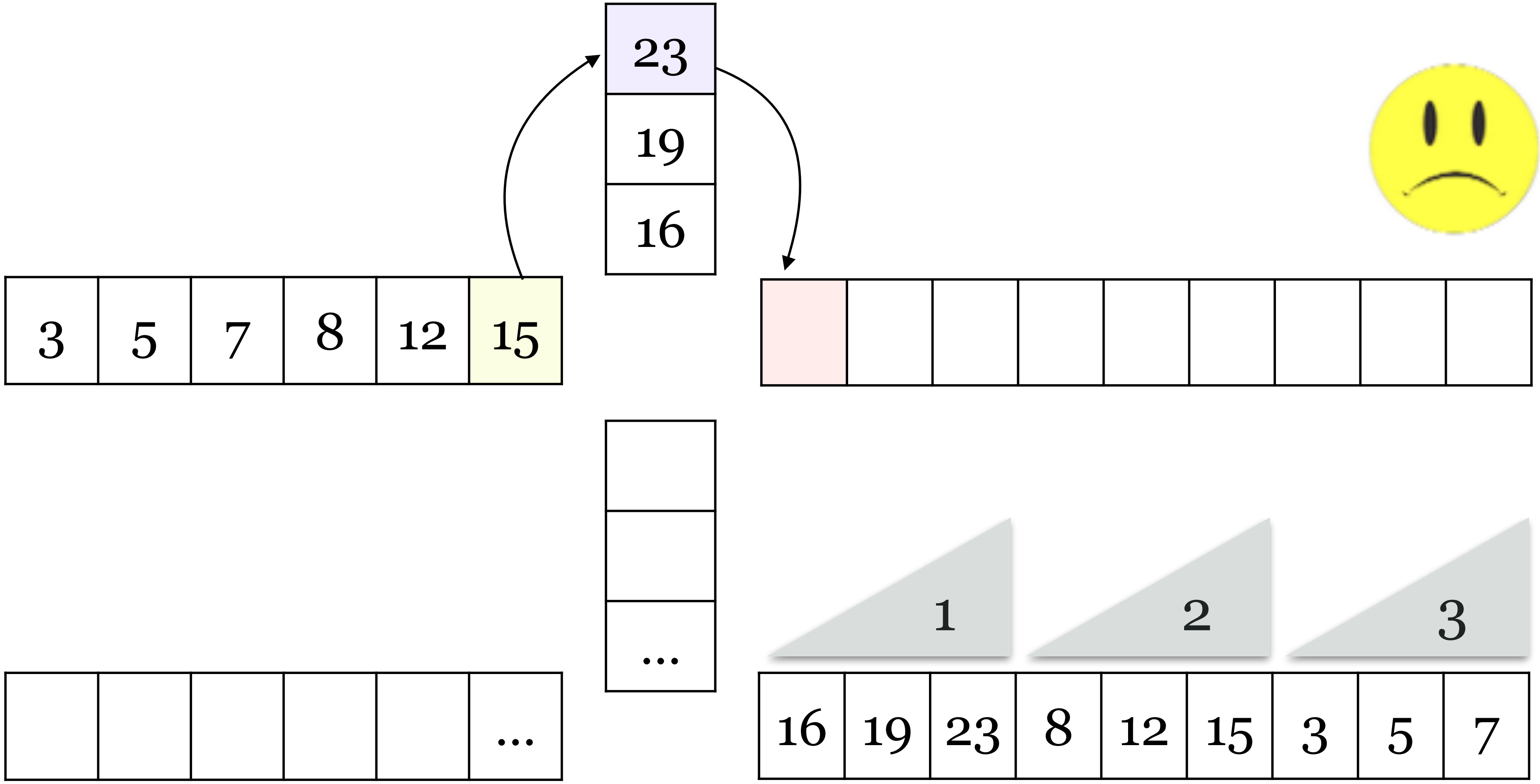
- On random data, expected length of a run is 2M



"The perpetual plow on its ceaseless cycle." - Knuth

Performance of Replacement Selection

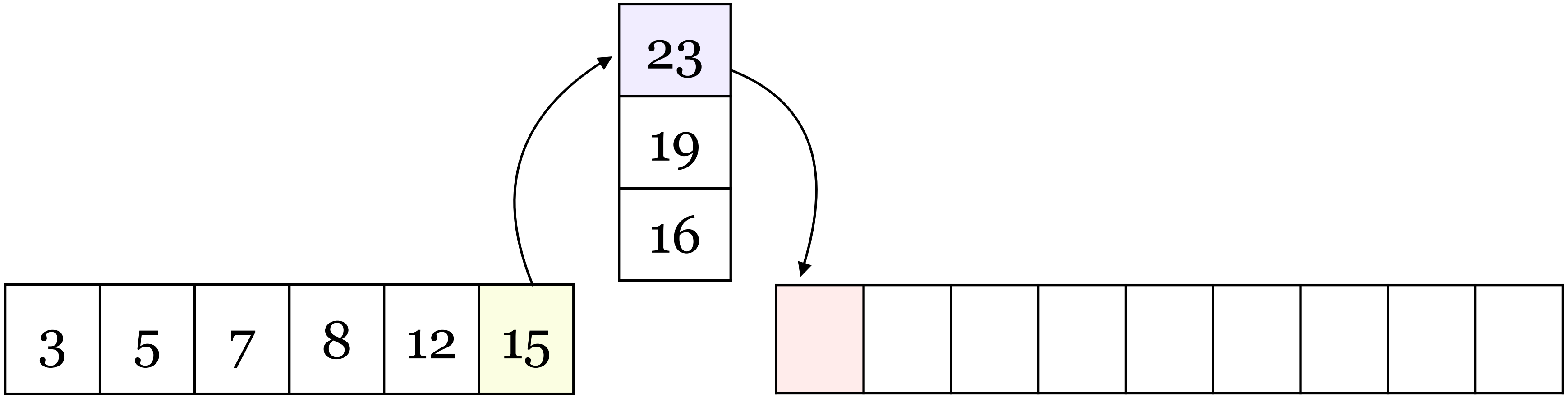
- However, on inversely sorted input…



Runs of length *M*

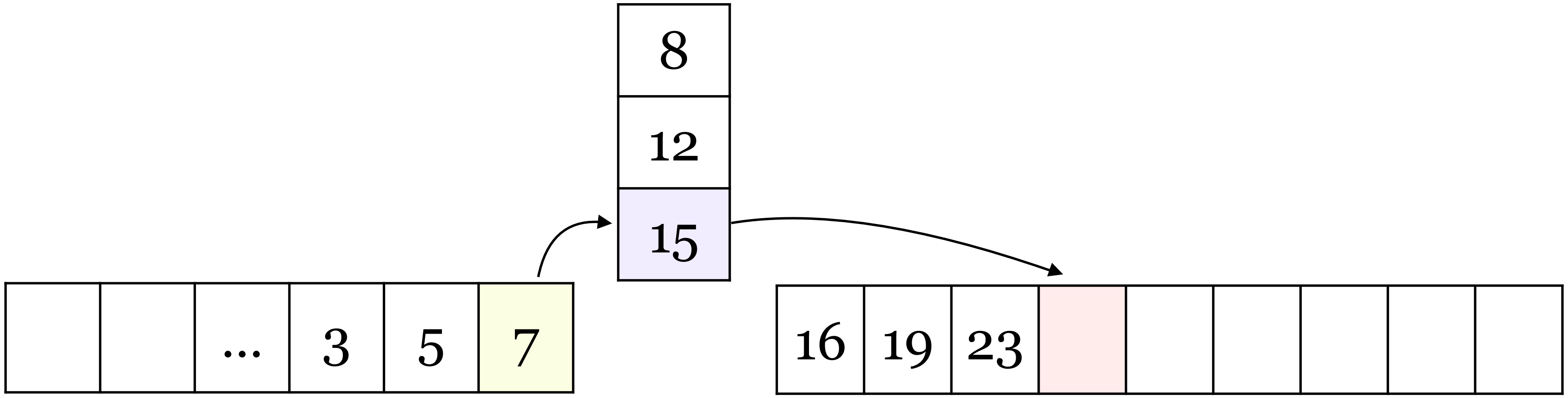
Alternating-Up-Down Replacement Selection

- Deterministically alternate between up and down runs



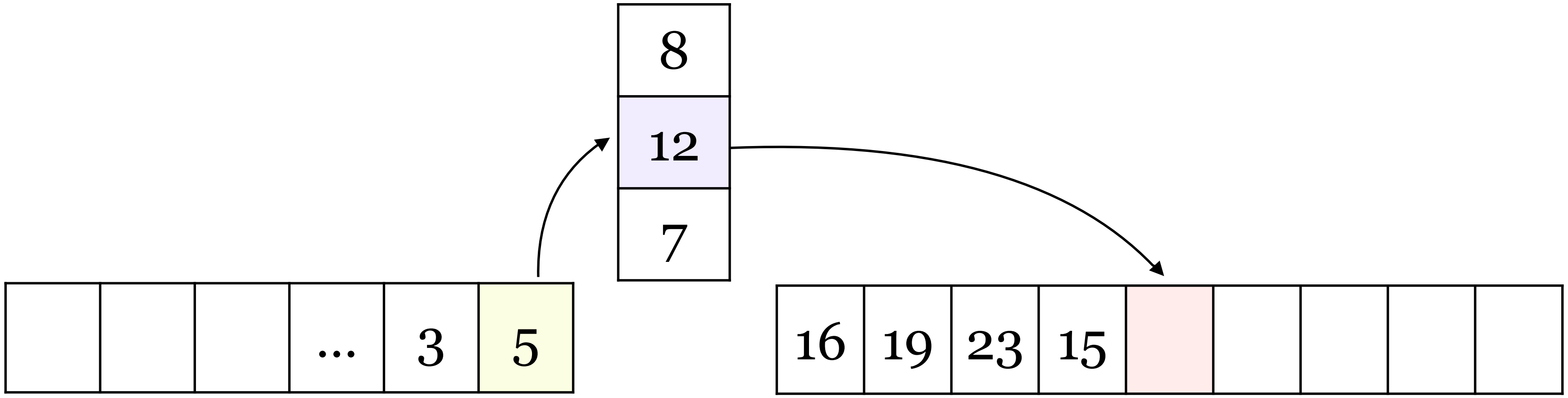
Alternating-Up-Down Replacement Selection

- Deterministically alternate between up and down runs



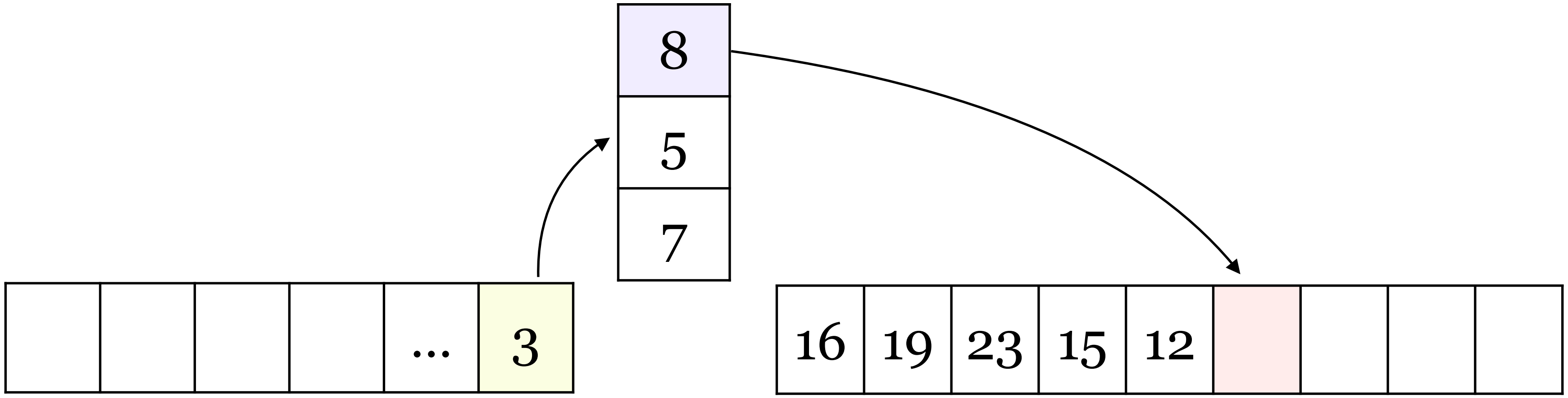
Alternating-Up-Down Replacement Selection

- Deterministically alternate between up and down runs



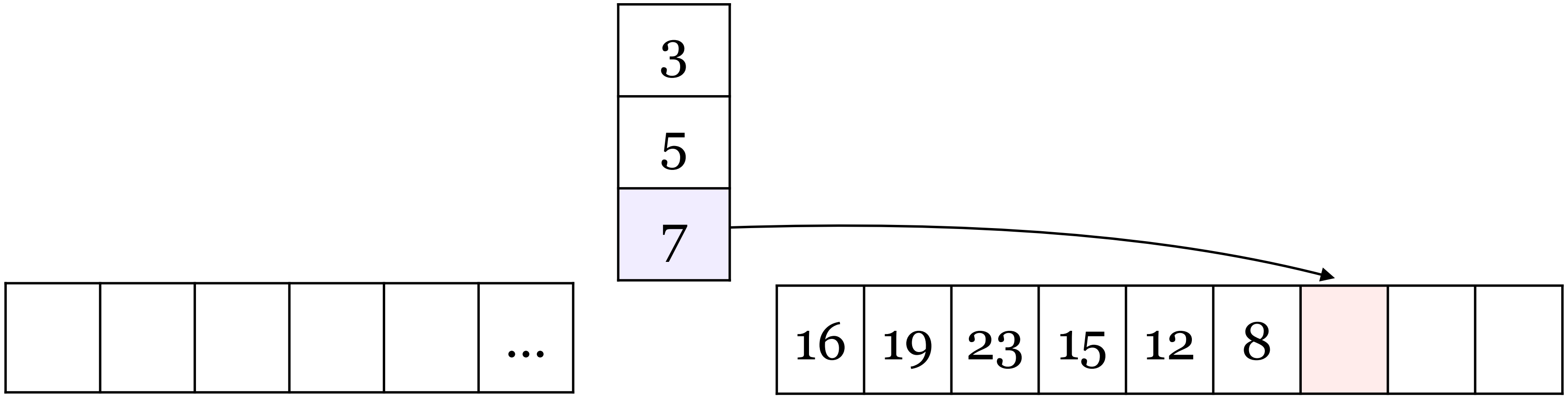
Alternating-Up-Down Replacement Selection

