Chapter 8: Online Algorithms

Advanced Algorithms

Online vs. offline | Competitive analysis | Ski rental | Paging: FIFO, LRU, OPT | List update | Data stream algorithms | Reservoir sampling | Incremental learning

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/advanced_algorithms/slides/Chapter_08_Online_Algorithms.html

Learning Outcomes

Model

Distinguish the online and offline problem models; explain what an irrevocable decision is.

Competitive analysis

Define the competitive ratio; prove the 2 − 1/B ratio for the ski rental problem.

Paging and lists

Count page faults step by step under FIFO, LRU and OPT; interpret why LRU is k-competitive and MTF is 2-competitive.

Data streams

Implement a moving average, a running median and reservoir sampling in Java; analyze time and memory per element.

Deciding Without the Full Picture

- Sometimes decisions must be made **without all the information**.
- **Online algorithms**:
  - when information arrives **piece by piece**,
  - they try to make the **best decision** with what is known so far.
- A decision is **irrevocable**: information that arrives later does not change past decisions.

A chess player moves without knowing the opponent's next move; once made, a move cannot be taken back.



Online Algorithms

- Like walking through a **maze**:
  - You cannot see the whole maze, only the path ahead.
- At each step you pick the best path with **limited information**.
- They allow **fast decisions** when information does not arrive **all at once**.
- Examples:
  - Which road to take in a traffic jam?
  - What is the next move in a chess game?
  - Adjusting traffic lights to current traffic.



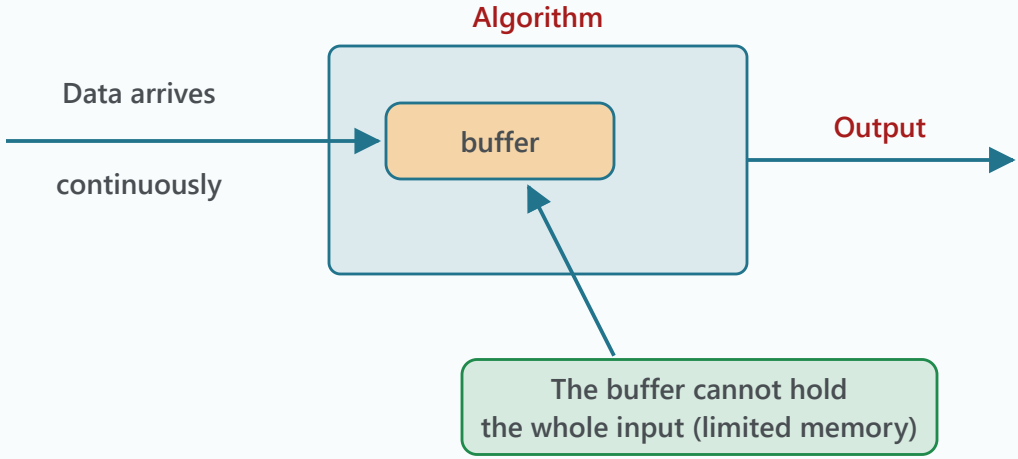
Online vs. Offline Problem Model

The input is a **request sequence** $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$. The difference lies in **when** the algorithm sees it.

	Offline	Online
Input	Fully known in advance	Arrives piece by piece; when σ_t arrives, σ_{t+1} is unknown
Decision	Made after seeing the whole input	Must be made when σ_t arrives; **irrevocable**
Example	Sorting, Dijkstra, knapsack (DP)	Cache management, ski rental, stock trading
Optimal solution	OPT can be computed	OPT is known only in hindsight
Measure	Time / space complexity	**Competitive ratio**: ALG / OPT

"Online" here does not mean *the internet*; it means **the input is revealed over time**. Insertion sort can run online; selection sort cannot.

The Online Algorithm Model



- Data arrives **continuously**; the algorithm processes each piece as it arrives and outputs immediately.
- The memory (buffer) **cannot hold the whole input**: the past survives only as a summary (a sum, a heap, a sample…).
- So the goal is **little work** per element (usually O(1) or O(log n)) and **little memory**.

Applications

Stock market
Buy or sell at the current price; future prices are unknown.

Caching
CPU, operating system, web: which page stays in memory?

Routing
Shortest paths in a network whose links keep changing.

Resource allocation
Dynamic sharing of bandwidth and CPU time.

Scheduling
Jobs are assigned to machines as they arrive.

Ads and recommendations
Which ad / product to show when a user arrives?

Machine learning
The model is updated with every new example (incremental learning).

Security
Detecting fraud and DDoS attacks on the stream.

Autonomous vehicles
Real-time decisions as sensor data arrives.

Common ground: the environment is **dynamic**, decisions are **real-time**, and knowledge of the future is **incomplete**.

Advantages and Disadvantages

Advantages

+ Work in **dynamic** environments; no need to wait for the input to end.
+ Make **real-time** decisions with low latency.
+ Can process very large, even **infinite**, streams with little memory.
+ The competitive ratio gives a **worst-case guarantee**.

Disadvantages

– **Wrong decisions** in early steps can affect later ones.
– In chess, one wrong move can lose the game.
– No guarantee of finding the **best global solution**.
– Sensitive to the early input; harder to design and analyze.

Design Approaches for Online Algorithms

Greedy

Handles each arrival in the way that looks best right now, ignoring global effects. Example: assign each request to the nearest idle server.

Summarizing

Keeps a small summary instead of the whole past: a sum, two heaps, a sample. Example: moving average, running median.

Adaptive

Learns from mistakes and changes its behavior over time. Example: LRU, move-to-front (MTF), incremental learning.

Deterministic

Always makes the same decision on the same input; a malicious input sequence (an adversary) can easily trap it.

Randomized

Adds chance to its decisions, so the adversary cannot predict them. Example: reservoir sampling, randomized marking.

Competitive Analysis

- On every input, the online algorithm ALG is compared with the best offline algorithm OPT, which **knows everything in advance**.
- ALG is **c-competitive** if for every request sequence σ

$$\mathrm{ALG}(\sigma) \le c \cdot \mathrm{OPT}(\sigma) + \alpha$$

  holds (α: a constant independent of σ; if α = 0, it is **strictly** competitive).
- The smallest such c is the **competitive ratio**: $c = \sup_{\sigma} \mathrm{ALG}(\sigma) / \mathrm{OPT}(\sigma)$.
- The **closer the ratio is to 1**, the less the missing knowledge of the future hurts.

Pessimistic view

The analysis is over the **worst input**: no averaging and no probabilistic assumptions.

Cost problems

Here cost is minimized ($c \ge 1$). For profit problems the ratio is defined as OPT / ALG.

COMPETITIVE ANALYSIS

The Adversary Model

To **lower-bound** the competitive ratio, imagine an adversary playing against the algorithm: it generates the input and runs OPT.

Deterministic ALG

The adversary knows the algorithm's code; at each step it issues the request that **hurts ALG the most**.

Oblivious adversary

Cannot see a randomized algorithm's coin flips; it fixes the sequence in advance. We measure ALG's expected cost.

Adaptive adversary

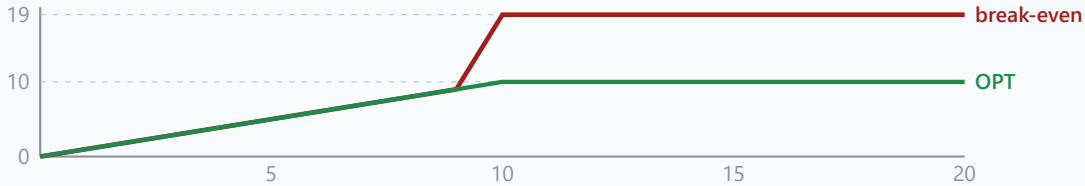
Sees ALG's random choices before picking the next request; randomness helps less.

Proof pattern: to show that "no deterministic algorithm can beat c", build for every ALG a sequence σ with $\mathrm{ALG}(\sigma) / \mathrm{OPT}(\sigma) \geq c$.

SKI RENTAL

The Ski Rental Problem

- Skis can be **rented for 1 TL per day** or **bought for B TL**.
- The number of ski days d is **unknown**: each morning you learn only whether you ski today.
- Each day's decision: **rent again** or **buy**?
- OPT knows d: **OPT(d) = min(d, B)**.



B = 10; x-axis: actual days d. Worst ratio at d = B: 19/10 = 2 − 1/B.

Always rent

As d → ∞, cost d vs. OPT = B: ratio **unbounded**.

Buy on day 1

With d = 1, cost B vs. OPT = 1: ratio **B**.

Break-even

Rent B − 1 days, buy on day B: ratio **2 − 1/B**.

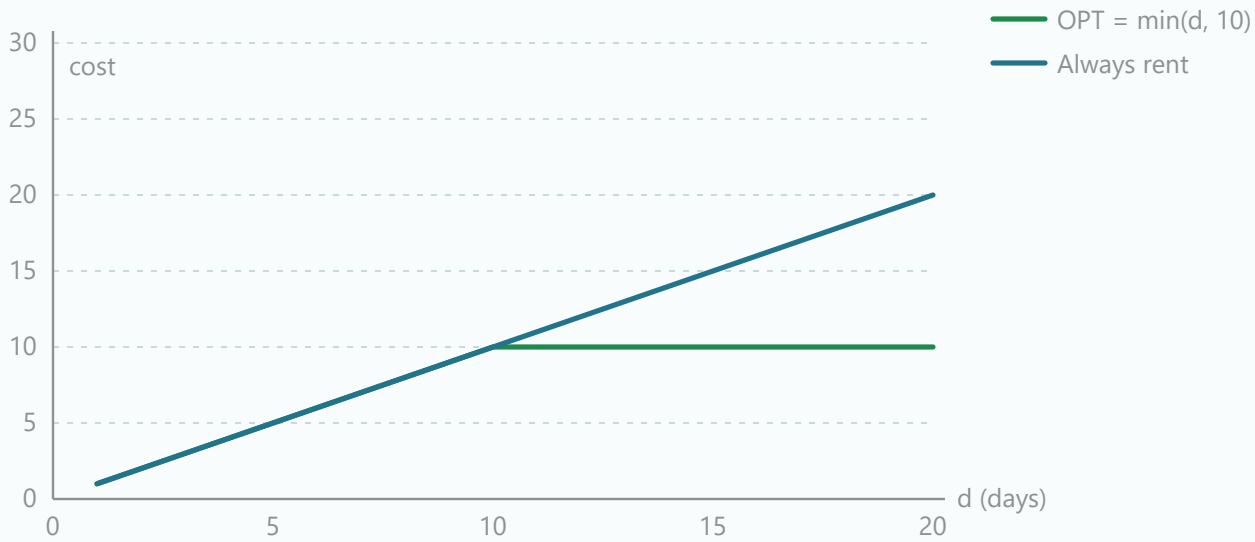
Same shape: hourly cloud server vs. a yearly plan; keeping a TCP connection open vs. closing it.