- Deterministically alternate between up and down runs



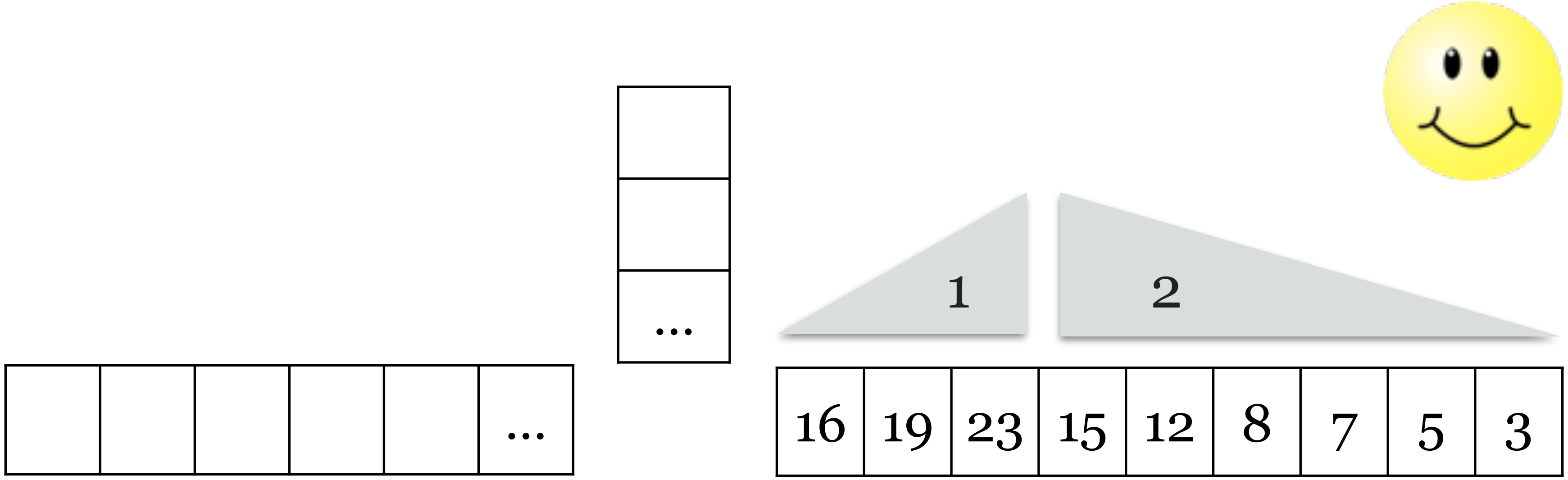
Alternating-Up-Down Replacement Selection

- Deterministically alternate between up and down runs



Alternating-Up-Down Replacement Selection

- Deterministically alternate between up and down runs



Runs of length > *M*

Alternating-Up-Down Replacement Selection

- *Is this better than replacement selection?*

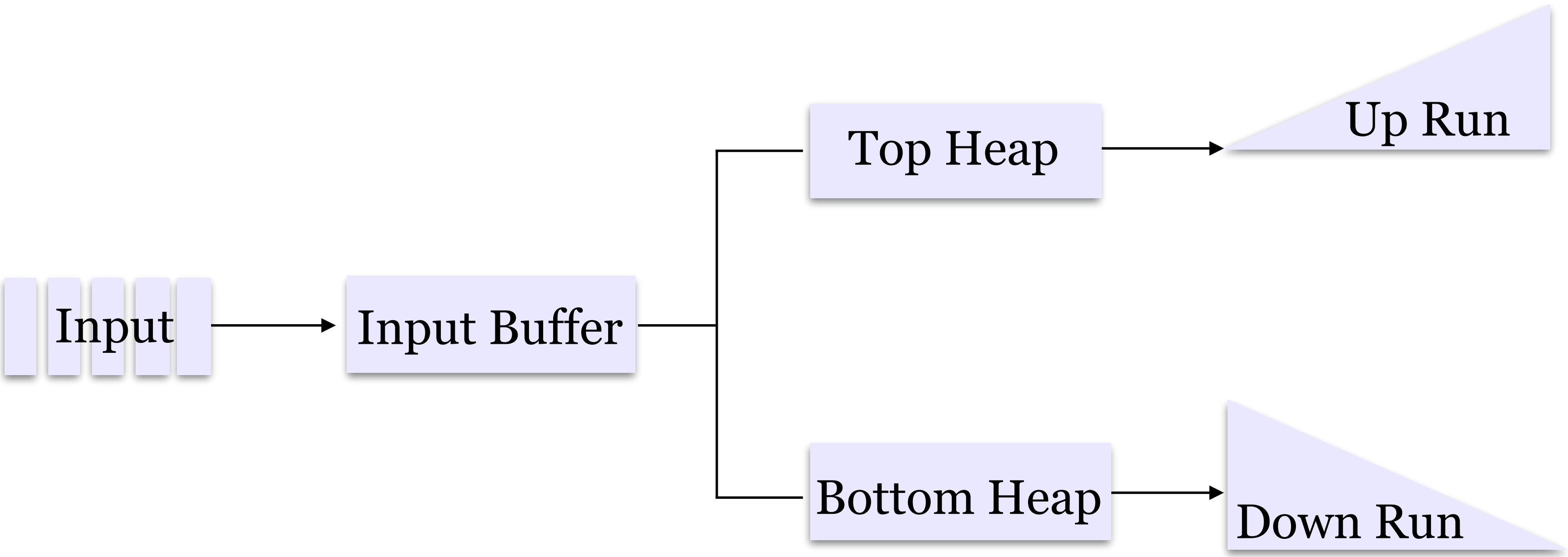
Alternating-Up-Down Replacement Selection

- *Is this better than replacement selection?*

- [Knuth 63] On random data, it is *worse*
    - Average run length is 1.5M, compared to 2M

Two-Way Replacement Selection

- [Martinez-Palau et al. VLDB 10]
  - Heuristically *choose* between an *up* and *down run*
  - Slightly better than Replacement Selection on *some* data



To run up or down, that is the question…

Our Main Contributions

- Theoretical foundation of the run generation problem
- Analyze structural properties of run generation algorithms

" My Momma always said smart things about life and chocolates... But I need to know the theory behind it.."

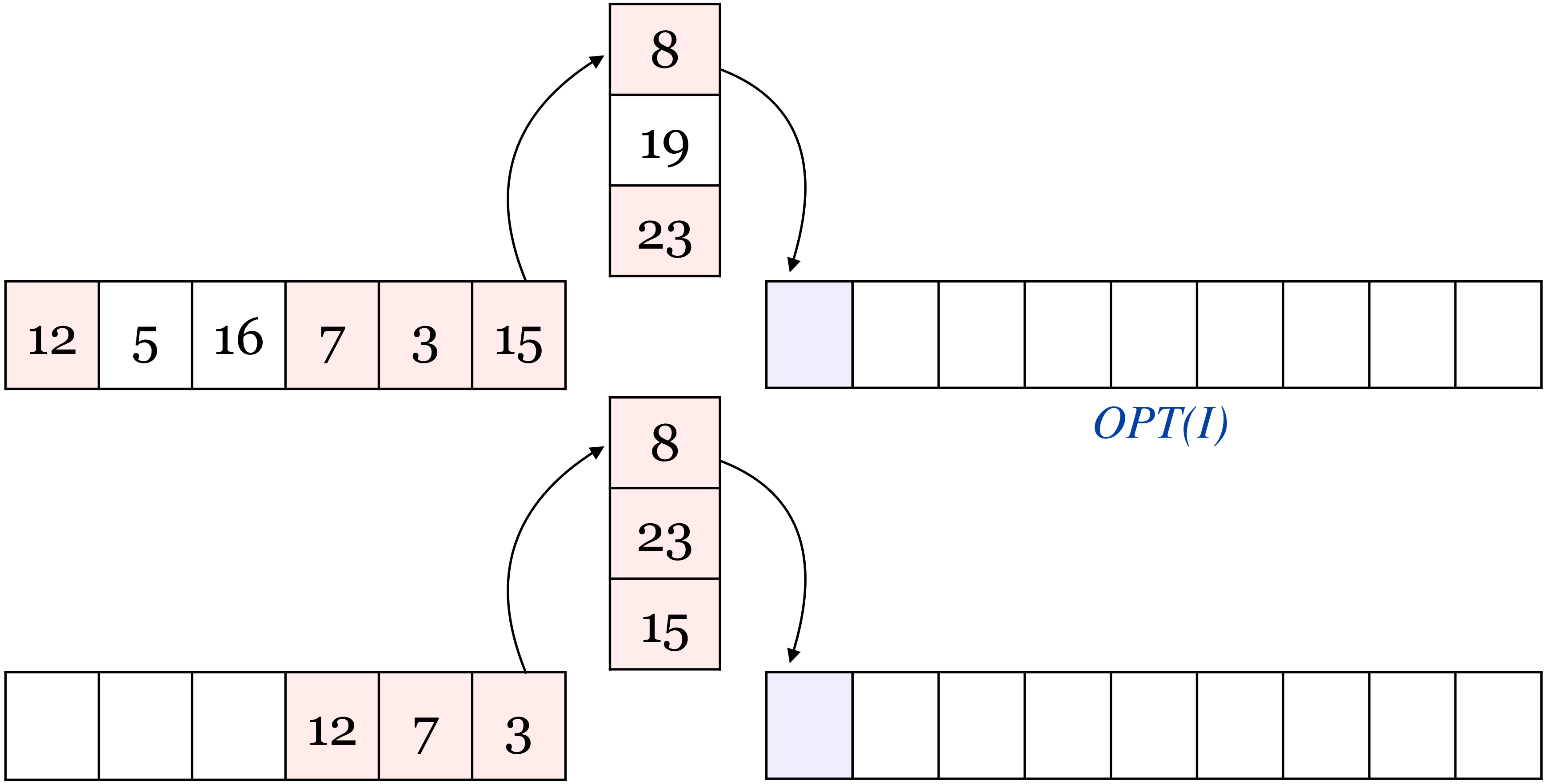
Our Results

- Alternating-Up-Down Replacement Selection is
    - 2-approximation
    - Best possible
- Improve approximation ratio with *resource augmentation*
- Improve performance when input is *nearly sorted*

" My Momma always said smart things about life and chocolates... But I need to know the theory behind it.."

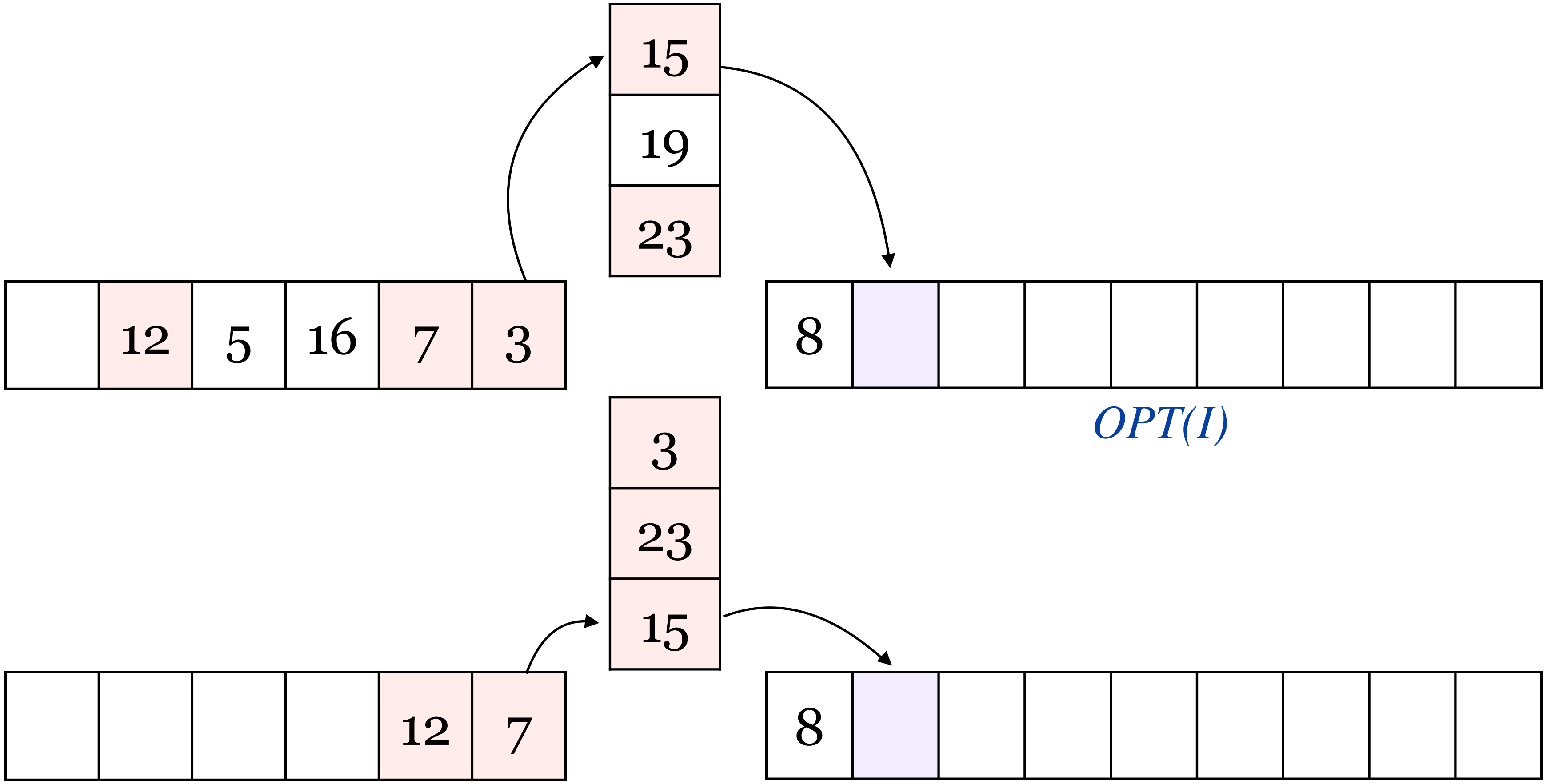
Structural Properties of Run Generation

- If *I’* is a subsequence of *I*, *OPT(I’)* $\leq$ *OPT(I)*



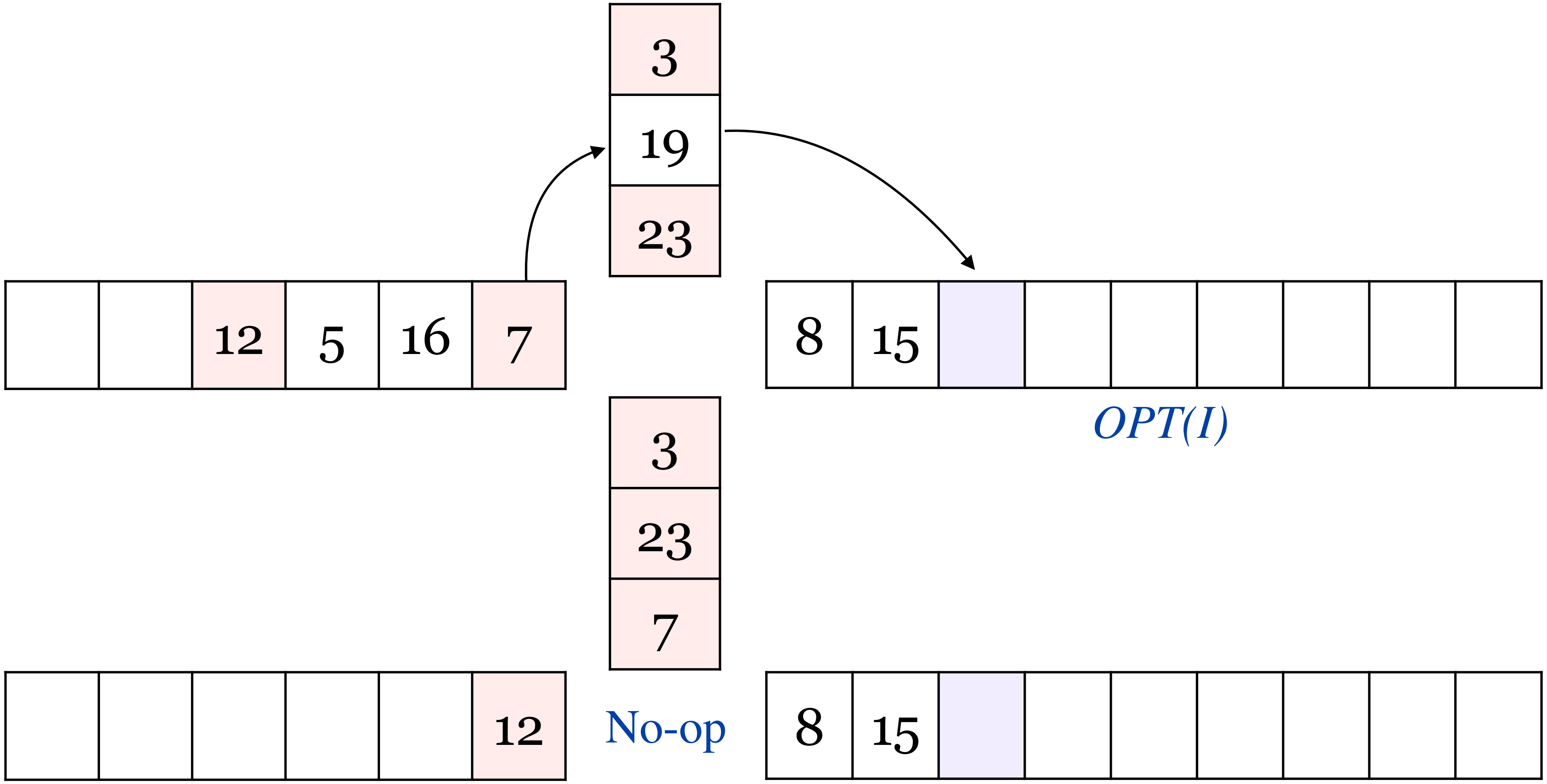
Structural Properties of Run Generation

- If *I'* is a subsequence of *I*, *OPT(I')* $\leq$ *OPT(I)*



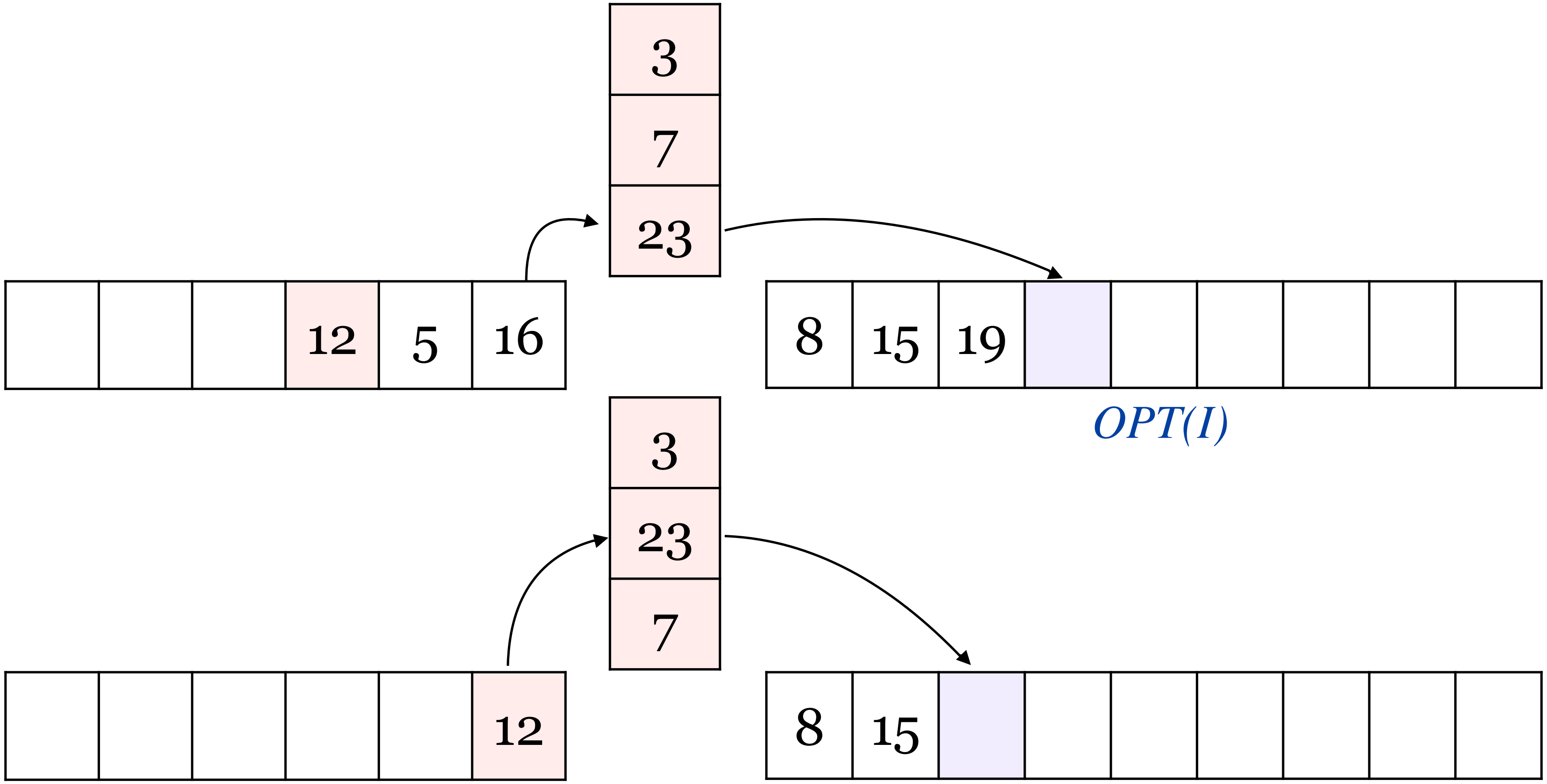
Structural Properties of Run Generation

- If *I'* is a subsequence of *I*, $OPT(I') \leq OPT(I)$



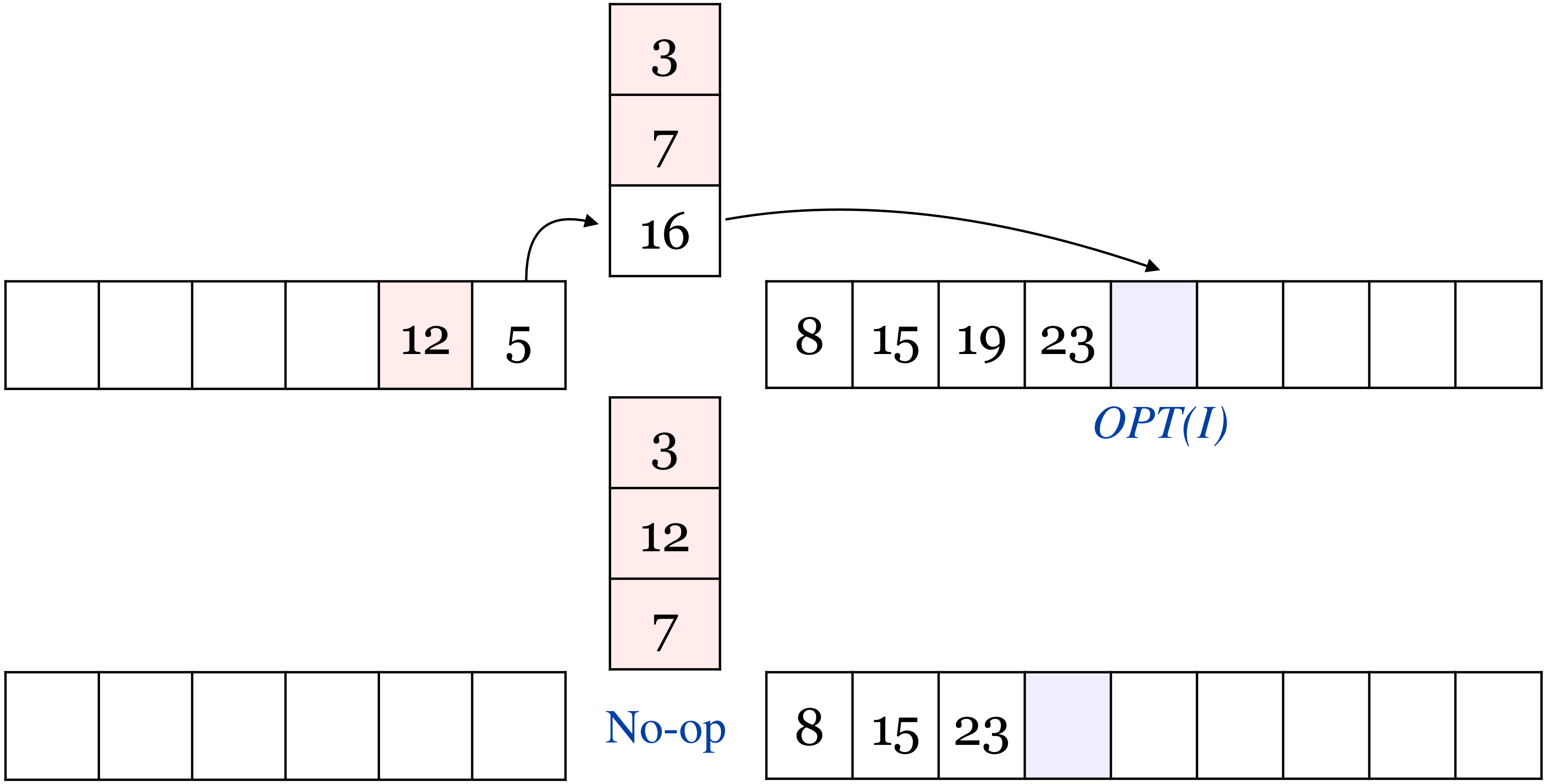
Structural Properties of Run Generation

- If *I’* is a subsequence of *I*, $OPT(I') \leq OPT(I)$



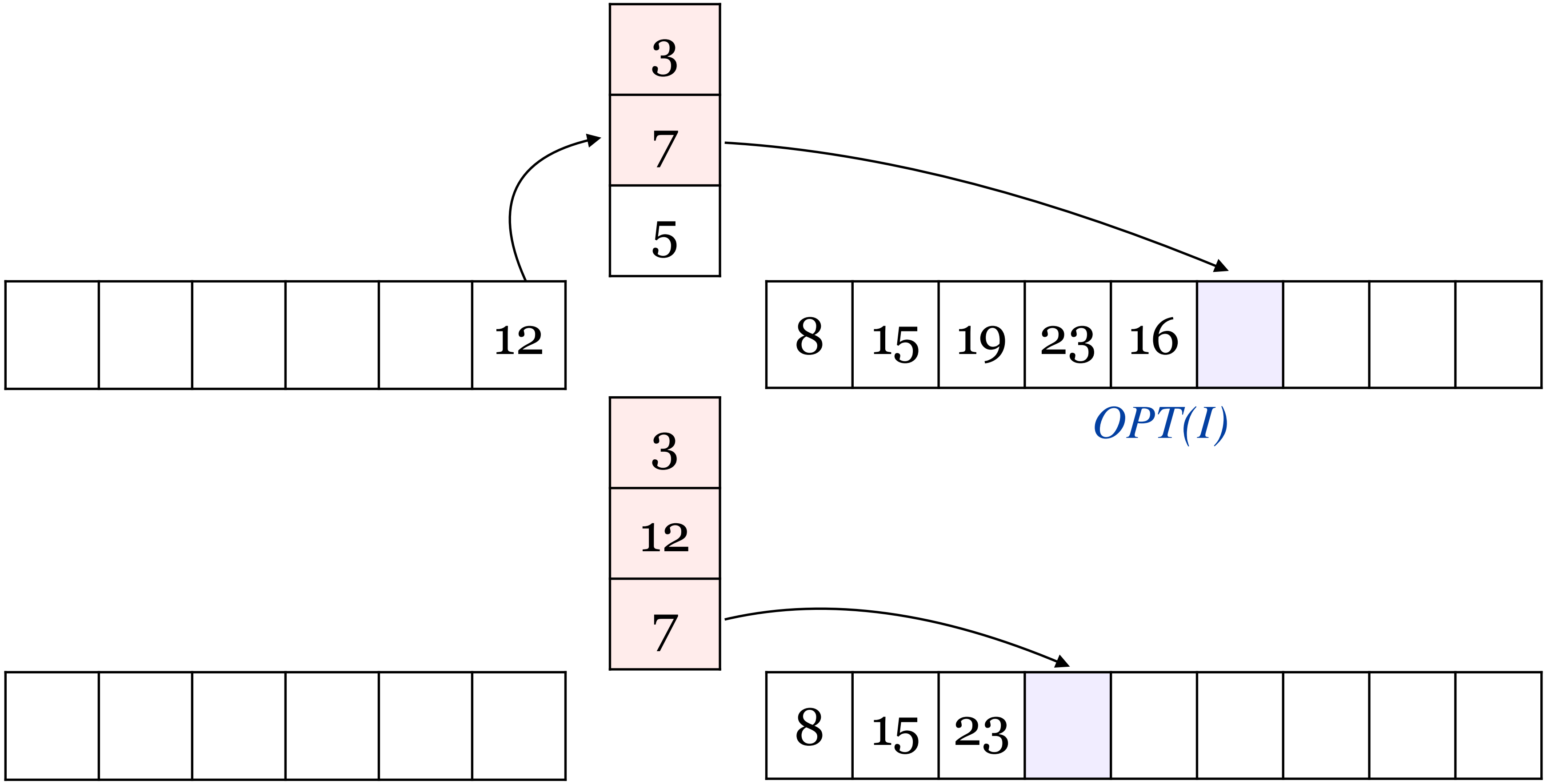
Structural Properties of Run Generation

- If *I'* is a subsequence of *I*, *OPT(I')* $\leq$ *OPT(I)*



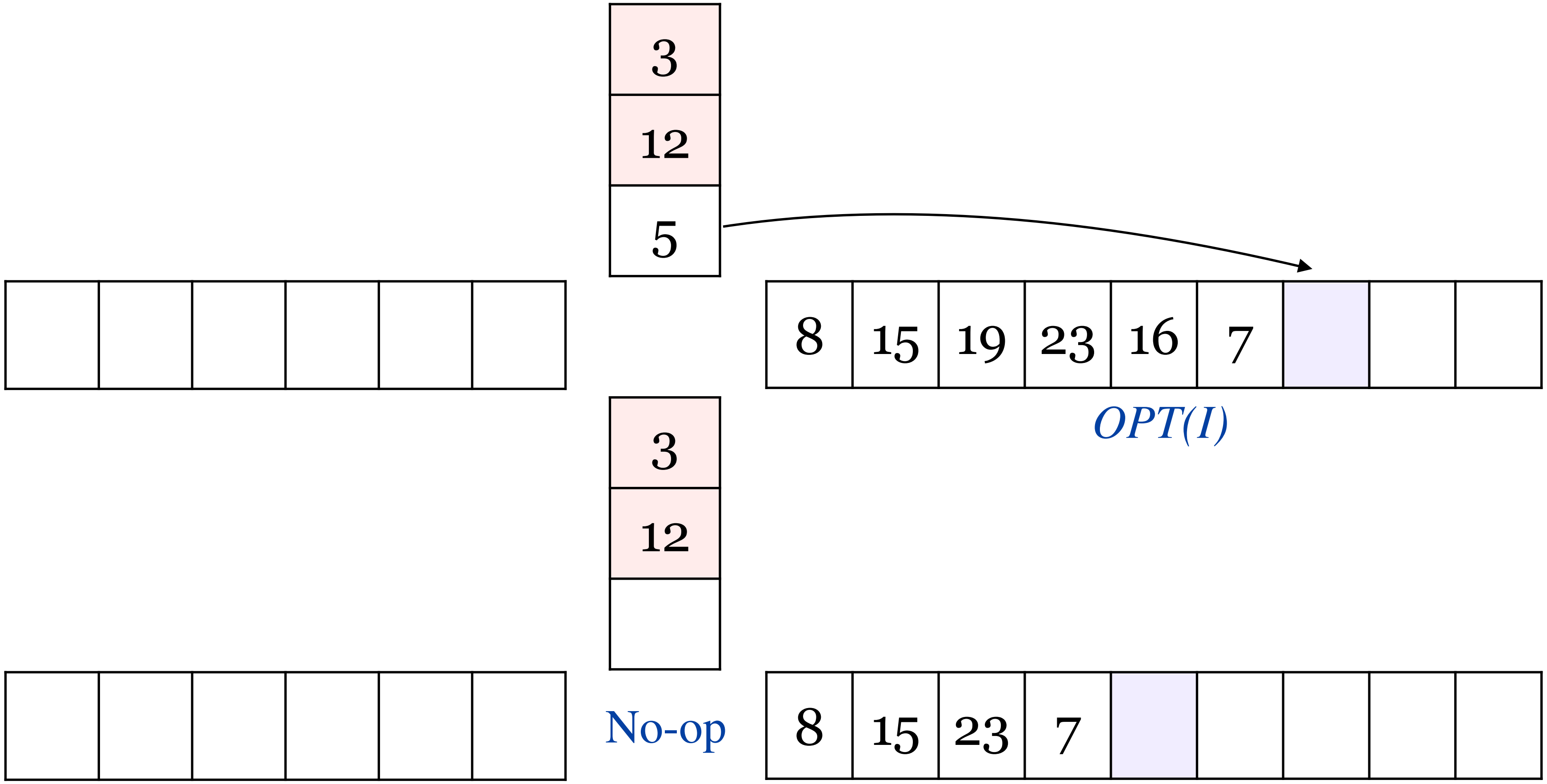
Structural Properties of Run Generation

- If *I'* is a subsequence of *I*, *OPT(I')* $\leq$ *OPT(I)*



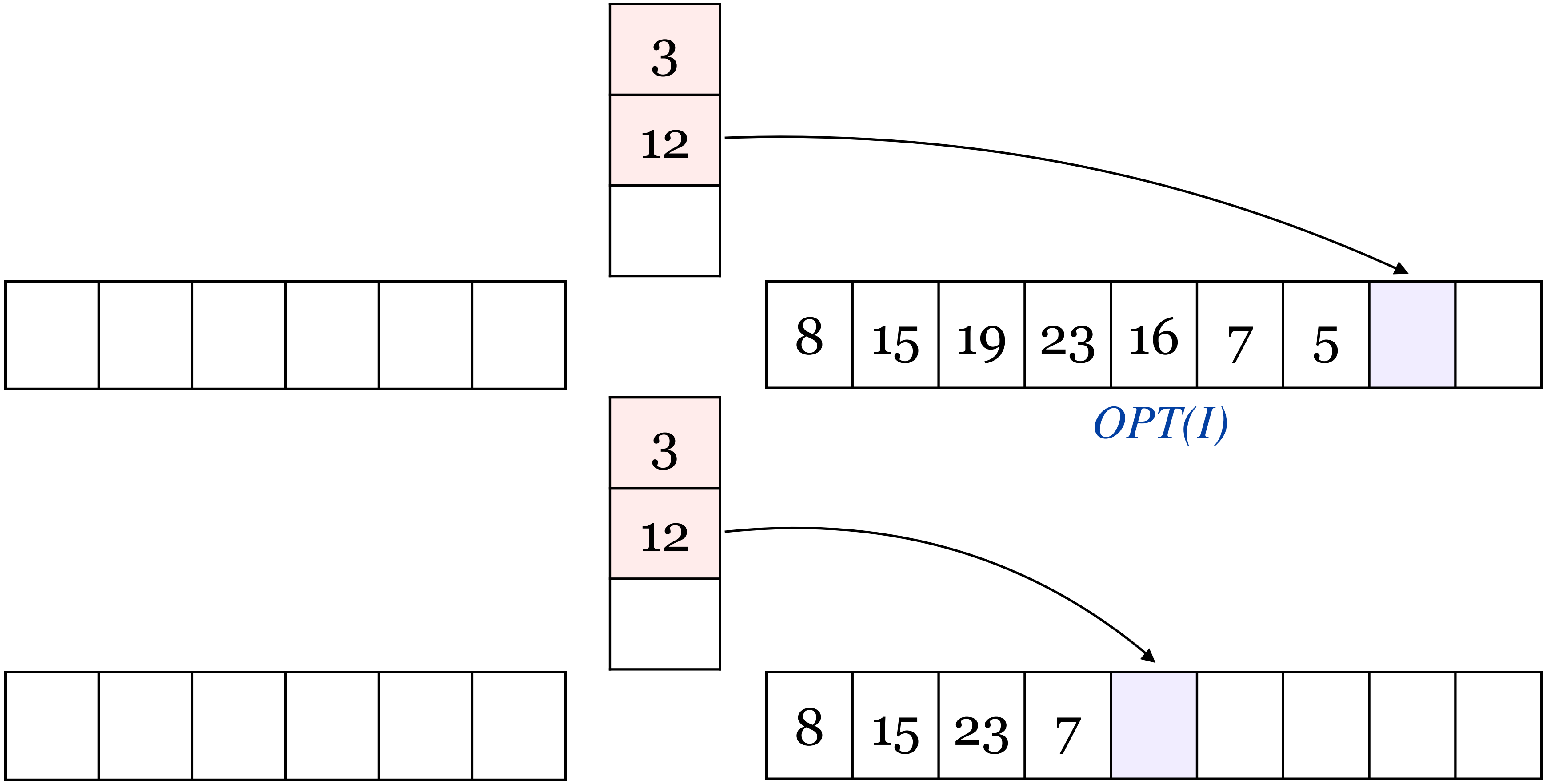
Structural Properties of Run Generation

- If *I'* is a subsequence of *I*, $OPT(I') \leq OPT(I)$