Cost of the Strategies (B = 10)



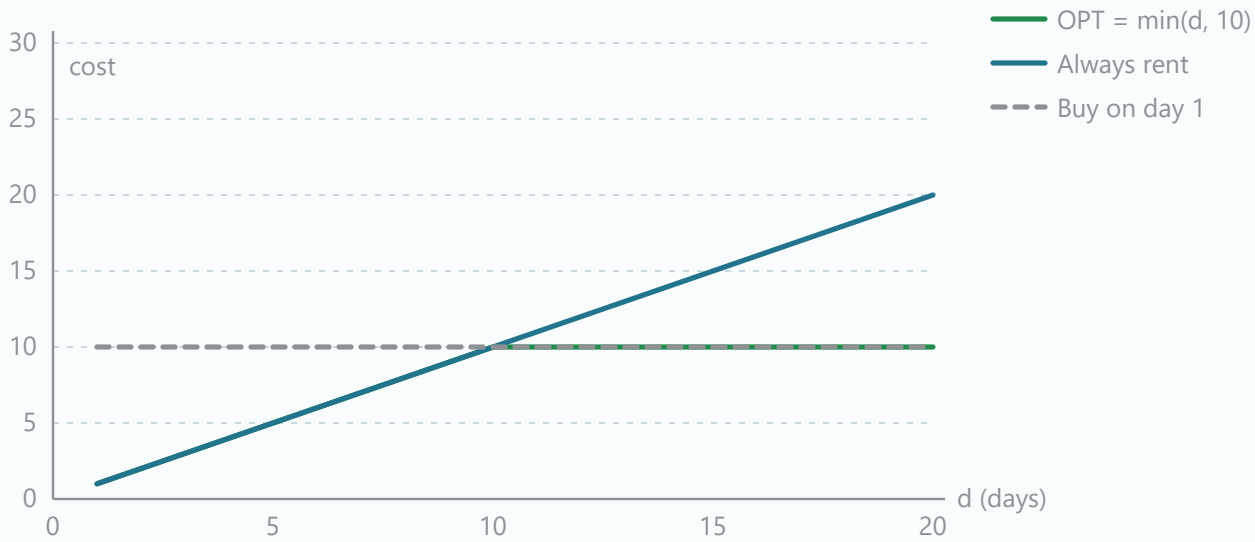
OPT, knowing the future: if d < 10 it always rents (cost d); if d ≥ 10 it buys on day 1 (cost 10).

Cost of the Strategies (B = 10)



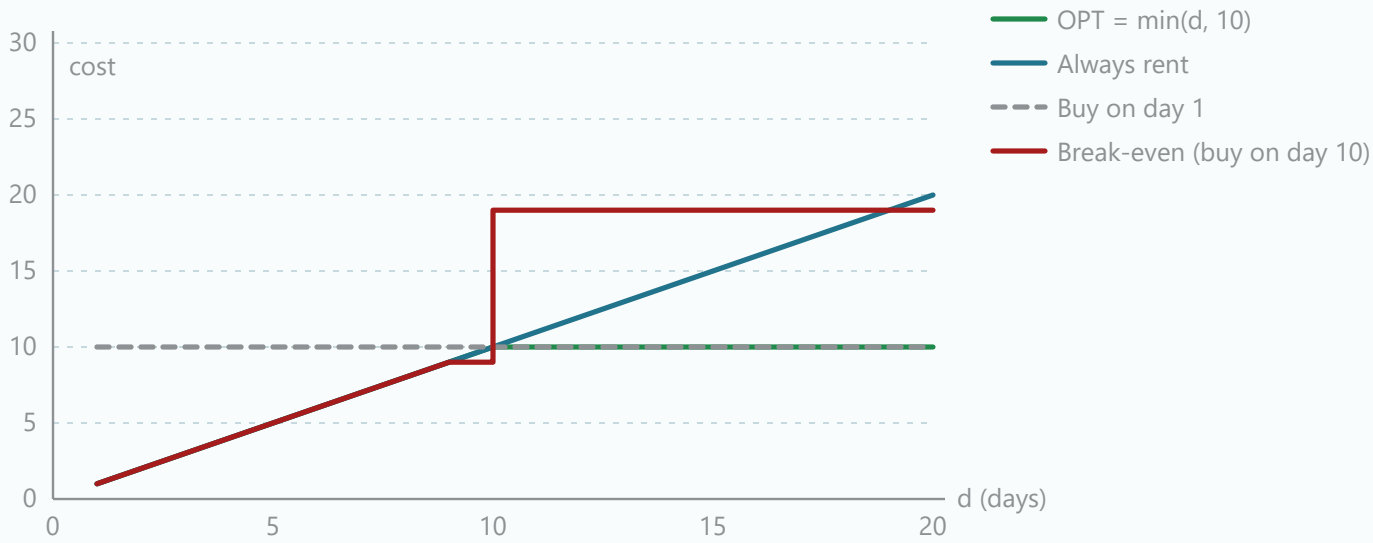
Always rent: cost d. As d grows the cost increases without bound while OPT stays at 10 → **not** competitive.

Cost of the Strategies (B = 10)



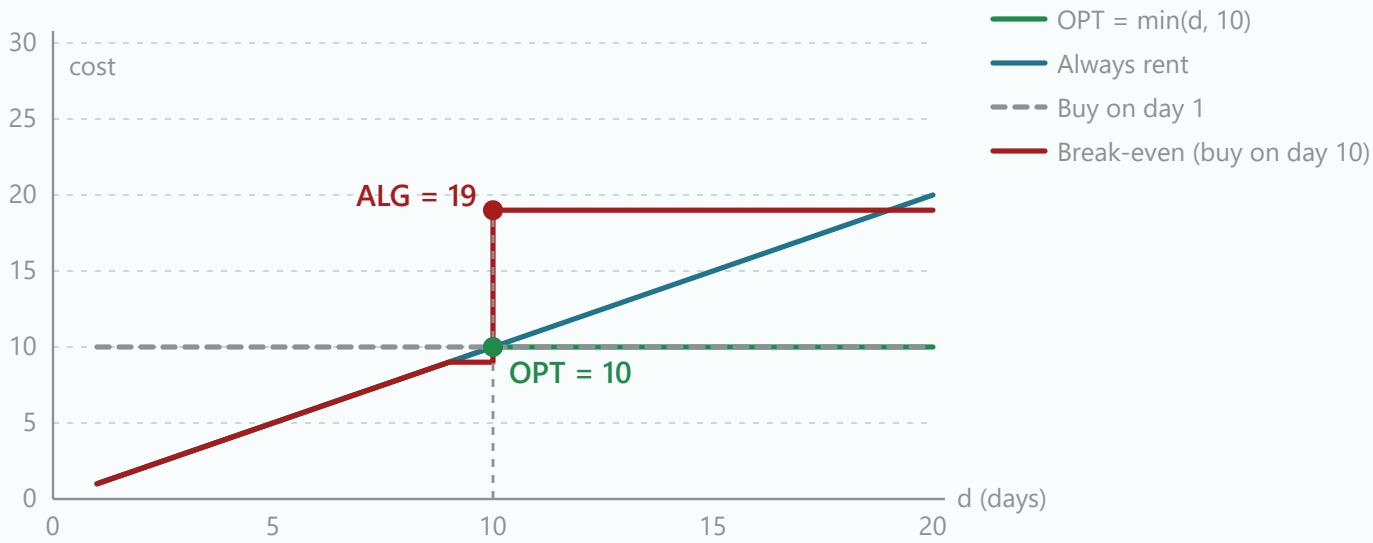
Buy on day 1: the cost is always 10. But if d = 1, OPT = 1 → ratio 10 = B. Bad.

Cost of the Strategies (B = 10)



Break-even: rent for 9 days, buy on day 10. While $d < 10$ it matches OPT; for $d \geq 10$ the cost is $9 + 10 = 19$.

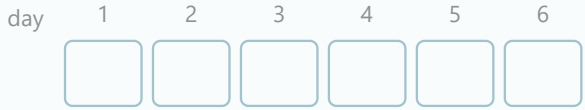
Cost of the Strategies (B = 10)



Worst case d = 10: ALG = 19, OPT = 10 → ratio **1.9 = 2 − 1/B**. No d is worse.

Break-Even Strategy: Code Trace

```
1  int breakEven(int B, int d) {  // d unknown
2    int cost = 0;
3    for (int day = 1; day <= d; day++) {
4      if (day < B) {
5        cost += 1;                // rent
6      } else {
7        cost += B;                // buy
8        break;                    // no more fees
9      }
10   }
11   return cost;
12 }
```



ALG = 0

B = 4 TL (to buy), rent 1 TL per day. The season will actually last d = 6 days; the algorithm does not know this and each morning sees only "am I skiing today?".

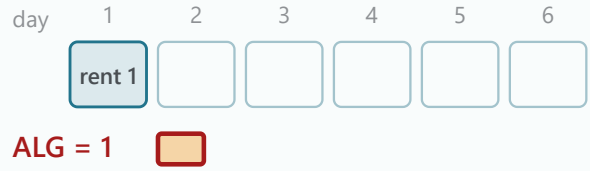
Step 1 / 7

VARIABLES

`B = 4` `day = –` `cost = 0`

Break-Even Strategy: Code Trace

```
1  int breakEven(int B, int d) {  // d unknown
2    int cost = 0;
▶    for (int day = 1; day <= d; day++) {
▶      if (day < B) {
▶        cost += 1;                  // rent
6      } else {
7        cost += B;                  // buy
8        break;                      // no more fees
9      }
10   }
11   return cost;
12 }
```



Day 1: 1 < B → rent. cost = 1.

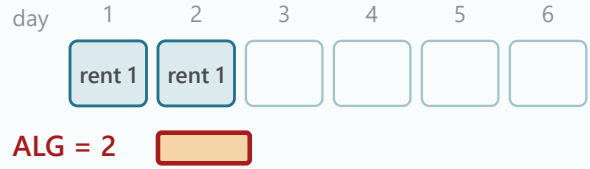
Step 2 / 7

VARIABLES

B = 4 | day = 1 | cost = 1

Break-Even Strategy: Code Trace

```
1  int breakEven(int B, int d) {  // d unknown
2    int cost = 0;
▶    for (int day = 1; day <= d; day++) {
▶      if (day < B) {
▶        cost += 1;                  // rent
6      } else {
7        cost += B;                  // buy
8        break;                      // no more fees
9      }
10   }
11   return cost;
12 }
```



Day 2: 2 < B → rent. cost = 2.

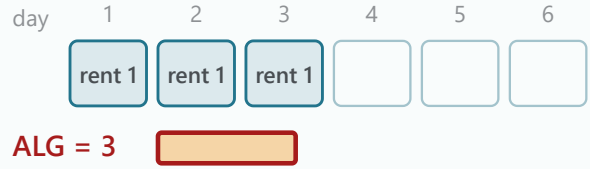
Step 3 / 7

VARIABLES

B = 4 | day = 2 | cost = 2

Break-Even Strategy: Code Trace

```
1  int breakEven(int B, int d) {  // d unknown
2    int cost = 0;
▶    for (int day = 1; day <= d; day++) {
▶      if (day < B) {
▶        cost += 1;                  // rent
6      } else {
7        cost += B;                  // buy
8        break;                      // no more fees
9      }
10   }
11   return cost;
12 }
```



Day 3: 3 < B → rent. cost = 3.