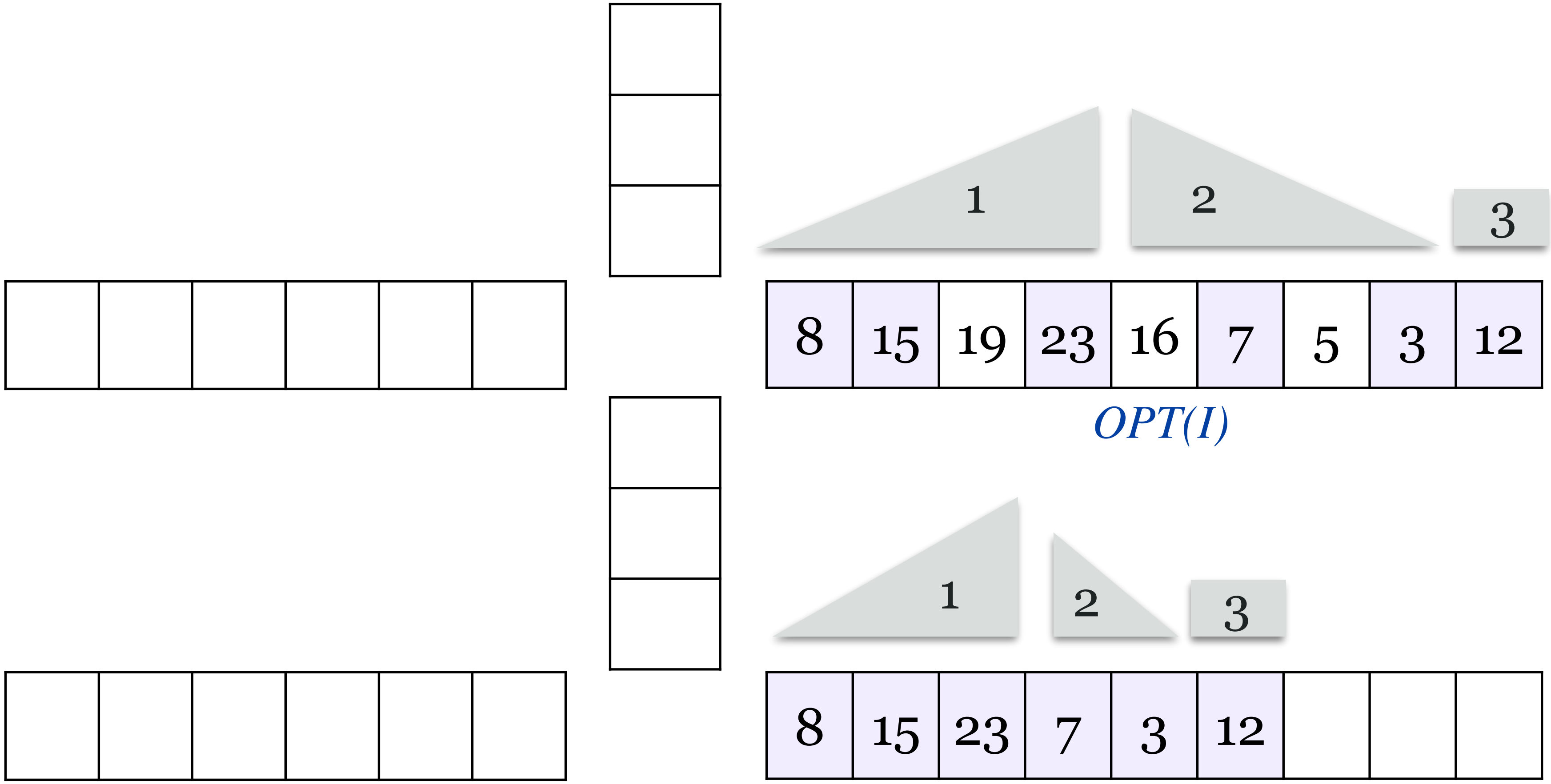
Structural Properties of Run Generation

- If I' is a subsequence of I, $OPT(I') \leq OPT(I)$



Structural Properties of Run Generation

- If *I'* is a subsequence of *I*, *OPT(I')* $\leq$ *OPT(I)*



1 2 3

8 15 19 23 16 7 5 3 12

OPT(I)

1 2 3

8 15 23 7 3 12

Structural Properties of Run Generation

Corollary

- Adding elements to an input stream cannot help

Without loss of generality

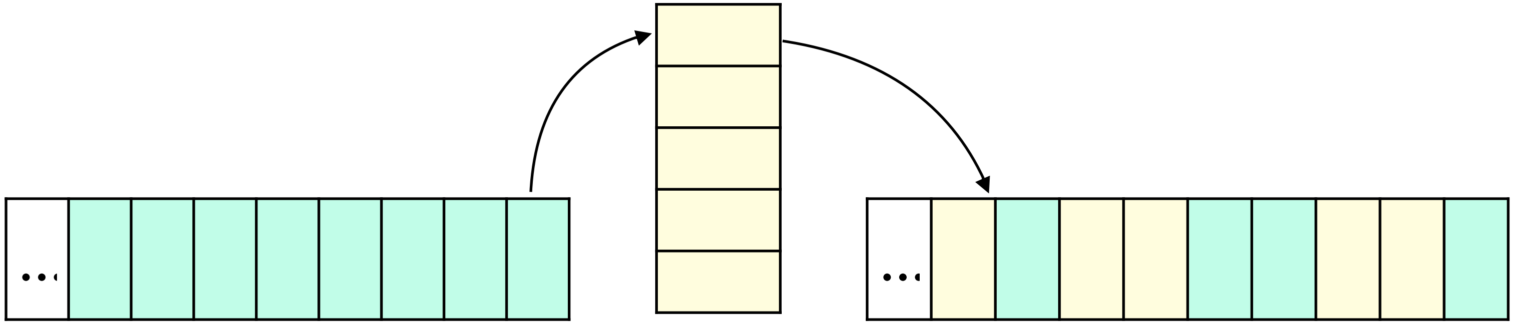
- Algorithm must always write *maximal runs*
  - Never end a run unless forced to
  - Never skip over elements

Structural Properties of Run Generation

Useful Observations

- At each decision point
  - Contents of buffer must have arrived during the last run

Initial buffer always gets written

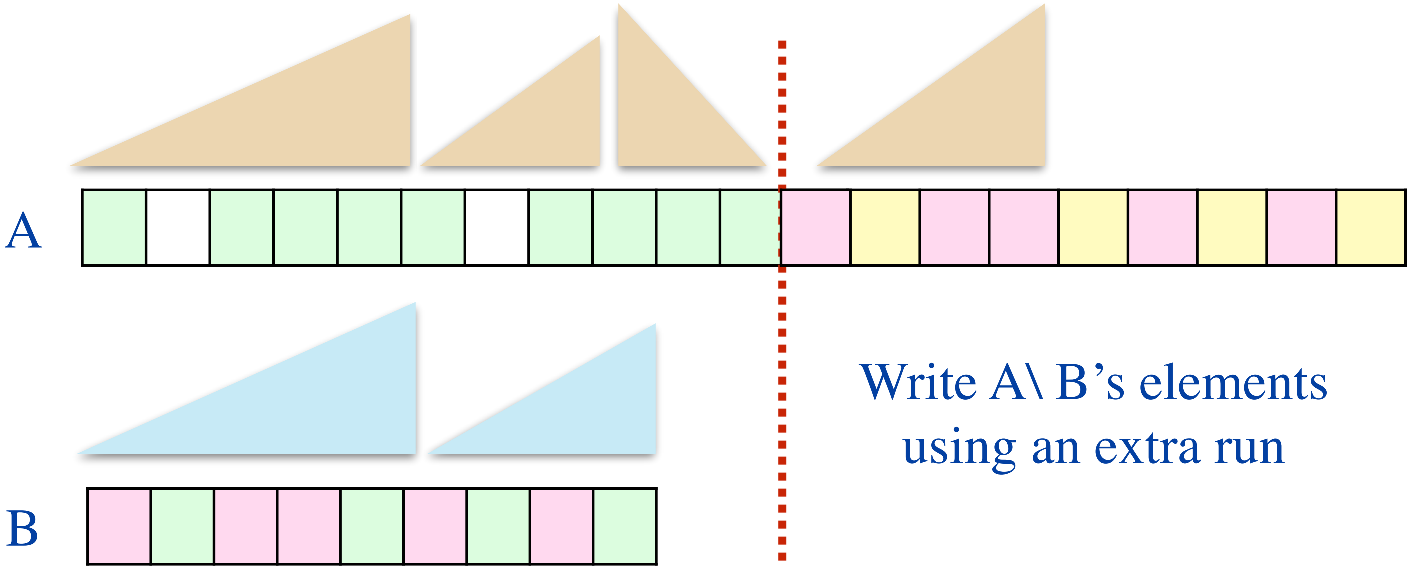


Structural Properties of Run Generation

Useful Observations

- At a decision point if there is a choice between
  A. Writing more elements (possibly using more runs)
  B. Writing less elements (using fewer runs)

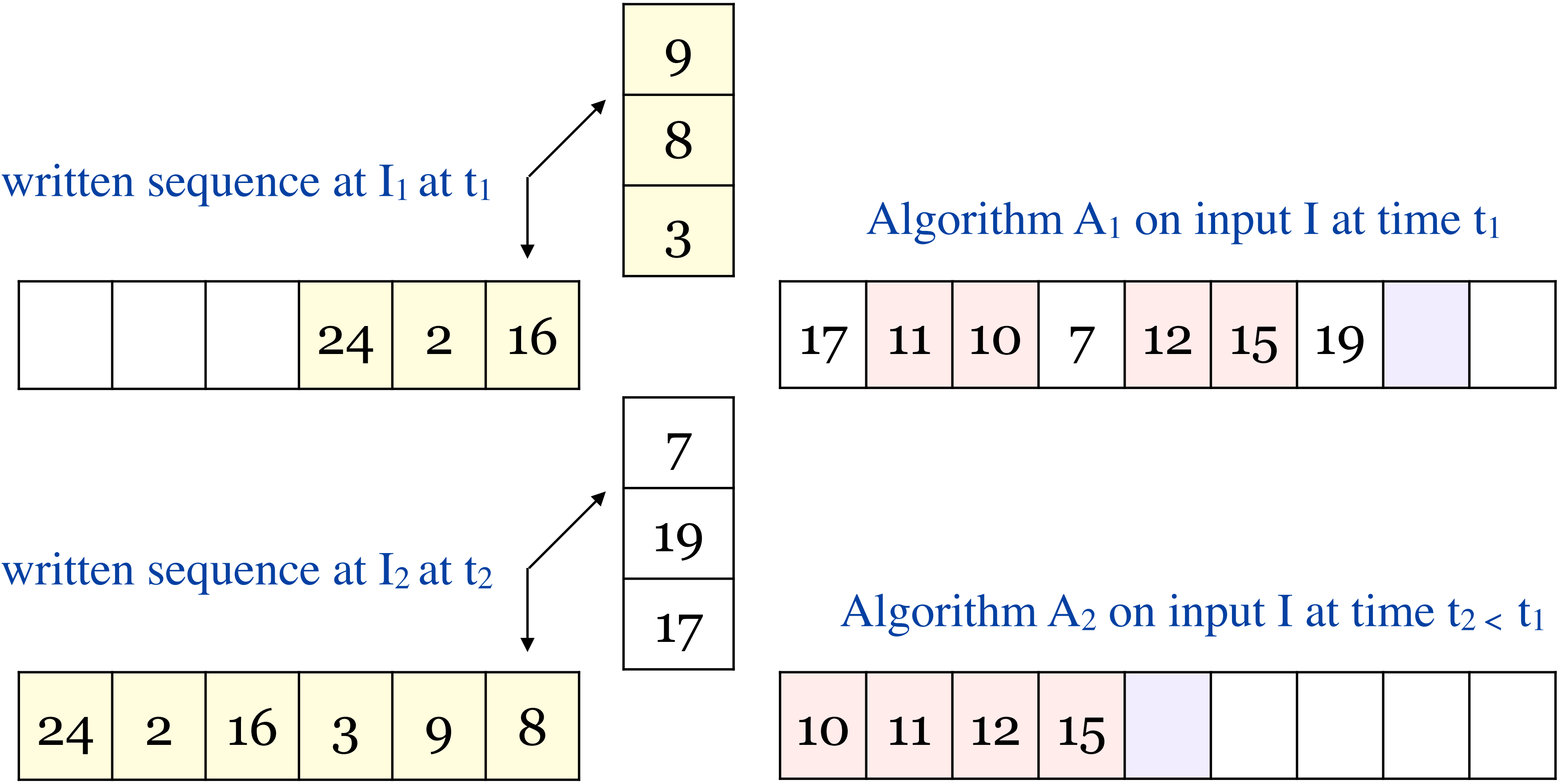
  Then A followed by an additional run *covers* B



Write A\ B's elements
using an extra run

Theorem: Alternating-Up-Down is a 2-Approx

- Writing extra elements never hurts - I_1 subsequence of I_2



Unwritten sequence at I_1 at t_1

9
8
3

			24	2	16

Algorithm A_1 on input I at time t_1

17	11	10	7	12	15	19		

7
19
17

Unwritten sequence at I_2 at t_2

24	2	16	3	9	8

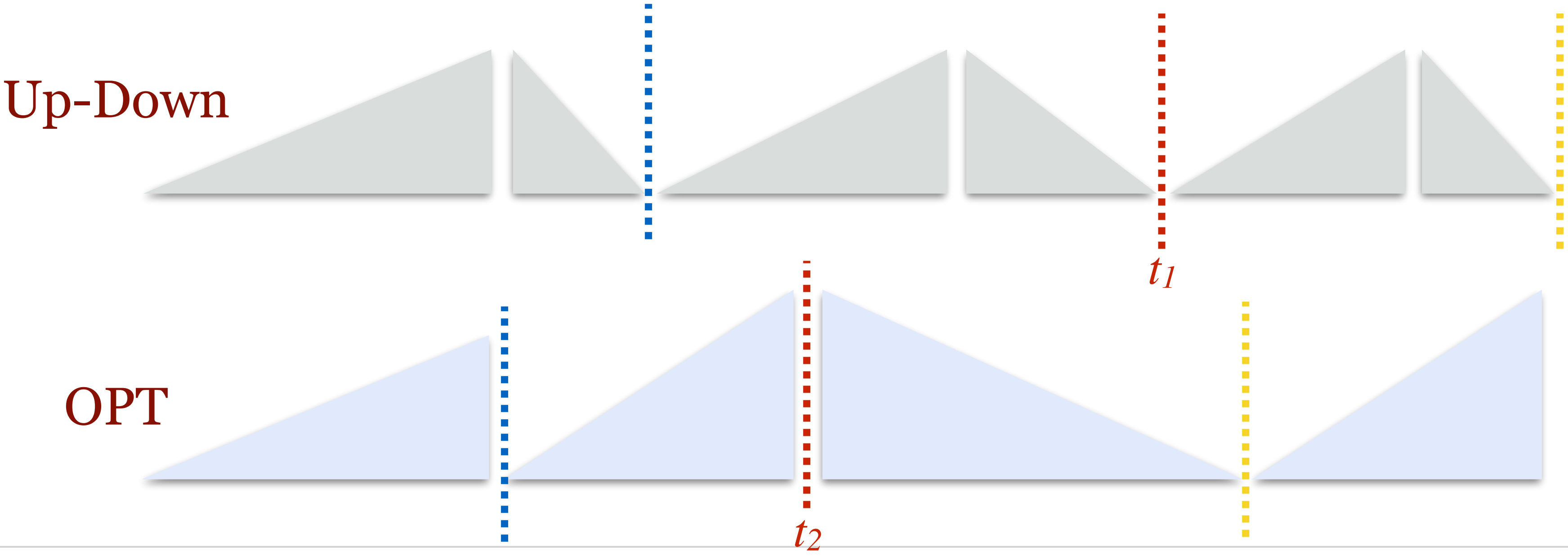
Algorithm A_2 on input I at time $t_2 < t_1$

10	11	12	15					

Theorem: Alternating-Up-Down is a 2-Approx

Proof Sketch

- At each decision point, suppose OPT goes up/down
  - A maximal up and down run goes at least as far
  - Every two runs cover at least one run of OPT



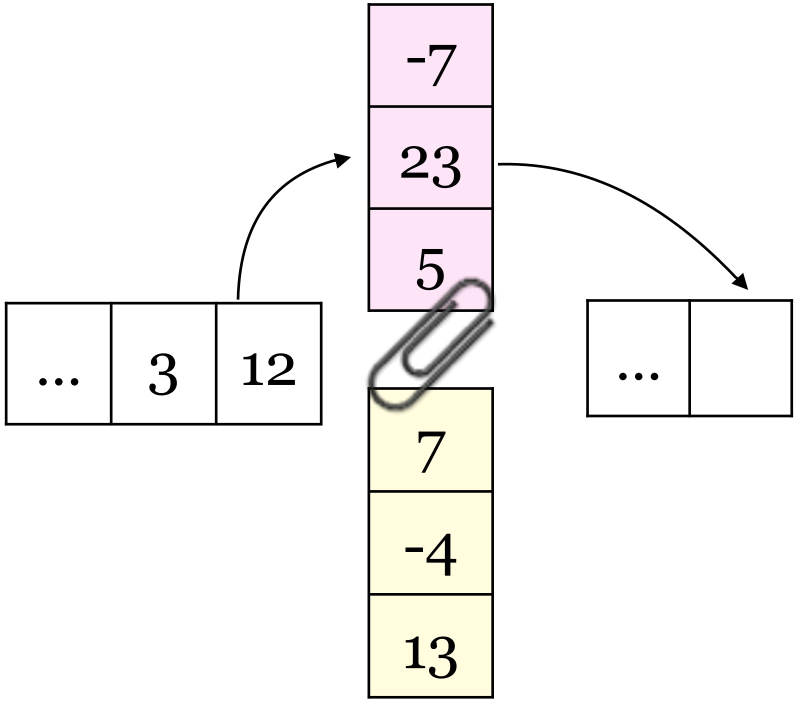
Lower Bounds

- No deterministic algorithm can do better than a 2-approx
  - Adversary switches the upcoming input wrt decision made
- No randomized algorithm can do better than a 1.5-approx
  - *Yao's minimax*

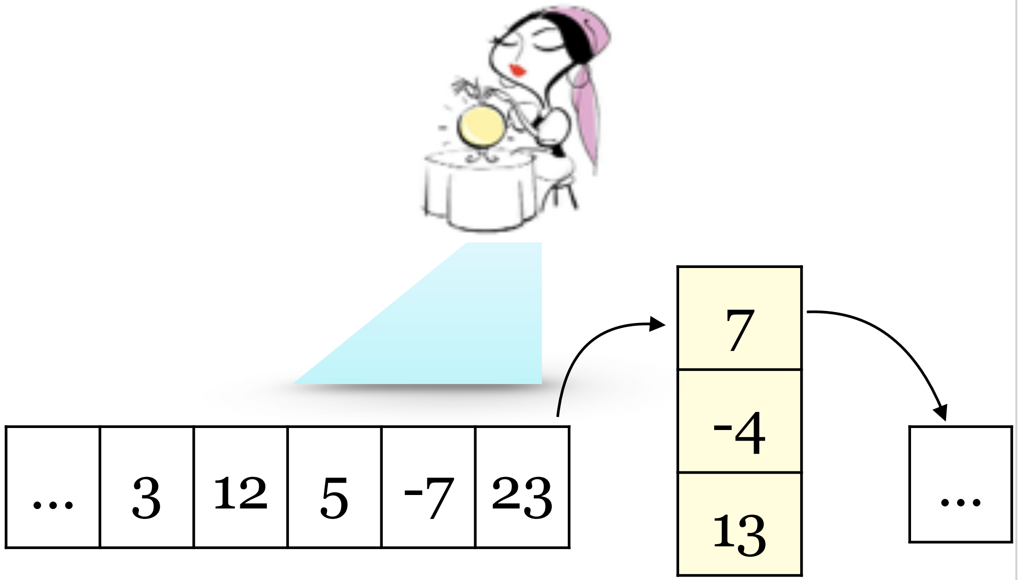
" Sh*t happens.."

Resource Augmentation

- No online algorithm can be better than a 2-approximation
  - *Can we do better with extra buffer or visibility?*

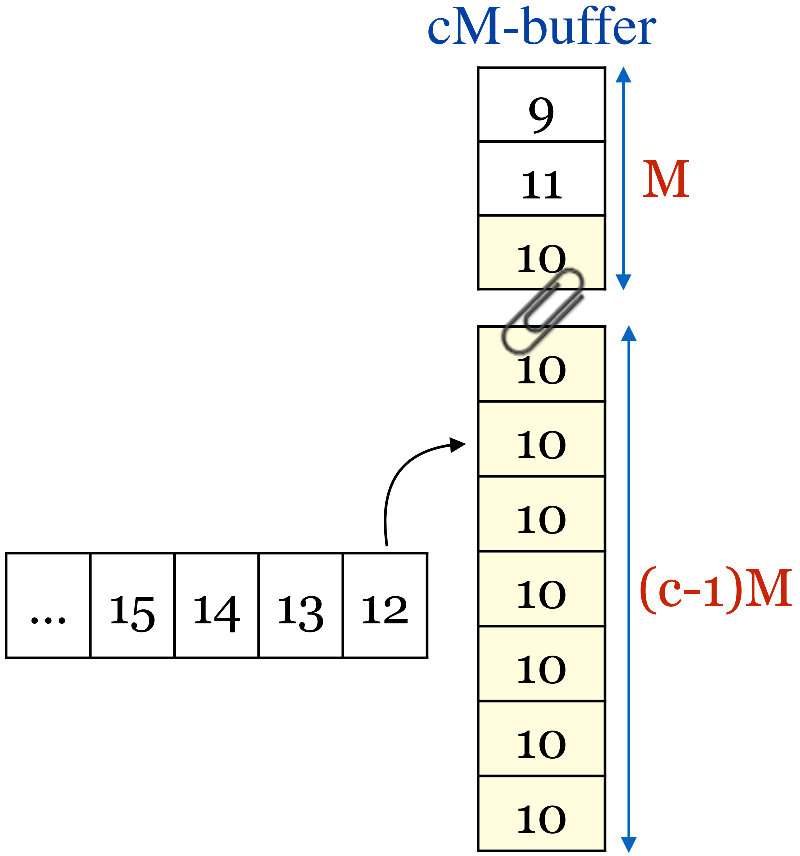

Extra buffer

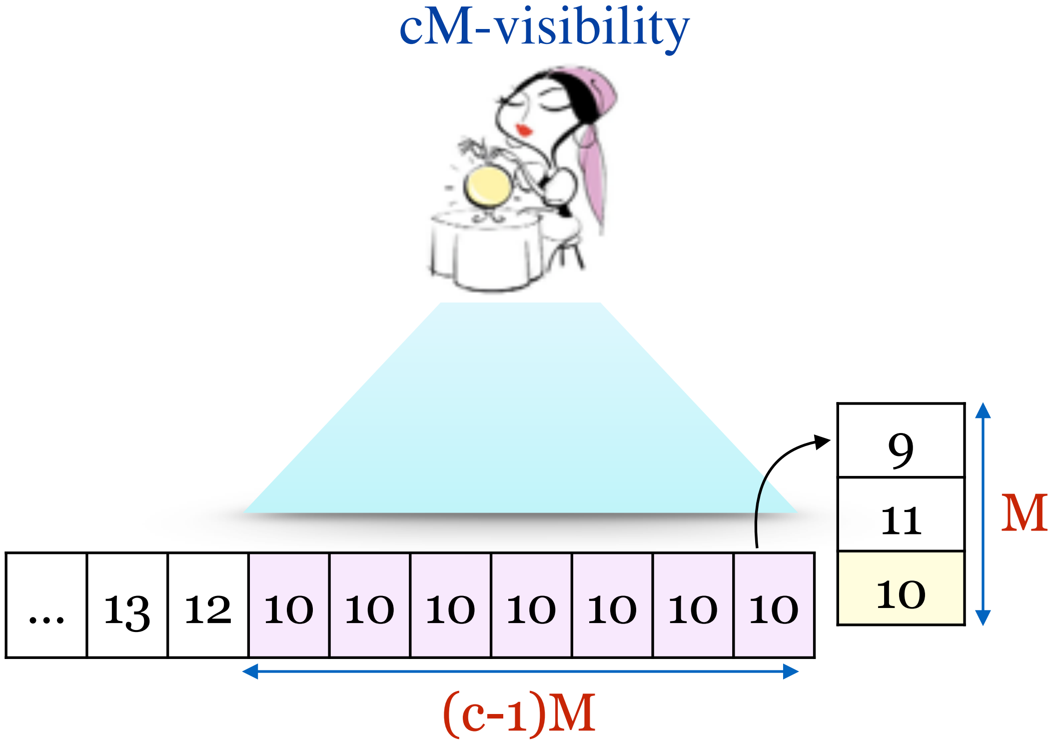
Regular buffer


Extra visibility

Resource Augmentation: No Duplicates

- Resource augmentation results require uniqueness
  - *Duplicates nullify extra buffer or visibility provided*