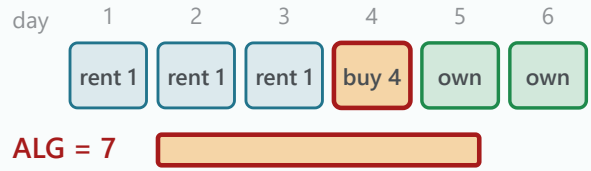
Step 4 / 7

VARIABLES

B = 4 | day = 3 | cost = 3

Break-Even Strategy: Code Trace

```
1  int breakEven(int B, int d) {  // d unknown
2    int cost = 0;
3    for (int day = 1; day <= d; day++) {
4      if (day < B) {
5        cost += 1;                  // rent
▶      } else {
▶        cost += B;                  // buy
8        break;                      // no more fees
9      }
10   }
11   return cost;
12 }
```



Day 4 = B: rent so far is B − 1 = 3 TL; now **buy**: cost = 3 + 4 = 7.

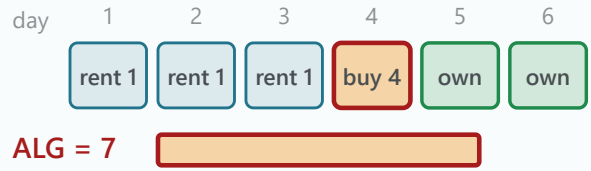
Step 5 / 7

VARIABLES

B = 4 | day = 4 | cost = 7

Break-Even Strategy: Code Trace

```
1  int breakEven(int B, int d) {  // d unknown
2    int cost = 0;
3    for (int day = 1; day <= d; day++) {
4      if (day < B) {
5        cost += 1;                // rent
6      } else {
7        cost += B;                // buy
▶        break;                   // no more fees
9      }
10   }
11   return cost;
12 }
```



`break` : the skis are ours now; days 5 and 6 are free.

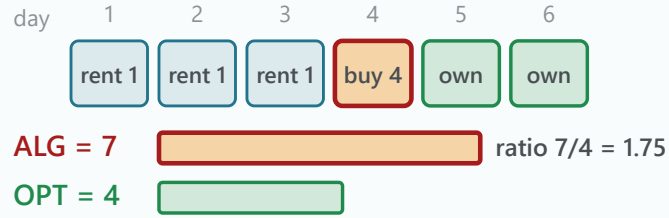
Step 6 / 7

VARIABLES

`B = 4` `day = 4` `cost = 7`

Break-Even Strategy: Code Trace

```
1  int breakEven(int B, int d) {  // d unknown
2    int cost = 0;
3    for (int day = 1; day <= d; day++) {
4      if (day < B) {
5        cost += 1;                // rent
6      } else {
7        cost += B;                // buy
8        break;                    // no more fees
9      }
10   }
▶    return cost;
12 }
```



ALG = 7, OPT = min(6, 4) = 4 (buy on day 1). Ratio 7/4 = 1.75 = 2 − 1/B. Whatever d is, the ratio never exceeds this.

Step 7 / 7

VARIABLES

`cost = 7` `OPT = min(d, B)=4`

Why Nothing Does Better: Lower Bound

Every deterministic strategy is described by a single number: **k** = the day it buys.

- **Adversary**: the weather turns right after the purchase, i.e., d = k.
- ALG = (k − 1) rent + B, OPT = min(k, B).
- k < B: ratio (k − 1 + B) / k ≥ (2B − 1) / B.
- k ≥ B: ratio (k − 1 + B) / B ≥ (2B − 1) / B.
- Best is **k = B**: ratio **2 − 1/B**.

Choosing k at random lowers the expected ratio to **e / (e − 1) ≈ 1.58** (Karlin et al., 1994).

k	rent	ALG	OPT	ratio
1	0	10	1	10
2	1	11	2	5.5
5	4	14	5	2.8
9	8	18	9	2
10	9	19	10	**1.9**
11	10	20	10	2
15	14	24	10	2.4
20	19	29	10	2.9

B = 10; d = k (the adversary's choice). The table shows that the worst ratio is smallest at k = B.

Try It: Ski Rental

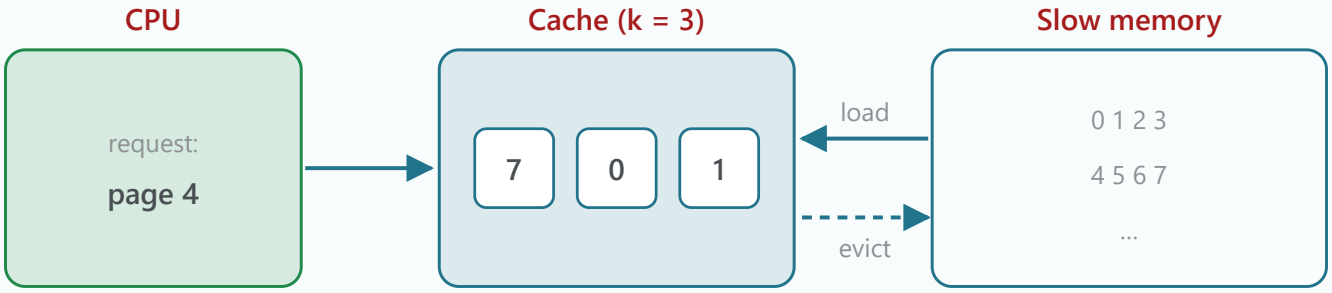
B **10** buying day k **10** actual days d **12**



9 days of rent + 10 TL to buy. Ratio ALG / OPT = **1.90** · worst ratio for this k (d = k) = **1.90** · best deterministic: 2 − 1/B = **1.90**

Strategy: rent for k − 1 days; on day k, if still skiing, buy. The worst ratio arises when the adversary picks d = k.

The Paging Problem



- Fast memory (the cache) holds **k pages**; slow memory is unbounded.
- If the requested page is in the cache, it is a **hit** (cost 0); otherwise a **page fault** (cost 1): the page is loaded.
- If the cache is full, **which page should be evicted**? Future requests are unknown: a typical online problem.
- Goal: minimize the total **number of page faults**.

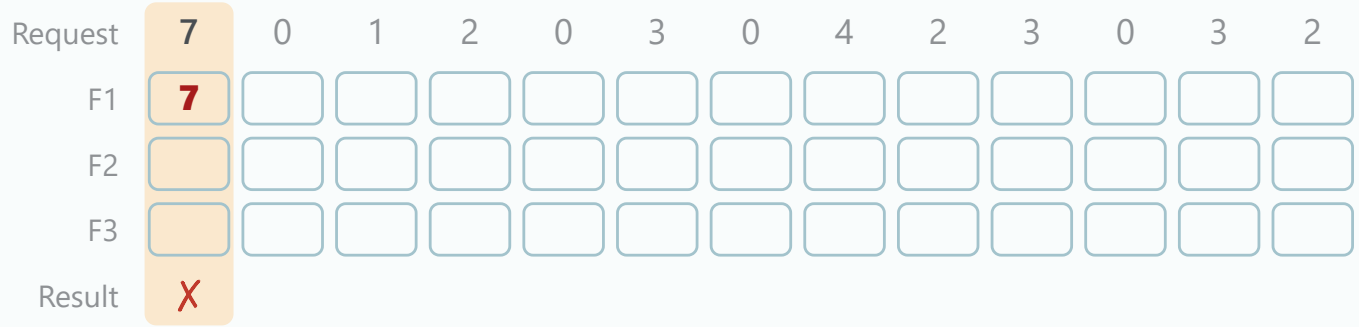
Page Eviction Policies

Policy	Which page does it evict?	Online?	Competitive ratio
FIFO	The page **loaded earliest** (a queue)	yes	k
LRU	The **least recently used** page	yes	k
FWF	On a fault with a full cache, evicts **all** pages (flush when full)	yes	k
LFU	The **least frequently used** page	yes	unbounded
LIFO	The page **loaded most recently**	yes	unbounded
OPT (MIN)	The page whose next use is **farthest in the future** (Belady, 1966)	no	1

OPT (longest forward distance) requires knowing the future; it cannot be implemented but serves as a **benchmark**. It provably incurs the fewest page faults.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

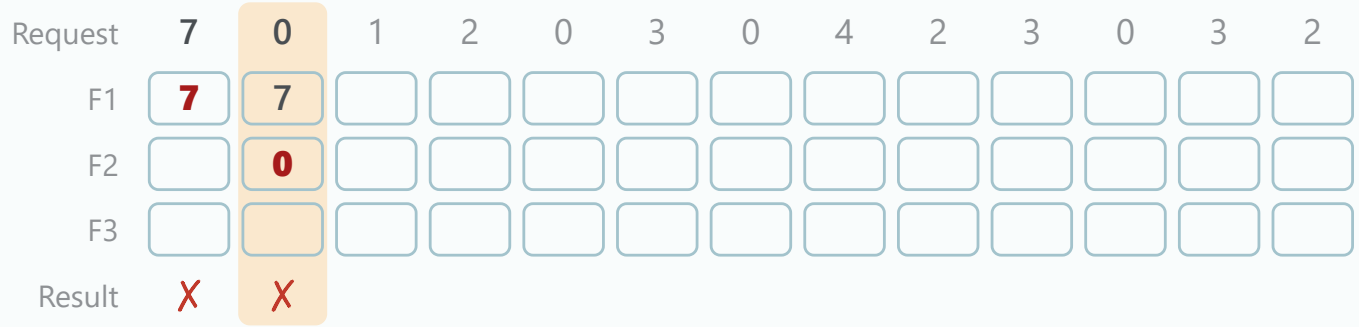


FIFO · k = 3 · page faults: 1

Request 1: page **7** missing → **page fault**; loaded into empty frame F1. Total faults: 1.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.



FIFO · k = 3 · page faults: 2

Request 2: page **0** missing → **page fault**; loaded into empty frame F2. Total faults: 2.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

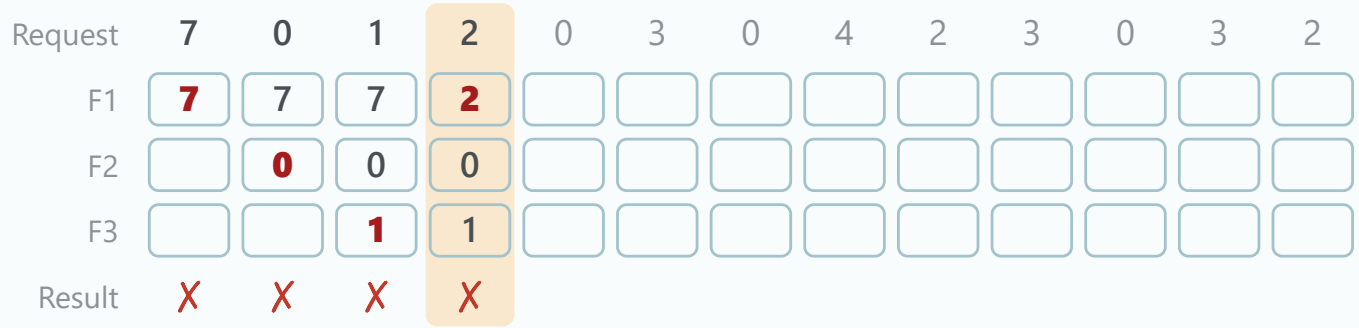
Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7										
F2		**0**	0										
F3			**1**										
Result	✗	✗	✗										

FIFO · k = 3 · page faults: 3

Request 3: page **1** missing → **page fault**; loaded into empty frame F3. Total faults: 3.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.