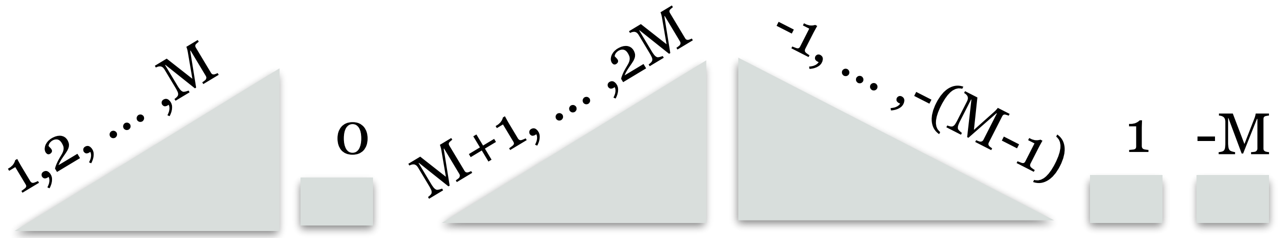


Main Idea Behind Resource Augmentation: What Would *Greedy* Do?

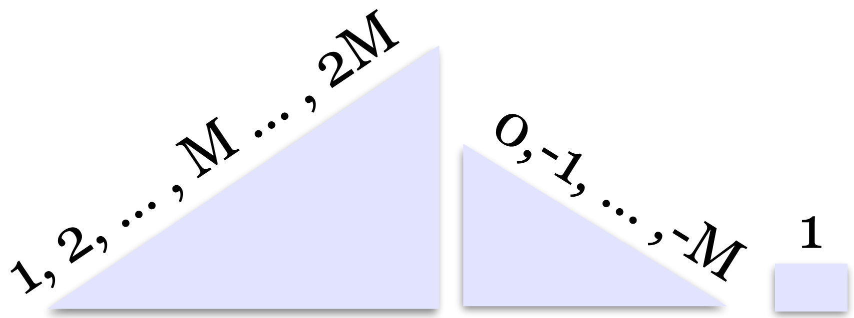
- Greedy chooses the longer run at every decision point
  - *Not* an online algorithm
- Greedy has some good guarantees
  - Upper bound and lower bound on run lengths

Note: Greedy is Not Optimal

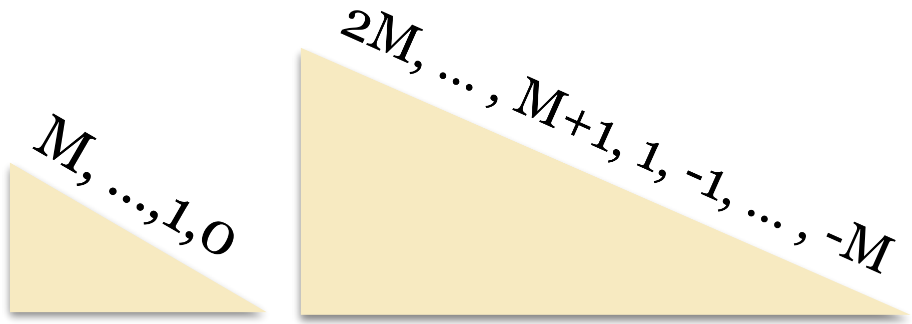
- Can be as bad as 1.5 times OPT



INPUT



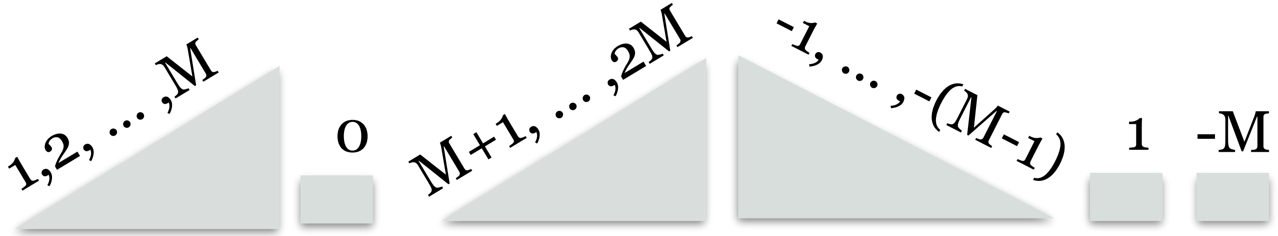
GREEDY



OPT

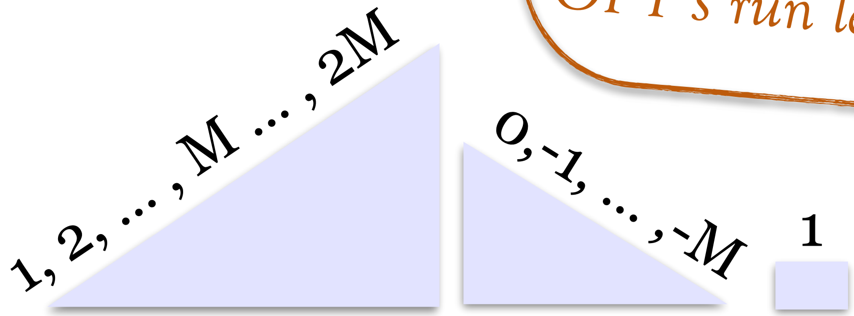
Note: Greedy is Not Optimal

- Can be as bad as 1.5 times OPT

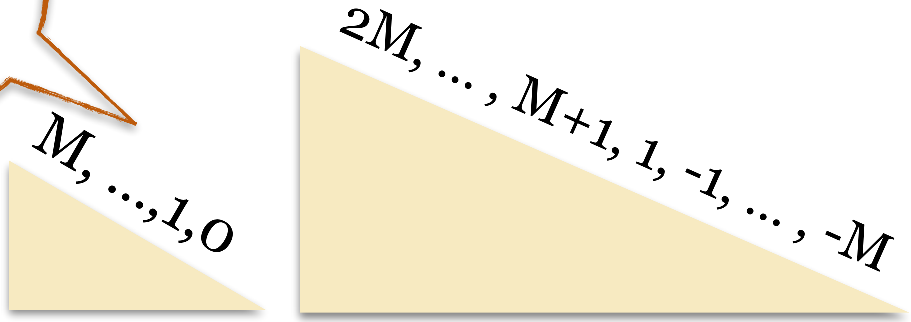


INPUT

No guarantee on OPT's run length



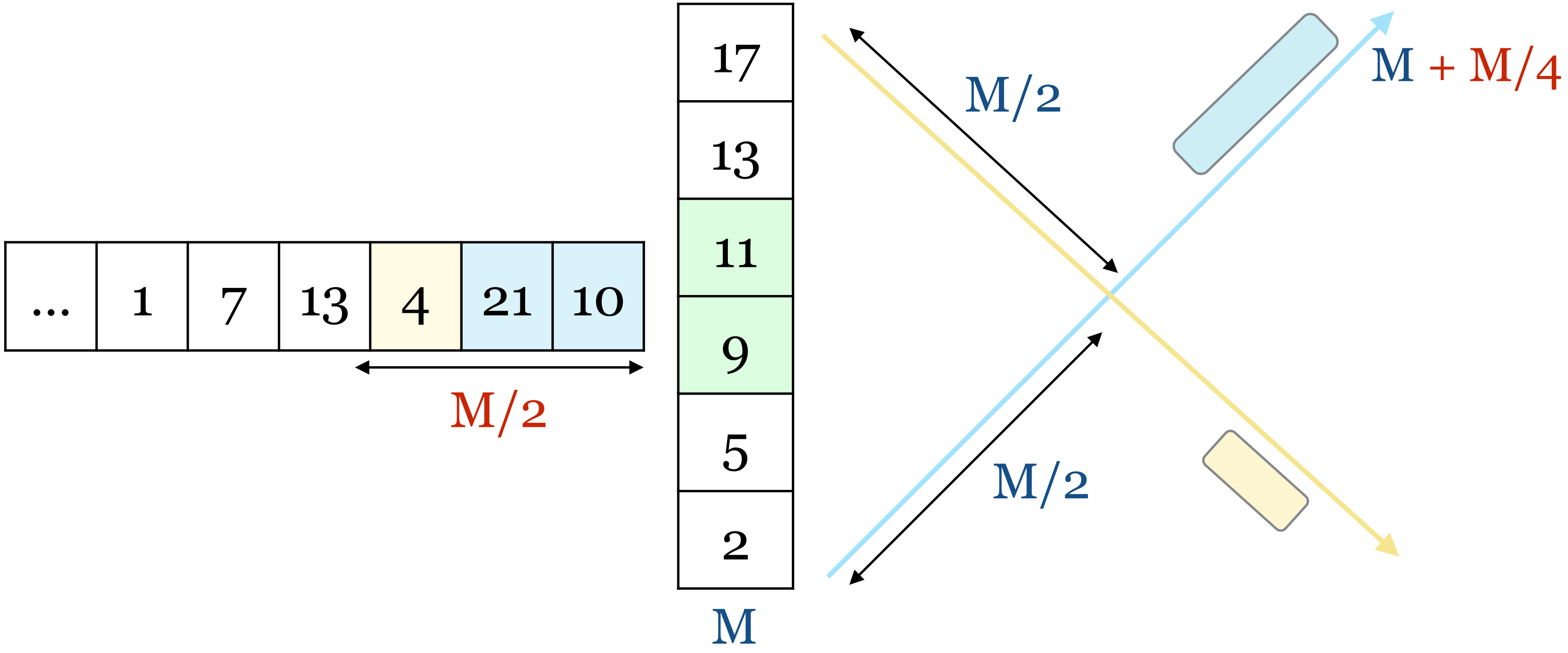
GREEDY



OPT

Guarantee on Greedy Runs

- Greedy has all (except last two) runs of length at least 1.25M
    - Consider elements arriving above and below the median



Greedy: How Long is the Not So Long Run?

Key Lemma

Given an input I with no duplicates, if the length of an initial run r_1 is greater than or equal to 3M, then the length of an initial run r_2 in the opposite direction is less than 3M.

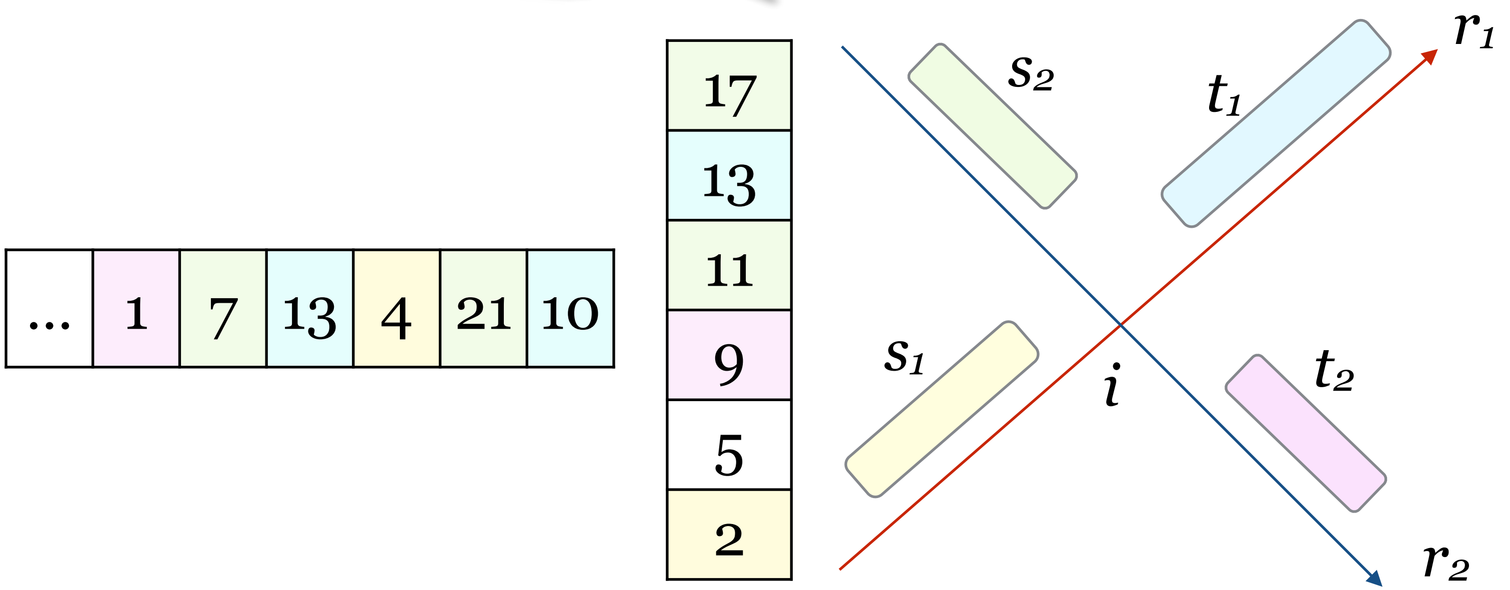
Take-away

- Don't have to look too far into the future to know greedy's choice

Sketchy Proof of Key Lemma

$s_1 \leq M$

s1 needs to fit in r2's buffer



Sketchy Proof of Key Lemma

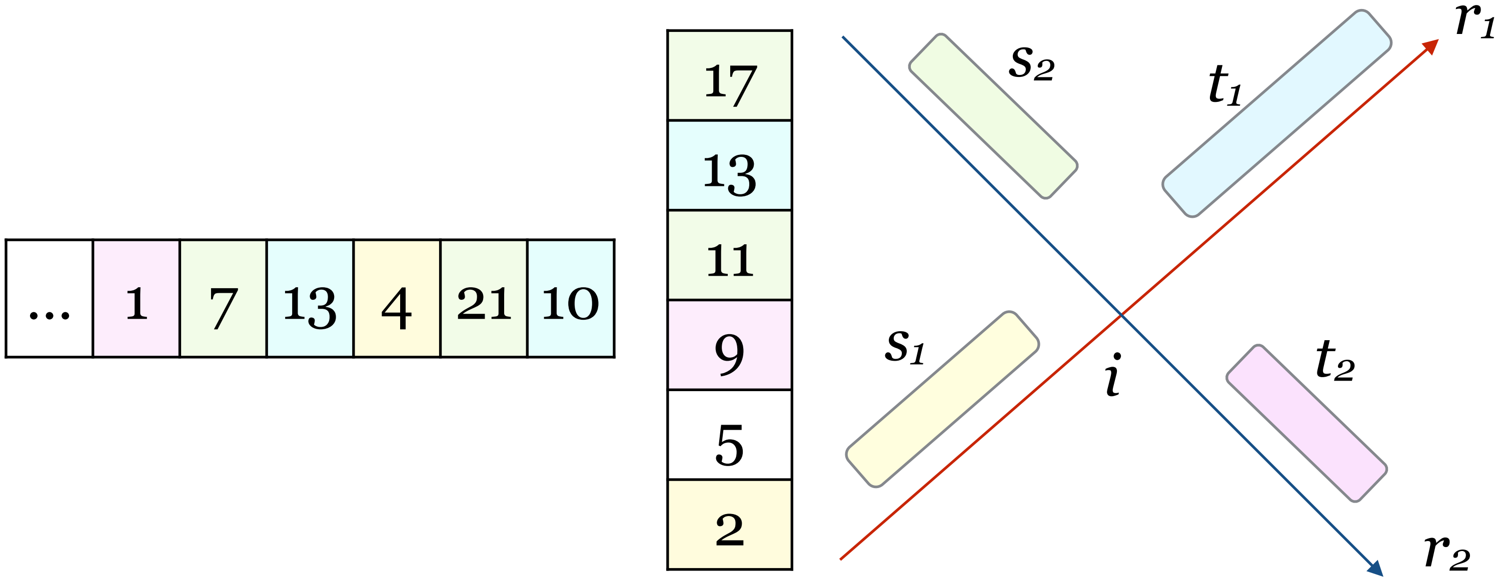
$s_1 \leq M$

$s_{2,N} + t_{1,B} \leq M$

$s_{2,N}$: Elements of s_2 not in initial buffer

$t_{1,B}$: Elements of t_1 in initial buffer

Both need to fit in r_1's buffer at i



Sketchy Proof of Key Lemma

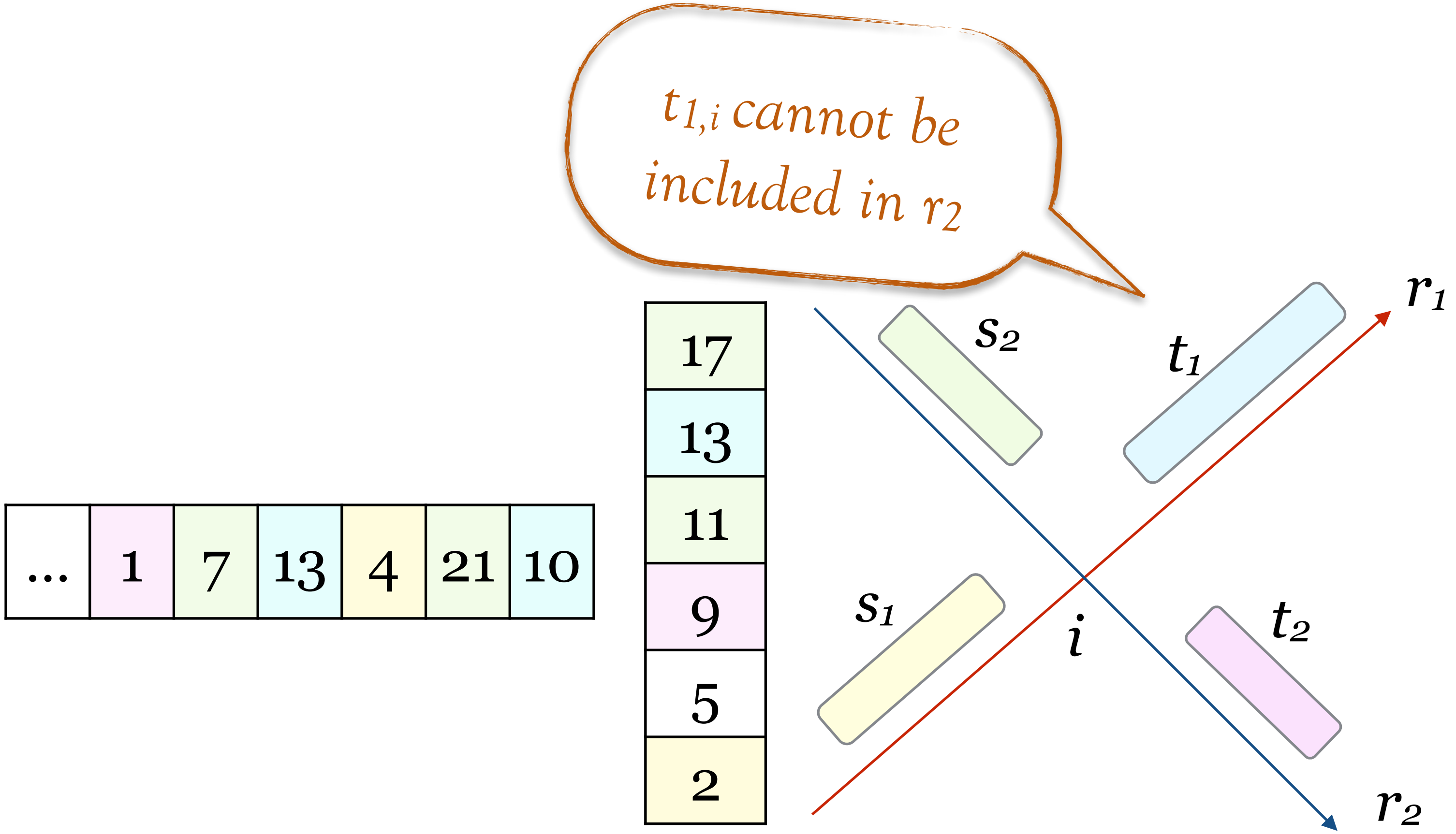
$s_1 \leq M$

$s_{2,N} + t_{1,B} \leq M$

$s_{2,N}$: Elements of s_2 not in initial buffer

$t_{1,B}$: Elements of t_1 in initial buffer

$t_{1,i}$: Elements in r_1 and read in after i



$t_{1,i}$ cannot be included in r_2

Sketchy Proof of Key Lemma

$s_1 \leq M$

$s_{2,N} + t_{1,B} \leq M$

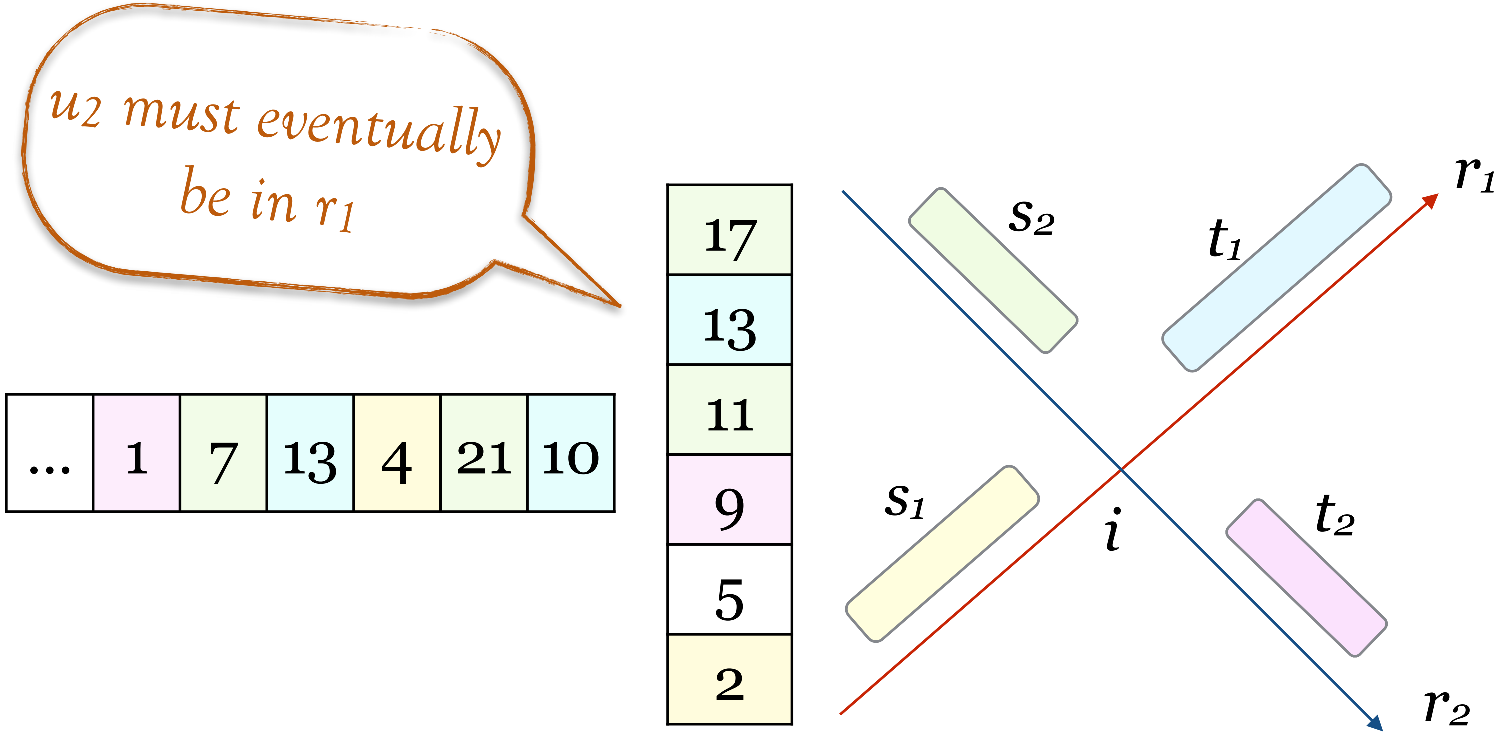
$u_2 \leq M$

$s_{2,N}$: Elements of s_2 not in initial buffer

$t_{1,B}$: Elements of t_1 in initial buffer

$t_{1,i}$: Elements in r_1 and read in after i

u_2 : Elements not in r_2 and read in before i



u_2 must eventually be in r_1

Sketchy Proof of Key Lemma

$s_1 \leq M$

$s_{2,N} + t_{1,B} \leq M$

$u_2 \leq M$

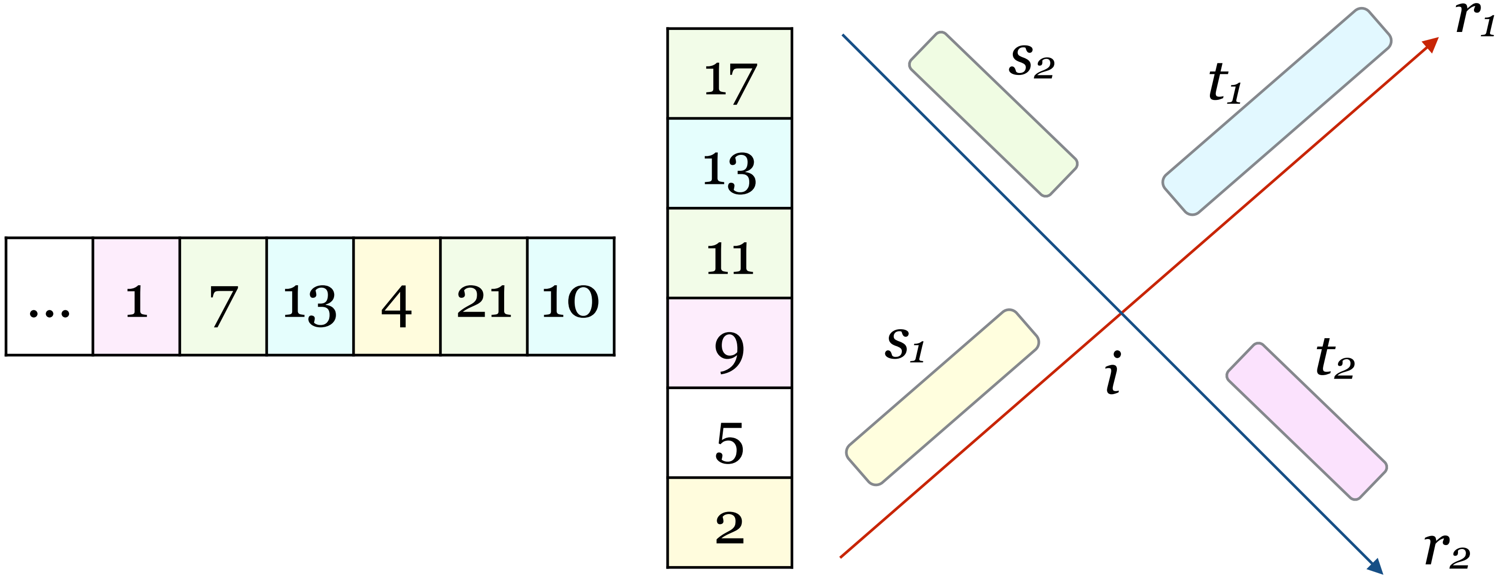
$s_{2,N}$: Elements of s_2 not in initial buffer