FIFO · k = 3 · page faults: 4

Request 4: page **2** missing → **page fault**; **7**, loaded earliest, is evicted. Total faults: 4.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2								
F2		**0**	0	0	**0**								
F3			**1**	1	1								
Result	✗	✗	✗	✗	✓								

FIFO · k = 3 · page faults: 4

Request 5: page **0** is cached → **hit**. The FIFO order **does not change**. Total faults: 4.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3.
✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2							
F2		**0**	0	0	**0**	**3**							
F3			**1**	1	1	1							
Result	✗	✗	✗	✗	✓	✗							

FIFO · k = 3 · page faults: 5

Request 6: page **3** missing → **page fault**; **0**, loaded earliest, is evicted. Total faults: 5.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2						
F2		**0**	0	0	**0**	**3**	3						
F3			**1**	1	1	1	**0**						
Result	✗	✗	✗	✗	✓	✗	✗						

FIFO · k = 3 · page faults: 6

Request 7: page **0** missing → **page fault**; **1**, loaded earliest, is evicted. Total faults: 6.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**					
F2		**0**	0	0	**0**	**3**	3	3					
F3			**1**	1	1	1	**0**	0					
Result	✗	✗	✗	✗	✓	✗	✗	✗					

FIFO · k = 3 · page faults: 7

Request 8: page **4** missing → **page fault**; **2**, loaded earliest, is evicted. Total faults: 7.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**	4				
F2		**0**	0	0	**0**	**3**	3	3	**2**				
F3			**1**	1	1	1	**0**	0	0				
Result	✗	✗	✗	✗	✓	✗	✗	✗	✗				

FIFO · k = 3 · page faults: 8

Request 9: page **2** missing → **page fault**; **3**, loaded earliest, is evicted. Total faults: 8.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**	4	4			
F2		**0**	0	0	**0**	**3**	3	3	**2**	2			
F3			**1**	1	1	1	**0**	0	0	**3**			
Result	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗			

FIFO · k = 3 · page faults: 9

Request 10: page **3** missing → **page fault**; **0**, loaded earliest, is evicted. Total faults: 9.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**		
F2		**0**	0	0	**0**	**3**	3	3	**2**	2	2		
F3			**1**	1	1	1	**0**	0	0	**3**	3		
Result	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗		

FIFO · k = 3 · page faults: 10

Request 11: page **0** missing → **page fault**; **4**, loaded earliest, is evicted. Total faults: 10.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3.
✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	
F2		**0**	0	0	**0**	**3**	3	3	**2**	2	2	2	
F3			**1**	1	1	1	**0**	0	0	**3**	3	**3**	
Result	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	

FIFO · k = 3 · page faults: 10

Request 12: page **3** is cached → **hit**. The FIFO order **does not change**. Total faults: 10.

FIFO Step by Step

FIFO: the cache behaves like a **queue**; first in, first out. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

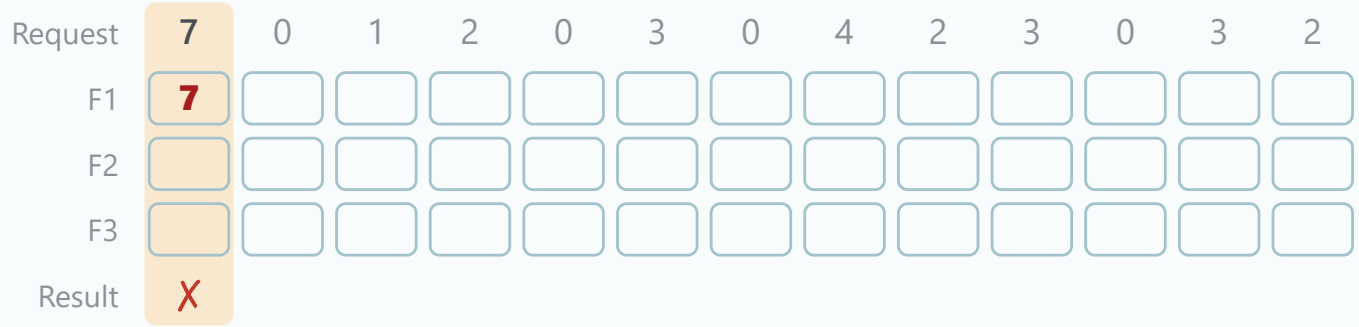
Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0
F2		**0**	0	0	**0**	**3**	3	3	**2**	2	2	2	**2**
F3			**1**	1	1	1	**0**	0	0	**3**	3	**3**	3
Result	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓

FIFO · k = 3 · page faults: 10

Request 13: page **2** is cached → **hit**. The FIFO order **does not change**. Total faults: 10.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

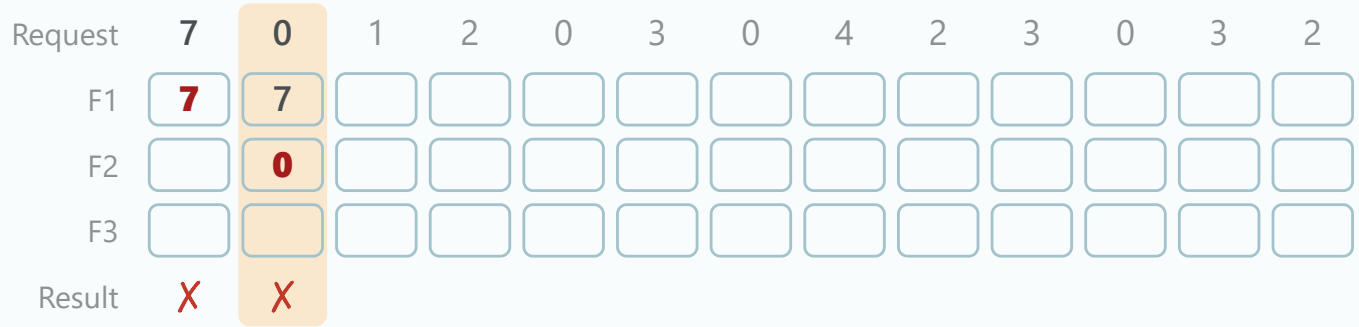


LRU · k = 3 · page faults: 1

Request 1: page **7** missing → **page fault**; loaded into empty frame F1. Total faults: 1.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

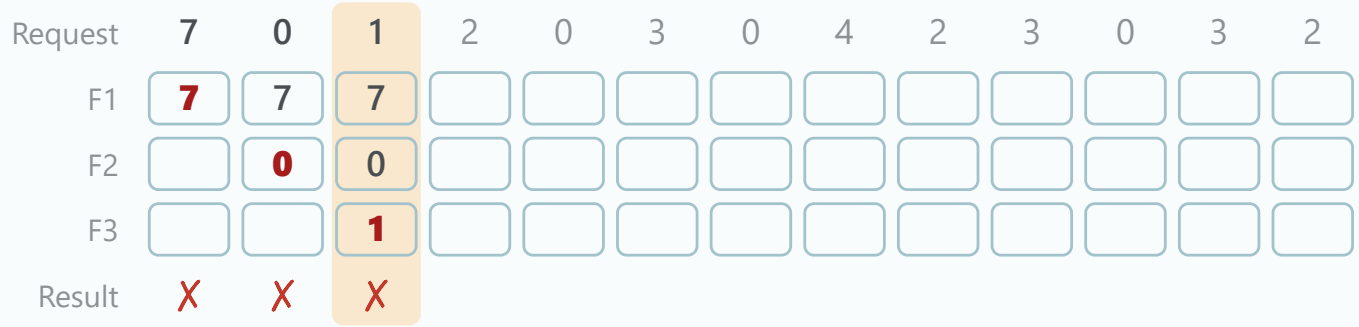


LRU · k = 3 · page faults: 2

Request 2: page **0** missing → **page fault**; loaded into empty frame F2. Total faults: 2.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

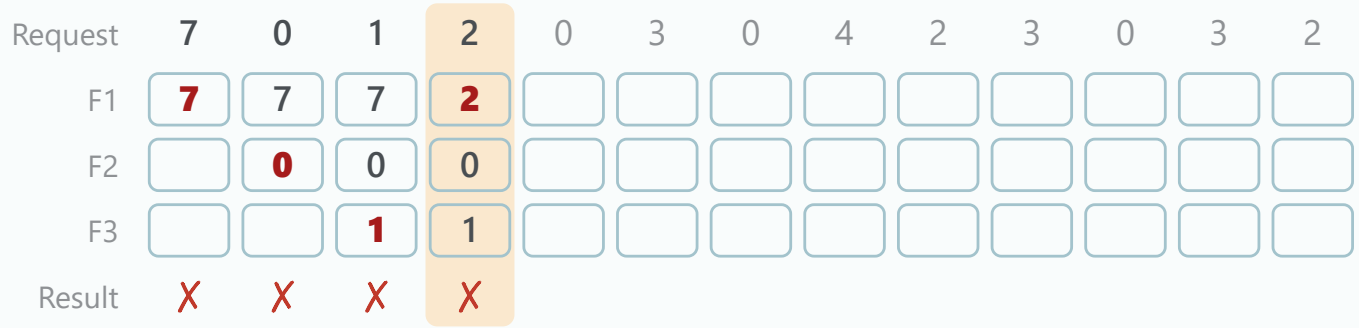


LRU · k = 3 · page faults: 3

Request 3: page **1** missing → **page fault**; loaded into empty frame F3. Total faults: 3.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.



LRU · k = 3 · page faults: 4

Request 4: page **2** missing → **page fault**; **7**, least recently used, is evicted. Total faults: 4.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2								
F2		**0**	0	0	**0**								
F3			**1**	1	1								
Result	✗	✗	✗	✗	✓								

LRU · k = 3 · page faults: 4

Request 5: page **0** is cached → **hit**. 0 is now the most recently used page. Total faults: 4.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2							
F2		**0**	0	0	**0**	0							
F3			**1**	1	1	**3**							
Result	✗	✗	✗	✗	✓	✗							

LRU · k = 3 · page faults: 5

Request 6: page **3** missing → **page fault**; **1**, least recently used, is evicted. Total faults: 5.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2						
F2		**0**	0	0	**0**	0	**0**						
F3			**1**	1	1	**3**	3						
Result	✗	✗	✗	✗	✓	✗	✓						

LRU · k = 3 · page faults: 5

Request 7: page **0** is cached → **hit**. 0 is now the most recently used page. Total faults: 5.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**					
F2		**0**	0	0	**0**	0	**0**	0					
F3			**1**	1	1	**3**	3	3					
Result	✗	✗	✗	✗	✓	✗	✓	✗					

LRU · k = 3 · page faults: 6

Request 8: page **4** missing → **page fault**; **2**, least recently used, is evicted. Total faults: 6.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**	4				
F2		**0**	0	0	**0**	0	**0**	0	0				
F3			**1**	1	1	**3**	3	3	**2**				
Result	✗	✗	✗	✗	✓	✗	✓	✗	✗				

LRU · k = 3 · page faults: 7

Request 9: page **2** missing → **page fault**; **3**, least recently used, is evicted. Total faults: 7.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**	4	4			
F2		**0**	0	0	**0**	0	**0**	0	0	**3**			
F3			**1**	1	1	**3**	3	3	**2**	2			
Result	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗			

LRU · k = 3 · page faults: 8

Request 10: page **3** missing → **page fault**; **0**, least recently used, is evicted. Total faults: 8.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**		
F2		**0**	0	0	**0**	0	**0**	0	0	**3**	3		
F3			**1**	1	1	**3**	3	3	**2**	2	2		
Result	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗		

LRU · k = 3 · page faults: 9

Request 11: page **0** missing → **page fault**; **4**, least recently used, is evicted. Total faults: 9.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	
F2		**0**	0	0	**0**	0	**0**	0	0	**3**	3	**3**	
F3			**1**	1	1	**3**	3	3	**2**	2	2	2	
Result	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓	

LRU · k = 3 · page faults: 9

Request 12: page **3** is cached → **hit**. 3 is now the most recently used page. Total faults: 9.

LRU Step by Step

LRU: track each page's **last use time**; evict the oldest. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

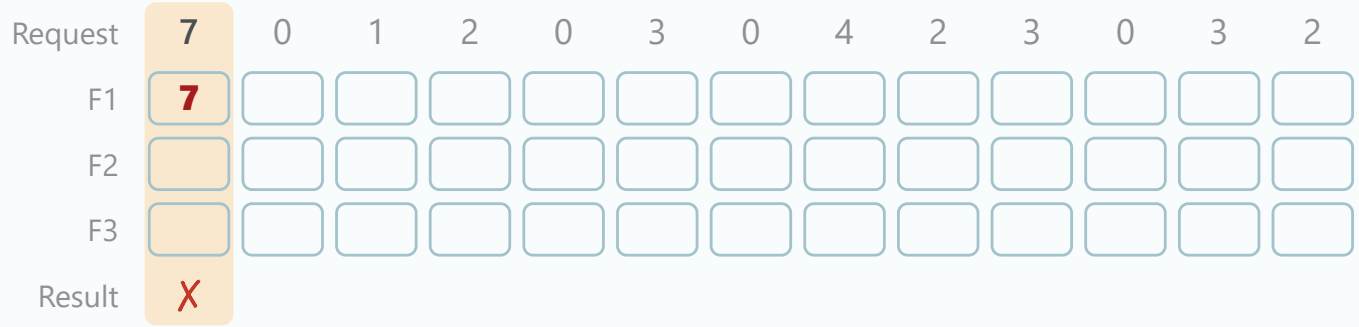
Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0
F2		**0**	0	0	**0**	0	**0**	0	0	**3**	3	**3**	3
F3			**1**	1	1	**3**	3	3	**2**	2	2	2	**2**
Result	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓	✓

LRU · k = 3 · page faults: 9

Request 13: page **2** is cached → **hit**. 2 is now the most recently used page. Total faults: 9.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

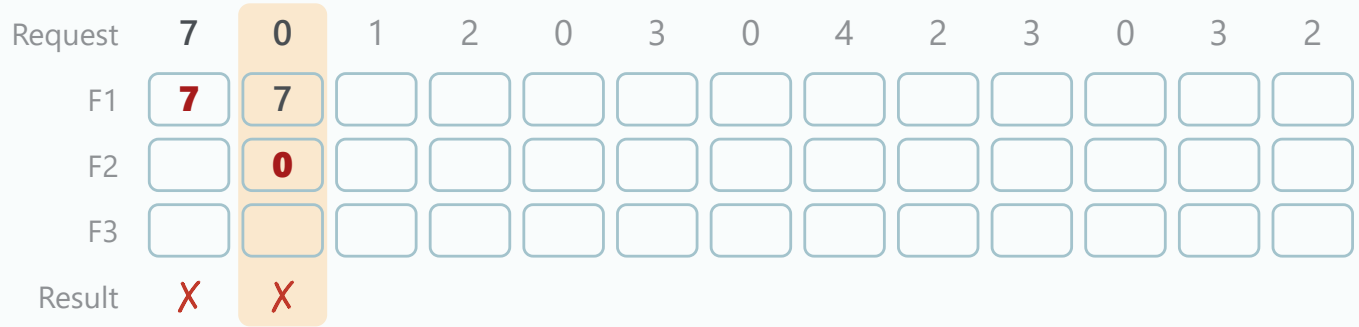


OPT · k = 3 · page faults: 1

Request 1: page **7** missing → **page fault**; loaded into empty frame F1. Total faults: 1.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.



OPT · k = 3 · page faults: 2

Request 2: page **0** missing → **page fault**; loaded into empty frame F2. Total faults: 2.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7										
F2		**0**	0										
F3			**1**										
Result	✗	✗	✗										

OPT · k = 3 · page faults: 3

Request 3: page **1** missing → **page fault**; loaded into empty frame F3. Total faults: 3.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**									
F2		**0**	0	0									
F3			**1**	1									
Result	✗	✗	✗	✗									

OPT · k = 3 · page faults: 4

Request 4: page **2** missing → **page fault**; **7** is never requested again → evicted. Total faults: 4.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2								
F2		**0**	0	0	**0**								
F3			**1**	1	1								
Result	✗	✗	✗	✗	✓								

OPT · k = 3 · page faults: 4

Request 5: page **0** is cached → **hit**. Total faults: 4.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2							
F2		**0**	0	0	**0**	0							
F3			**1**	1	1	**3**							
Result	✗	✗	✗	✗	✓	✗							

OPT · k = 3 · page faults: 5

Request 6: page **3** missing → **page fault**; **1** is never requested again → evicted. Total faults: 5.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2						
F2		**0**	0	0	**0**	0	**0**						
F3			**1**	1	1	**3**	3						
Result	✗	✗	✗	✗	✓	✗	✓						

OPT · k = 3 · page faults: 5

Request 7: page **0** is cached → **hit**. Total faults: 5.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	2					
F2		**0**	0	0	**0**	0	**0**	**4**					
F3			**1**	1	1	**3**	3	3					
Result	✗	✗	✗	✗	✓	✗	✓	✗					

OPT · k = 3 · page faults: 6

Request 8: page **4** missing → **page fault**; **0**, whose next use is farthest (request 11), is evicted. Total faults: 6.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	2	**2**				
F2		**0**	0	0	**0**	0	**0**	**4**	4				
F3			**1**	1	1	**3**	3	3	3				
Result	✗	✗	✗	✗	✓	✗	✓	✗	✓				

OPT · k = 3 · page faults: 6

Request 9: page **2** is cached → **hit**. Total faults: 6.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	2	**2**	2			
F2		**0**	0	0	**0**	0	**0**	**4**	4	4			
F3			**1**	1	1	**3**	3	3	3	**3**			
Result	✗	✗	✗	✗	✓	✗	✓	✗	✓	✓			

OPT · k = 3 · page faults: 6

Request 10: page **3** is cached → **hit**. Total faults: 6.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	2	**2**	2	2		
F2		**0**	0	0	**0**	0	**0**	**4**	4	4	**0**		
F3			**1**	1	1	**3**	3	3	3	**3**	3		
Result	✗	✗	✗	✗	✓	✗	✓	✗	✓	✓	✗		

OPT · k = 3 · page faults: 7

Request 11: page **0** missing → **page fault**; **4** is never requested again → evicted. Total faults: 7.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	2	**2**	2	2	2	
F2		**0**	0	0	**0**	0	**0**	**4**	4	4	**0**	0	
F3			**1**	1	1	**3**	3	3	3	**3**	3	**3**	
Result	✗	✗	✗	✗	✓	✗	✓	✗	✓	✓	✗	✓	

OPT · k = 3 · page faults: 7

Request 12: page **3** is cached → **hit**. Total faults: 7.

OPT Step by Step

OPT: knows the future; evicts the page whose next use is **farthest away**. Request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. ✗: fault, ✓: hit.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	**7**	7	7	**2**	2	2	2	2	**2**	2	2	2	**2**
F2		**0**	0	0	**0**	0	**0**	**4**	4	4	**0**	0	0
F3			**1**	1	1	**3**	3	3	3	**3**	3	**3**	3
Result	✗	✗	✗	✗	✓	✗	✓	✗	✓	✓	✗	✓	✓

OPT · k = 3 · page faults: 7

Request 13: page **2** is cached → **hit**. Total faults: 7.

Comparing the Three Policies

Policy	Page faults	Hits	ALG / OPT	Information needed
FIFO	10	3	1.43	Load order (queue)
LRU	9	4	1.29	Last use time
OPT	7	6	1	All future requests

- The first k = 3 faults are **unavoidable** (compulsory or cold misses): every policy pays them.
- LRU does well on sequences with **locality of reference**: a page used recently is likely to be used again soon. Here page 0 is requested often, so FIFO evicts it needlessly.
- Real systems use cheap approximations of LRU: the **clock algorithm** (second chance).

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
▶    var cache = new LinkedList<Integer>();
▶    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
8        faults++;
9        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
11       cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```

request: 1 2 3 1 4 2

cache: [ ] [ ] [ ]

oldest newest

The cache is empty, `faults = 0`. The head of the list is the least recently used page, the tail the most recently used.

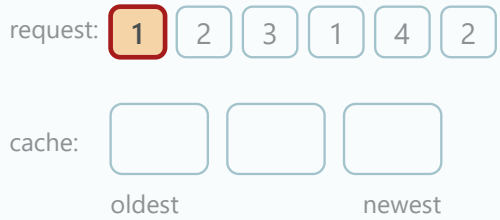
Step 1 / 19

VARIABLES

`k = 3` `page = –` `faults = 0` `cache = []`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
▶    for (int page : requests) {
▶      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
8        faults++;
9        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
11       cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



Page 1 is not in the list: `remove` returns false → else branch.

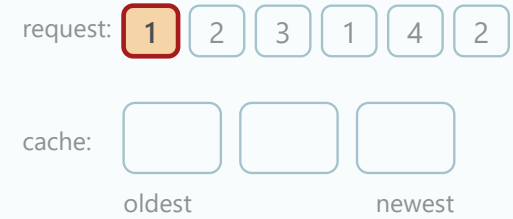
Step 2 / 19

VARIABLES

`k = 3` `page = 1` `faults = 0` `cache = []`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
▶        faults++;
▶        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
11       cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



`faults = 1` . The cache has room (size 0 < 3) → no eviction.

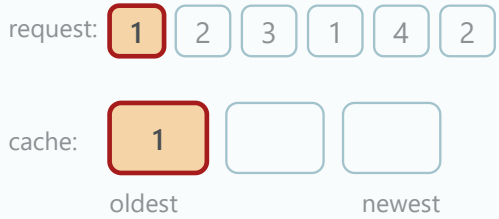
Step 3 / 19

VARIABLES

`k = 3` `page = 1` `faults = 1` `cache = []`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
8        faults++;
9        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
▶        cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



Page 1 is appended at the tail.