$t_{1,B}$: Elements of t_1 in initial buffer

$t_{1,i}$: Elements in r_1 and read in after i

u_2 : Elements not in r_2 and read in before i

$$\boxed{r_1 \leq s_1 + s_{2,N} + t_{1,B} + t_{1,i} + u_2}$$



Sketchy Proof of Key Lemma

$s_1 \leq M$

$s_{2,N} + t_{1,B} \leq M$

$u_2 \leq M$

$s_{2,N}$: Elements of s_2 not in initial buffer

$t_{1,B}$: Elements of t_1 in initial buffer

$t_{1,i}$: Elements in r_1 and read in after i

u_2 : Elements not in r_2 and read in before i

$$\boxed{r_1 \leq s_1 + s_{2,N} + t_{1,B} + t_{1,i} + u_2}$$

Weaker bound of 4M

$$\boxed{\text{If } r_1 \geq 4M \text{ then } t_{1,i} \geq M}$$

Sketchy Proof of Key Lemma

$s_1 \leq M$

$s_{2,N} + t_{1,B} \leq M$

$u_2 \leq M$

$s_{2,N}$: Elements of s_2 not in initial buffer

$t_{1,B}$: Elements of t_1 in initial buffer

$t_{1,i}$: Elements in r_1 and read in after i

u_2 : Elements not in r_2 and read in before i

$$\boxed{r_1 \leq s_1 + s_{2,N} + t_{1,B} + t_{1,i} + u_2}$$

Weaker bound of 4M

But $t_{1,i}$ needs to fit in r_2's buffer

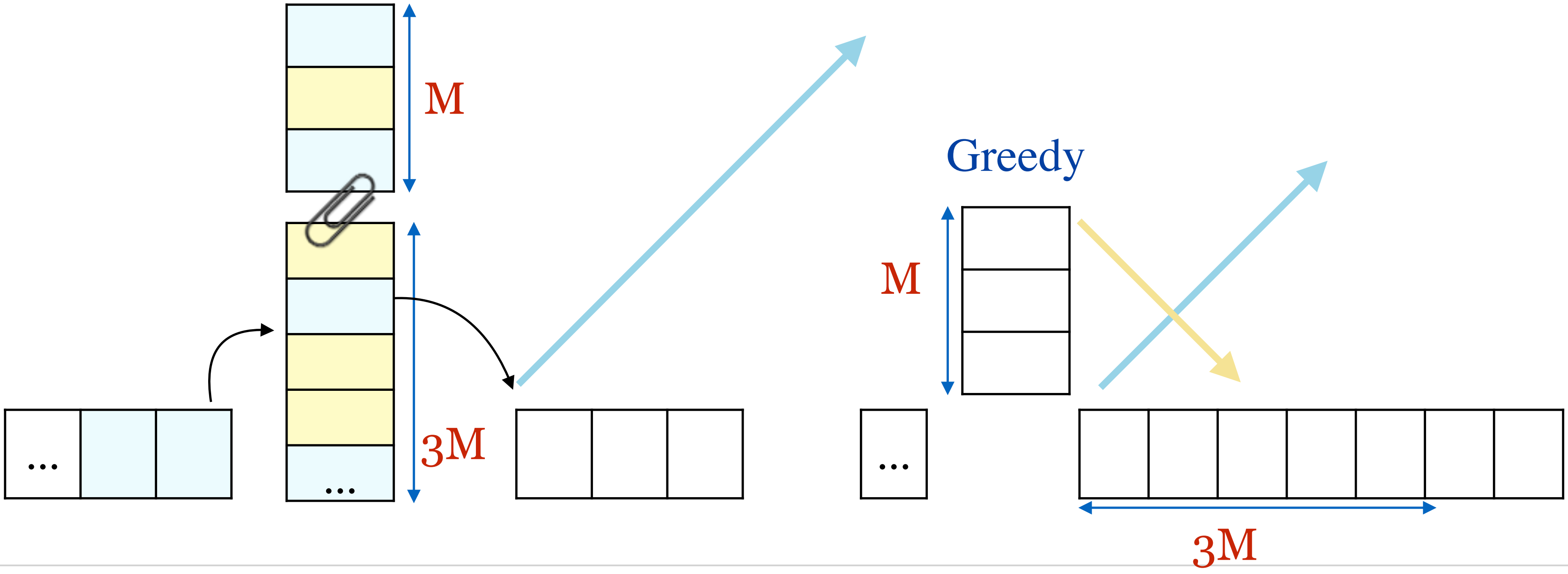
$$\boxed{\text{If } r_1 \geq 4M \text{ then } t_{1,i} \geq M}$$

$$\boxed{r_2 < 4M}$$

Theorem: Matching OPT with 4M buffer

Algorithm

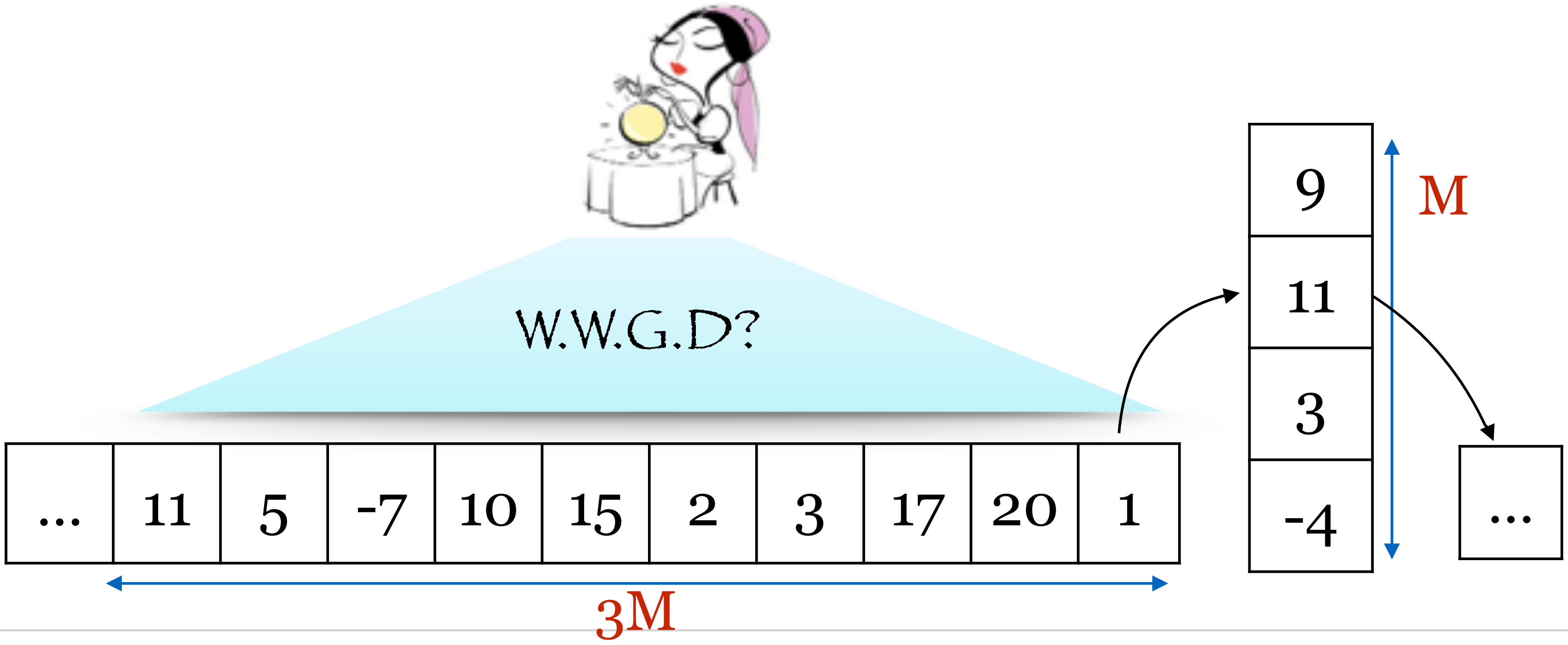
1. Read elements until entire buffer (4M) is full
2. Determine what greedy (with M buffer) would do
3. Write a maximal run in greedy's direction



Theorem: 1.5-Approximation with 4M-visibility

Algorithm

1. Determine what greedy (with M buffer) would do
2. Write a maximal run in greedy's direction
3. Write two more - in the same and opposite direction



Theorem: 1.5-Approximation with 4M-visibility

Algorithm

1. Determine what greedy (with M buffer) would do
2. Write a maximal run in greedy's direction
3. Write two more - in the same and opposite direction

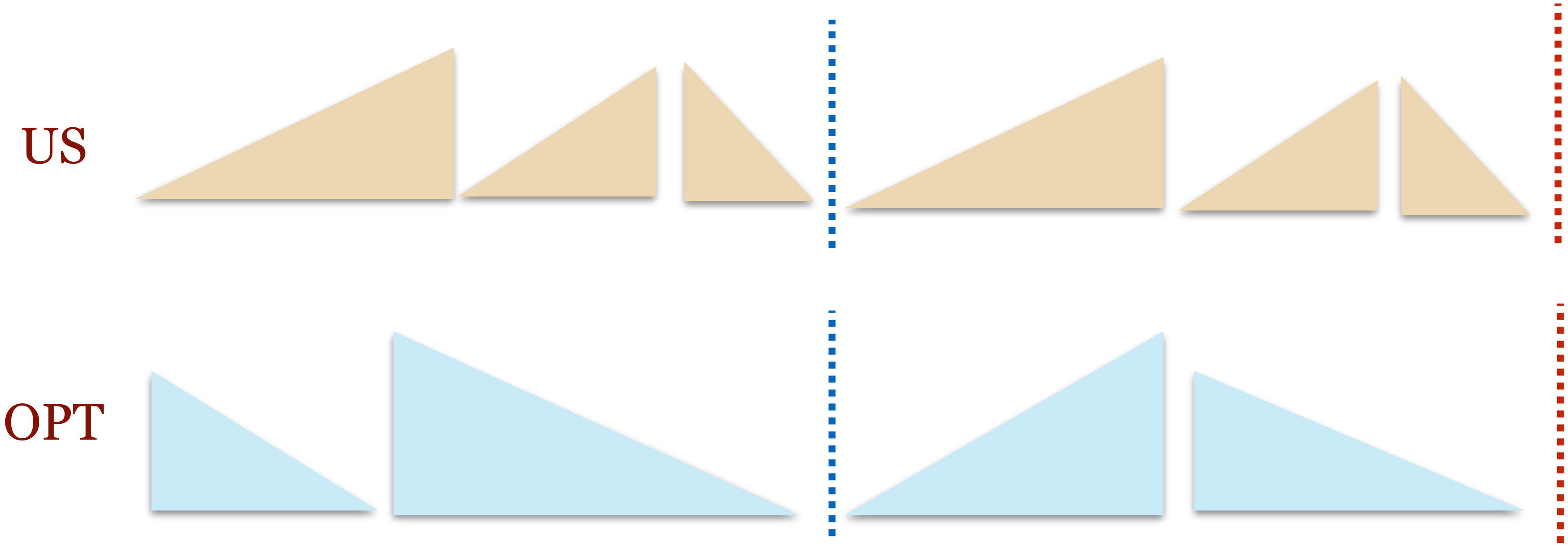
Lemma

At any decision point, if OPT chooses a non-greedy run (say down), it's next run must be in the same direction (down).

Theorem: 1.5-Approximation with 4M-visibility

Algorithm

1. Determine what greedy (with M buffer) would do
2. Write a maximal run in greedy’s direction
3. Write two more - in the same and opposite direction



Lower Bound on Resource Augmentation

Almost tight

- With a buffer of size 4M-2
    - No deterministic algorithm can do better than 1.5-approx
- Above lower bound implies lower bound for 4M-2 visibility

Offline Run Generation Problem

- An offline algorithm knows the entire input in advance
    - *Algorithm with N-visibility*
- Polynomial time offline optimal algorithm? - *still open!!*

“ My ~~Momma~~ Michael was so sure that dynamic programming would be great....”

Run Generation on Nearly-Sorted Input

Definition

An input is c-nearly sorted if there exists an optimal algorithm whose output consists of runs of length at least cM.

Other Results

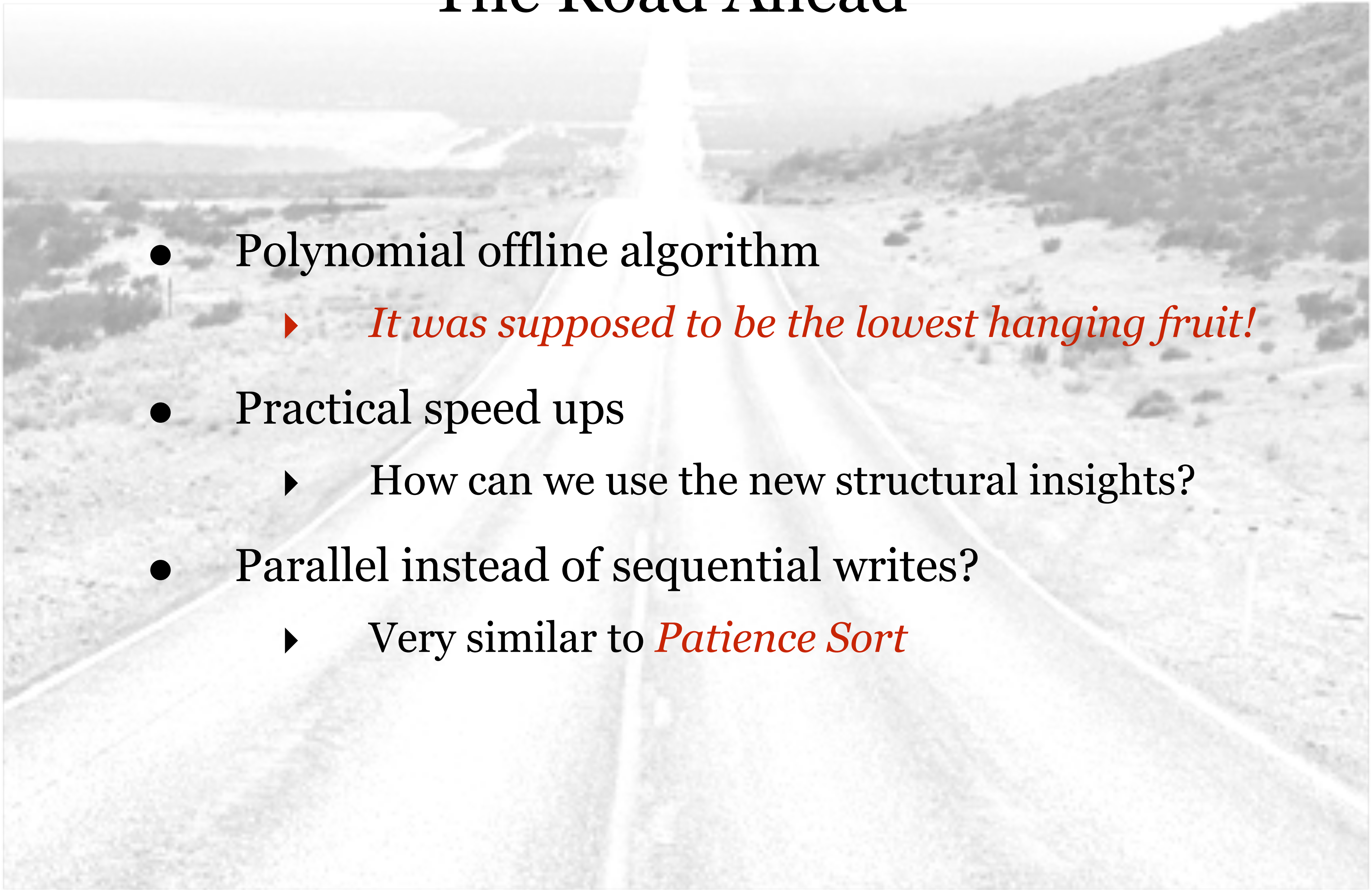
- Randomized 1.5-approx with 2M-buffer on 3-nearly sorted
- Greedy offline algorithm on 5-nearly sorted is optimal

Summary of Our Results

Approximation Factor	Buffer Size	Visibility	Online	Nearly Sorted
2	M	M	Yes	-
1.5	M	4M	Yes	-
1	4M	4M	Yes	-
$(1+\varepsilon)$	M	N	No	-
1.5	2M	2M	Yes	3M
1	M	N	No	5M

“Run Generation is *not a* box of chocolates.”

The Road Ahead



- Polynomial offline algorithm
  - *It was supposed to be the lowest hanging fruit!*
- Practical speed ups
  - How can we use the new structural insights?
- Parallel instead of sequential writes?
  - Very similar to *Patience Sort*

A Shout Out to the Team!

" And that's all I have to say about that.."