Step 4 / 19

VARIABLES

`k = 3` `page = 1` `faults = 1`

`cache = [1]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
▶   for (int page : requests) {
▶     if (cache.remove((Integer) page)) {
6       cache.addLast(page); // make newest
7     } else {
8       faults++;
9       if (cache.size() == k)
10        cache.removeFirst(); // drop oldest
11      cache.addLast(page);
12    }
13   }
14   return faults;
15 }
```



Page 2 is not in the list: `remove` returns false → else branch.

Step 5 / 19

VARIABLES

`k = 3` `page = 2` `faults = 1` `cache = [1]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
▶        faults++;
▶        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
11       cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



`faults = 2`. The cache has room (size 1 < 3) → no eviction.

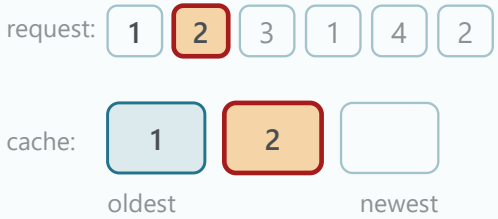
Step 6 / 19

VARIABLES

`k = 3` `page = 2` `faults = 2` `cache = [1]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
8        faults++;
9        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
▶        cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



Page 2 is appended at the tail.

Step 7 / 19

VARIABLES

`k = 3` `page = 2` `faults = 2`

`cache = [1, 2]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
▶    for (int page : requests) {
▶      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
8        faults++;
9        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
11       cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



Page 3 is not in the list: `remove` returns false → else branch.

Step 8 / 19

VARIABLES

`k = 3` `page = 3` `faults = 2` `cache = [1, 2]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
▶        faults++;
▶        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
11       cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



`faults = 3`. The cache has room (size 2 < 3) → no eviction.

Step 9 / 19

VARIABLES

`k = 3` `page = 3` `faults = 3` `cache = [1, 2]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
8        faults++;
9        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
▶        cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



Page 3 is appended at the tail.

Step 10 / 19

VARIABLES

`k = 3` `page = 3` `faults = 3`

`cache = [1, 2, 3]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
▶    for (int page : requests) {
▶      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
8        faults++;
9        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
11       cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



Page 1 is in the list: `remove` returns true and removes it → **hit**.

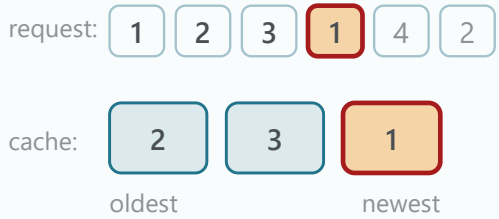
Step 11 / 19

VARIABLES

`k = 3` `page = 1` `faults = 3` `cache = [2, 3]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
▶        cache.addLast(page); // make newest
7      } else {
8        faults++;
9        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
11       cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



Page 1 is appended at the tail: now the **most recently** used.

Step 12 / 19

VARIABLES

`k = 3` `page = 1` `faults = 3`

`cache = [2, 3, 1]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
▶   for (int page : requests) {
▶     if (cache.remove((Integer) page)) {
6       cache.addLast(page); // make newest
7     } else {
8       faults++;
9       if (cache.size() == k)
10        cache.removeFirst(); // drop oldest
11      cache.addLast(page);
12    }
13   }
14   return faults;
15 }
```



Page 4 is not in the list: `remove` returns false → else branch.

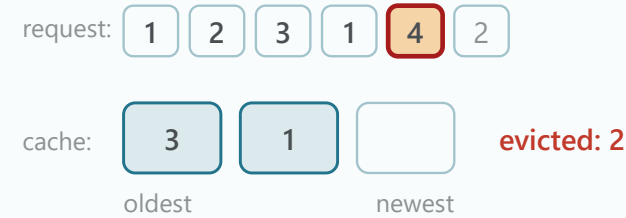
Step 13 / 19

VARIABLES

`k = 3` `page = 4` `faults = 3` `cache = [2, 3, 1]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
▶        faults++;
▶        if (cache.size() == k)
▶          cache.removeFirst(); // drop oldest
11       cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



`faults = 4`. The cache is full → the head (least recently used) **2** is evicted.

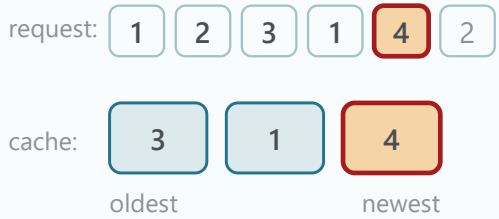
Step 14 / 19

VARIABLES

`k = 3` `page = 4` `faults = 4` `cache = [3, 1]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
8        faults++;
9        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
▶        cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



Page 4 is appended at the tail.

Step 15 / 19

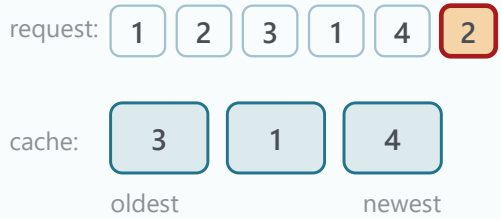
VARIABLES

`k = 3` `page = 4` `faults = 4`

`cache = [3, 1, 4]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
▶   for (int page : requests) {
▶     if (cache.remove((Integer) page)) {
6       cache.addLast(page); // make newest
7     } else {
8       faults++;
9       if (cache.size() == k)
10        cache.removeFirst(); // drop oldest
11      cache.addLast(page);
12    }
13   }
14   return faults;
15 }
```



Page 2 is not in the list: `remove` returns false → else branch.

Step 16 / 19

VARIABLES

`k = 3` `page = 2` `faults = 4` `cache = [3, 1, 4]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
▶        faults++;
▶        if (cache.size() == k)
▶          cache.removeFirst(); // drop oldest
11       cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```

request: 1 2 3 1 4 **2**

cache: 1 4 ( ) — evicted: 3

oldest … newest

`faults = 5`. The cache is full → the head (least recently used) **3** is evicted.

Step 17 / 19

VARIABLES

`k = 3` `page = 2` `faults = 5` `cache = [1, 4]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
8        faults++;
9        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
▶        cache.addLast(page);
12     }
13   }
14   return faults;
15 }
```



Page 2 is appended at the tail.

Step 18 / 19

VARIABLES

`k = 3` `page = 2` `faults = 5`

`cache = [1, 4, 2]`

LRU: Code Trace

```
1  int lruFaults(int[] requests, int k) {
2    var cache = new LinkedList<Integer>();
3    int faults = 0;
4    for (int page : requests) {
5      if (cache.remove((Integer) page)) {
6        cache.addLast(page); // make newest
7      } else {
8        faults++;
9        if (cache.size() == k)
10         cache.removeFirst(); // drop oldest
11       cache.addLast(page);
12     }
13   }
▶    return faults;
15 }
```

request: 1 2 3 1 4 2

cache: 1 4 2

oldest newest

Sequence done: **5 page faults** and 1 hit in 6 requests. Since `remove` scans the list, each request costs O(k); with a HashMap + doubly linked list it becomes O(1).

Step 19 / 19

VARIABLES

`k = 3` `page = –` `faults = 5`

`cache = [1, 4, 2]`

LRU Cache: O(1) Implementation

```
class LruCache<K, V>
    extends LinkedHashMap<K, V> {
  private final int k;

  LruCache(int k) {
    super(16, 0.75f, true); // access order
    this.k = k;
  }

  @Override
  protected boolean removeEldestEntry(
      Map.Entry<K, V> eldest) {
    return size() > k;       // evict beyond k
  }
}
```

- `LinkedHashMap` = **hash table + doubly linked list**.
- `true` (accessOrder): every `get` / `put` moves the entry to the **end** of the list.
- `removeEldestEntry`: once the size exceeds k, the **head** (least recently used) is removed.
- Find, move, remove: all **O(1)**; memory O(k).

For FIFO, build the same class with `false` (insertion order): accesses do not change the order.

Theorem: Deterministic Lower Bound k

- k + 1 distinct pages suffice. At every step the adversary requests the one page **not** in ALG's cache → ALG faults on **every request**.
- When OPT (evict farthest) faults, the page it evicts is not requested for at least k − 1 requests → OPT faults at most **once every k requests**.
- Hence ALG / OPT → **k** (Sleator and Tarjan, 1985).

In the example (k = 3, against LRU): LRU 12 faults, OPT 6; excluding the first 3 compulsory faults, 9 to 3.

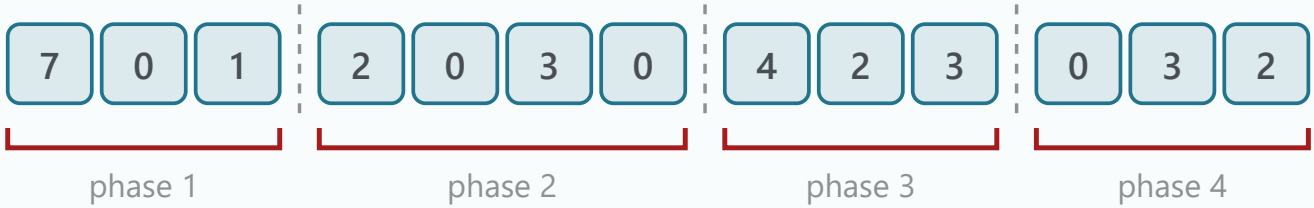
Request	1	2	3	4	1	2	3	4	1	2	3	4
F1	1	1	1	4	4	4	3	3	3	2	2	2
F2		2	2	2	1	1	1	4	4	4	3	3
F3			3	3	3	2	2	2	1	1	1	4
Result	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

LRU · k = 3 · page faults: 12

Request	1	2	3	4	1	2	3	4	1	2	3	4
F1	1	1	1	1	1	1	1	1	1	2	2	2
F2		2	2	2	2	2	3	3	3	3	3	3
F3			3	4	4	4	4	4	4	4	4	4
Result	✗	✗	✗	✗	✓	✓	✗	✓	✓	✗	✓	✓

OPT · k = 3 · page faults: 6

Theorem: LRU (and FIFO) Are k-Competitive



- Split the sequence into **phases**: each phase is a maximal block with **k distinct pages**; the (k + 1)-th distinct page starts a new phase.
- **LRU faults at most k times per phase:** a page requested in a phase is not evicted before the phase ends (some page used less recently always exists). Each distinct page faults at most once.
- **OPT faults at least once per two consecutive phases:** from the 2nd request of one phase to the 1st request of the next, k + 1 distinct pages are requested; they do not fit in k frames.
- For m phases, LRU ≤ k · m and OPT ≥ m − 1 → **LRU ≤ k · OPT + k**.

The example sequence has 4 phases: LRU = 9 ≤ 3 · 4, OPT = 7 ≥ 3. Theory bounds the worst case; on real sequences, locality keeps LRU much closer to OPT.

Belady's Anomaly: More Memory, More Faults

Request	1	2	3	4	1	2	5	1	2	3	4	5
F1	**1**	1	1	**4**	4	4	**5**	5	5	5	5	**5**
F2		**2**	2	2	**1**	1	1	**1**	1	**3**	3	3
F3			**3**	3	3	**2**	2	2	**2**	2	**4**	4
Result	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓

FIFO · k = 3 · page faults: 9

Request	1	2	3	4	1	2	5	1	2	3	4	5
F1	**1**	1	1	1	**1**	1	**5**	5	5	5	**4**	4
F2		**2**	2	2	2	**2**	2	**1**	1	1	1	**5**
F3			**3**	3	3	3	3	3	**2**	2	2	2
F4				**4**	4	4	4	4	4	**3**	3	3
Result	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗

FIFO · k = 4 · page faults: 10

- Sequence: 1 2 3 4 1 2 5 1 2 3 4 5.
- FIFO, k = 3: **9 faults**; k = 4: **10 faults**. The cache grew, and so did the faults!
- LRU on the same sequence: k = 3 → 10, k = 4 → 8. LRU and OPT are **stack algorithms**: the pages in k frames are a subset of those in k + 1 frames → no anomaly.

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
▶    Queue<Integer> queue = new LinkedList<>();
▶    Set<Integer> cached = new HashSet<>();
▶    int faults = 0;
5    for (int p : requests) {
6      if (cached.contains(p)) continue;   // hit
7      faults++;
8      if (cached.size() == k)
9        cached.remove(queue.poll()); // oldest
10     queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



queue

faults = 0

oldest

Belady's sequence 1 2 3 4 1 2 5 1 2 3 4 5, k = 3. The queue keeps the load order; the set answers "is it cached?" quickly.

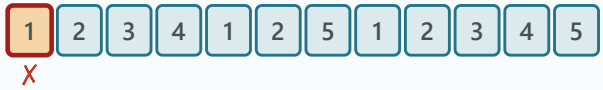
Step 1 / 14

VARIABLES

`k = 3` `faults = 0`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
6      if (cached.contains(p)) continue;   // hit
       faults++;
       if (cached.size() == k)
9        cached.remove(queue.poll()); // oldest
       queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



queue [1] oldest

faults = 1

p = 1 missing → fault 1 (compulsory); there is room, so it joins the queue.

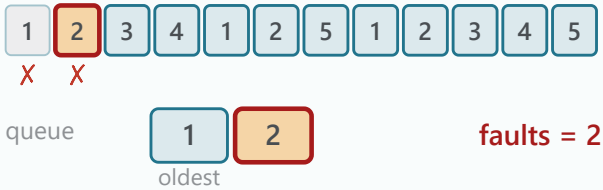
Step 2 / 14

VARIABLES

`p = 1` `faults = 1` `queue = [1]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
6      if (cached.contains(p)) continue;   // hit
▶      faults++;
▶      if (cached.size() == k)
9        cached.remove(queue.poll()); // oldest
▶      queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



p = 2 missing → fault 2 (compulsory); there is room, so it joins the queue.

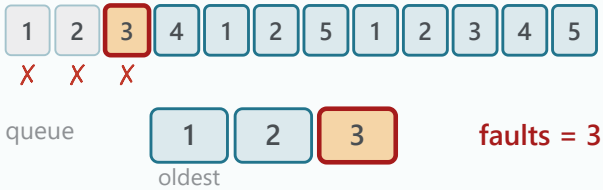
Step 3 / 14

VARIABLES

`p = 2` `faults = 2` `queue = [1, 2]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
6      if (cached.contains(p)) continue;   // hit
▶      faults++;
▶      if (cached.size() == k)
9        cached.remove(queue.poll()); // oldest
▶      queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



p = 3 missing → fault 3 (compulsory); there is room, so it joins the queue.

Step 4 / 14

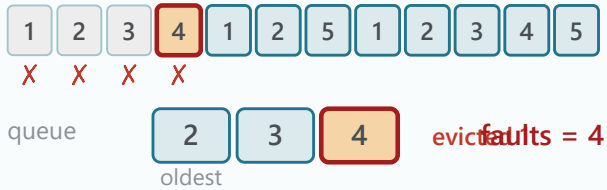
VARIABLES

`p = 3` `faults = 3`

`queue = [1, 2, 3]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
6      if (cached.contains(p)) continue;   // hit
▶      faults++;
▶      if (cached.size() == k)
▶        cached.remove(queue.poll()); // oldest
▶      queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



p = 4 missing → fault 4. The cache is full: **1**, loaded earliest, leaves and 4 joins the back of the queue.

Step 5 / 14

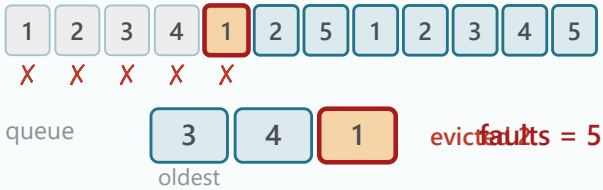
VARIABLES

`p = 4` `faults = 4`

`queue = [2, 3, 4]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
6      if (cached.contains(p)) continue;   // hit
▶      faults++;
▶      if (cached.size() == k)
▶        cached.remove(queue.poll()); // oldest
▶      queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



p = 1 missing → fault 5. The cache is full: **2**, loaded earliest, leaves and 1 joins the back of the queue.

Step 6 / 14

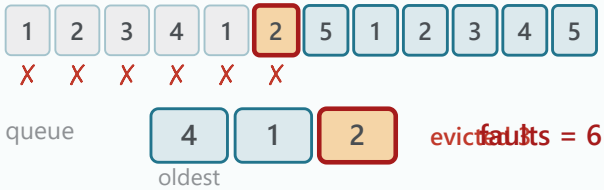
VARIABLES

`p = 1` `faults = 5`

`queue = [3, 4, 1]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
6      if (cached.contains(p)) continue;   // hit
▶      faults++;
▶      if (cached.size() == k)
▶        cached.remove(queue.poll()); // oldest
▶      queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



p = 2 missing → fault 6. The cache is full: **3**, loaded earliest, leaves and 2 joins the back of the queue.

Step 7 / 14

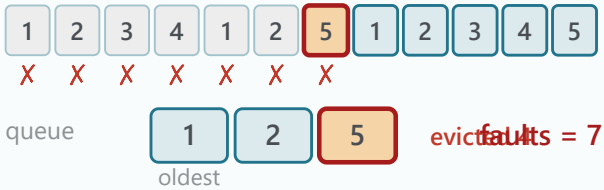
VARIABLES

`p = 2` `faults = 6`

`queue = [4, 1, 2]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
6      if (cached.contains(p)) continue;   // hit
▶      faults++;
▶      if (cached.size() == k)
▶        cached.remove(queue.poll()); // oldest
▶      queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



p = 5 missing → fault 7. The cache is full: **4**, loaded earliest, leaves and 5 joins the back of the queue.

Step 8 / 14

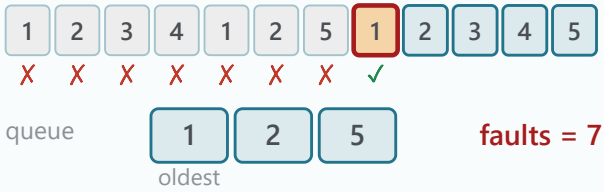
VARIABLES

`p = 5` `faults = 7`

`queue = [1, 2, 5]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (cached.contains(p)) continue;   // hit
7      faults++;
8      if (cached.size() == k)
9        cached.remove(queue.poll()); // oldest
10     queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



p = 1 is cached → **hit**. In FIFO a hit does not change the queue order.

Step 9 / 14

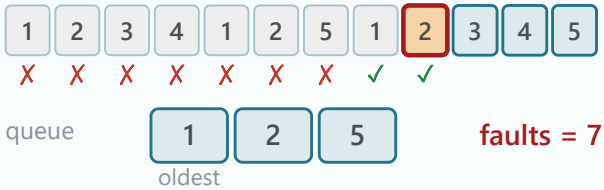
VARIABLES

`p = 1` `faults = 7`

`queue = [1, 2, 5]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (cached.contains(p)) continue;   // hit
7      faults++;
8      if (cached.size() == k)
9        cached.remove(queue.poll()); // oldest
10     queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



p = 2 is cached → **hit**. In FIFO a hit does not change the queue order.

Step 10 / 14

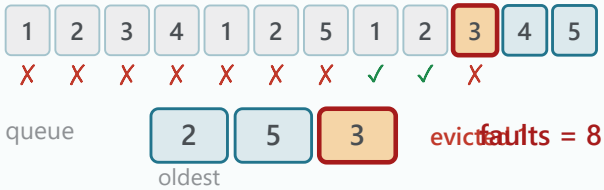
VARIABLES

`p = 2` `faults = 7`

`queue = [1, 2, 5]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
6      if (cached.contains(p)) continue;   // hit
▶      faults++;
▶      if (cached.size() == k)
▶        cached.remove(queue.poll()); // oldest
▶      queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



p = 3 missing → fault 8. The cache is full: **1**, loaded earliest, leaves and 3 joins the back of the queue.

Step 11 / 14

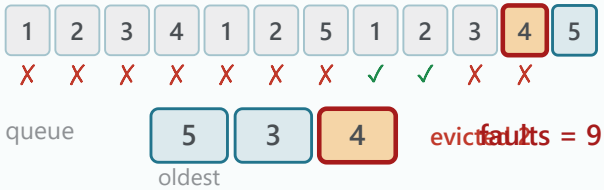
VARIABLES

`p = 3` `faults = 8`

`queue = [2, 5, 3]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
6      if (cached.contains(p)) continue;   // hit
▶      faults++;
▶      if (cached.size() == k)
▶        cached.remove(queue.poll()); // oldest
▶      queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



p = 4 missing → fault 9. The cache is full: **2**, loaded earliest, leaves and 4 joins the back of the queue.

Step 12 / 14

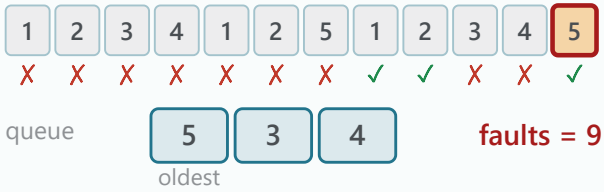
VARIABLES

`p = 4` `faults = 9`

`queue = [5, 3, 4]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (cached.contains(p)) continue;   // hit
7      faults++;
8      if (cached.size() == k)
9        cached.remove(queue.poll()); // oldest
10     queue.offer(p); cached.add(p);
11   }
12   return faults;
13 }
```



p = 5 is cached → **hit**. In FIFO a hit does not change the queue order.

Step 13 / 14

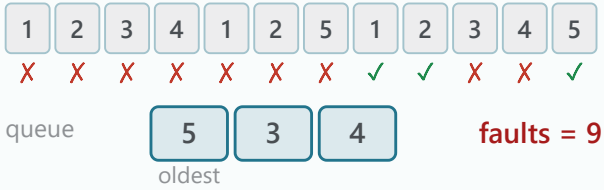
VARIABLES

`p = 5` `faults = 9`

`queue = [5, 3, 4]`

FIFO on Belady's Sequence: Code Trace

```
1  int fifoFaults(int[] requests, int k) {
2    Queue<Integer> queue = new LinkedList<>();
3    Set<Integer> cached = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
6      if (cached.contains(p)) continue;   // hit
7      faults++;
8      if (cached.size() == k)
9        cached.remove(queue.poll()); // oldest
10     queue.offer(p); cached.add(p);
11   }
▶    return faults;
13 }
```



`return 9`. The same code with k = 4 gives 10 faults: more frames, more faults (Belady's anomaly).

Step 14 / 14

VARIABLES

`faults = 9` `returned = 9`

Randomized Paging: The Marking Algorithm

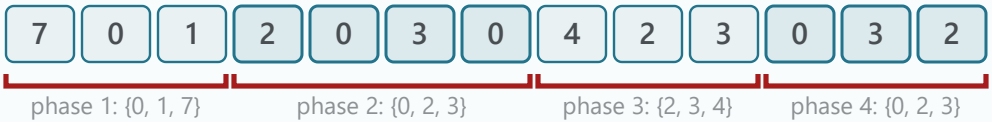
Marking

1. At the start of a phase, all pages are unmarked.
2. The requested page is **marked** (hit or fault).
3. On a fault, evict a **random** page among the **unmarked** ones.
4. If all pages are marked, a new phase begins (step 1).

Results

- Expected ratio $\mathbf{2H_k}$ against an oblivious adversary (Fiat et al., 1991).
- $H_k = 1 + 1/2 + \ldots + 1/k \approx \ln k$. For $k = 8$, $H_8 \approx 2.72 \rightarrow$ ratio ≈ 5.4 (8 for deterministic).
- No randomized algorithm can beat $\mathbf{H_k}$.
- LRU is also a marking algorithm (it picks the oldest unmarked page).

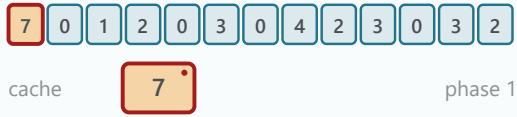
k = 3: each phase has at most k distinct pages; the (k + 1)-th distinct page starts a new phase



Randomness defeats the adversary's strategy of "request the page ALG does not have": it cannot know which page was evicted.

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (!cache.contains(p)) {
▶        faults++;
▶        if (cache.size() == k) {
9          if (marked.size() == k) marked.clear(); // new phase
10         List<Integer> cand = new ArrayList<>(cache);
11         cand.removeAll(marked);       // unmarked ones
12         int i = rng.nextInt(cand.size());
13         cache.remove(cand.get(i));
14       }
▶        cache.add(p);
16     }
▶      marked.add(p);
18   }
19   return faults;
20 }
```



faults = 1

Requests 7 0 1 2 0 3 0 4 2 3 0 3 2, k = 3. p = 7 missing → fault 1; added to the cache and marked (•).

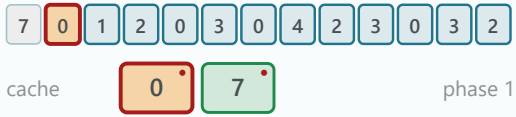
Step 1 / 14

VARIABLES

`p = 7` `faults = 1` `cache = [7]` `marked = [7]`

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (!cache.contains(p)) {
▶        faults++;
▶        if (cache.size() == k) {
9          if (marked.size() == k) marked.clear(); // new phase
10         List<Integer> cand = new ArrayList<>(cache);
11         cand.removeAll(marked);       // unmarked ones
12         int i = rng.nextInt(cand.size());
13         cache.remove(cand.get(i));
14       }
▶        cache.add(p);
16     }
▶      marked.add(p);
18   }
19   return faults;
20 }
```



faults = 2

p = 0 missing → fault 2 (compulsory); there is room, so it is added and marked.

Step 2 / 14

VARIABLES

`p = 0` `faults = 2`

`cache = [0, 7]`

`marked = [0, 7]`

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (!cache.contains(p)) {
▶        faults++;
▶        if (cache.size() == k) {
9          if (marked.size() == k) marked.clear(); // new
   phase
10         List<Integer> cand = new ArrayList<>(cache);
11         cand.removeAll(marked);       // unmarked ones
12         int i = rng.nextInt(cand.size());
13         cache.remove(cand.get(i));
14       }
▶        cache.add(p);
16     }
▶      marked.add(p);
18   }
19   return faults;
20 }
```



faults = 3

p = 1 missing → fault 3 (compulsory); there is room, so it is added and marked.

Step 3 / 14

VARIABLES

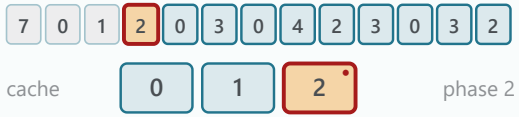
`p = 1` `faults = 3`

`cache = [0, 1, 7]`

`marked = [0, 1, 7]`

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (!cache.contains(p)) {
▶        faults++;
▶        if (cache.size() == k) {
▶          if (marked.size() == k) marked.clear(); // new
   phase
▶          List<Integer> cand = new ArrayList<>(cache);
▶          cand.removeAll(marked);       // unmarked ones
▶          int i = rng.nextInt(cand.size());
▶          cache.remove(cand.get(i));
14       }
▶        cache.add(p);
16     }
▶      marked.add(p);
18   }
19   return faults;
20 }
```



candidates [0, 1, 7] → evict 7 — faults = 4

All pages were marked → **new phase**: marks are cleared. p = 2 missing → fault 4. **7** is evicted at random from the unmarked candidates [0, 1, 7].

Step 4 / 14

VARIABLES

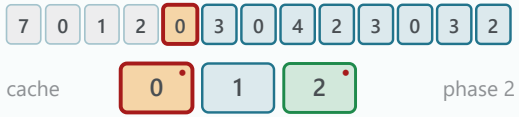
`p = 2` `faults = 4`

`cache = [0, 1, 2]`

`marked = [2]`

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (!cache.contains(p)) {
7        faults++;
8        if (cache.size() == k) {
9          if (marked.size() == k) marked.clear(); // new
   phase
10         List<Integer> cand = new ArrayList<>(cache);
11         cand.removeAll(marked);       // unmarked ones
12         int i = rng.nextInt(cand.size());
13         cache.remove(cand.get(i));
14       }
15       cache.add(p);
16     }
▶      marked.add(p);
18   }
19   return faults;
20 }
```



faults = 4

p = 0 is cached → **hit**; it is just marked (•).

Step 5 / 14

VARIABLES

p = 0 | faults = 4

cache = [0, 1, 2]

marked = [0, 2]

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (!cache.contains(p)) {
▶        faults++;
▶        if (cache.size() == k) {
▶          if (marked.size() == k) marked.clear(); // new phase
▶          List<Integer> cand = new ArrayList<>(cache);
▶          cand.removeAll(marked);       // unmarked ones
▶          int i = rng.nextInt(cand.size());
▶          cache.remove(cand.get(i));
14       }
▶        cache.add(p);
16     }
▶      marked.add(p);
18   }
19   return faults;
20 }
```



candidates [1] → evict 1 — faults = 5

p = 3 missing → fault 5. **1** is evicted at random from the unmarked candidates [1].

Step 6 / 14

VARIABLES

`p = 3` `faults = 5`

`cache = [0, 2, 3]`

`marked = [0, 2, 3]`

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (!cache.contains(p)) {
7        faults++;
8        if (cache.size() == k) {
9          if (marked.size() == k) marked.clear(); // new
   phase
10         List<Integer> cand = new ArrayList<>(cache);
11         cand.removeAll(marked);       // unmarked ones
12         int i = rng.nextInt(cand.size());
13         cache.remove(cand.get(i));
14       }
15       cache.add(p);
16     }
▶      marked.add(p);
18   }
19   return faults;
20 }
```



faults = 5

p = 0 is cached → **hit**; it is just marked (•).

Step 7 / 14

VARIABLES

p = 0 | faults = 5

cache = [0, 2, 3]

marked = [0, 2, 3]

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (!cache.contains(p)) {
▶        faults++;
▶        if (cache.size() == k) {
▶          if (marked.size() == k) marked.clear(); // new
   phase
▶          List<Integer> cand = new ArrayList<>(cache);
▶          cand.removeAll(marked);       // unmarked ones
▶          int i = rng.nextInt(cand.size());
▶          cache.remove(cand.get(i));
14       }
▶        cache.add(p);
16     }
▶      marked.add(p);
18   }
19   return faults;
20 }
```



All pages were marked → **new phase**: marks are cleared. p = 4 missing → fault 6. **3** is evicted at random from the unmarked candidates [0, 2, 3].

Step 8 / 14

VARIABLES

`p = 4` `faults = 6`

`cache = [0, 2, 4]`

`marked = [4]`

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (!cache.contains(p)) {
7        faults++;
8        if (cache.size() == k) {
9          if (marked.size() == k) marked.clear(); // new
   phase
10         List<Integer> cand = new ArrayList<>(cache);
11         cand.removeAll(marked);       // unmarked ones
12         int i = rng.nextInt(cand.size());
13         cache.remove(cand.get(i));
14       }
15       cache.add(p);
16     }
▶      marked.add(p);
18   }
19   return faults;
20 }
```



faults = 6

p = 2 is cached → **hit**; it is just marked (•).

Step 9 / 14

VARIABLES

`p = 2` `faults = 6`

`cache = [0, 2, 4]`

`marked = [2, 4]`

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (!cache.contains(p)) {
▶        faults++;
▶        if (cache.size() == k) {
▶          if (marked.size() == k) marked.clear(); // new
   phase
▶          List<Integer> cand = new ArrayList<>(cache);
▶          cand.removeAll(marked);       // unmarked ones
▶          int i = rng.nextInt(cand.size());
▶          cache.remove(cand.get(i));
14       }
▶        cache.add(p);
16     }
▶      marked.add(p);
18   }
19   return faults;
20 }
```



candidates [0] → evict 0 — **faults = 7**

p = 3 missing → fault 7. **0** is evicted at random from the unmarked candidates [0].

Step 10 / 14

VARIABLES

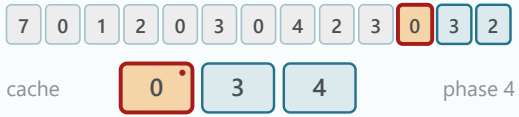
`p = 3` `faults = 7`

`cache = [2, 3, 4]`

`marked = [2, 3, 4]`

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
       if (!cache.contains(p)) {
         faults++;
         if (cache.size() == k) {
           if (marked.size() == k) marked.clear(); // new phase
           List<Integer> cand = new ArrayList<>(cache);
           cand.removeAll(marked);       // unmarked ones
           int i = rng.nextInt(cand.size());
           cache.remove(cand.get(i));
         }
         cache.add(p);
       }
       marked.add(p);
     }
     return faults;
   }
```



candidates [2, 3, 4] → evict 2

faults = 8

All pages were marked → **new phase**: marks are cleared. p = 0 missing → fault 8. **2** is evicted at random from the unmarked candidates [2, 3, 4].

Step 11 / 14

VARIABLES

p = 0 faults = 8

cache = [0, 3, 4]

marked = [0]

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
▶      if (!cache.contains(p)) {
7        faults++;
8        if (cache.size() == k) {
9          if (marked.size() == k) marked.clear(); // new
   phase
10         List<Integer> cand = new ArrayList<>(cache);
11         cand.removeAll(marked);       // unmarked ones
12         int i = rng.nextInt(cand.size());
13         cache.remove(cand.get(i));
14       }
15       cache.add(p);
16     }
▶      marked.add(p);
18   }
19   return faults;
20 }
```



faults = 8

p = 3 is cached → **hit**; it is just marked (•).

Step 12 / 14

VARIABLES

p = 3 | faults = 8

cache = [0, 3, 4]

marked = [0, 3]

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
       if (!cache.contains(p)) {
         faults++;
         if (cache.size() == k) {
           if (marked.size() == k) marked.clear(); // new phase
           List<Integer> cand = new ArrayList<>(cache);
           cand.removeAll(marked);       // unmarked ones
           int i = rng.nextInt(cand.size());
           cache.remove(cand.get(i));
         }
         cache.add(p);
       }
       marked.add(p);
     }
     return faults;
   }
```



p = 2 missing → fault 9. **4** is evicted at random from the unmarked candidates [4].

Step 13 / 14

VARIABLES

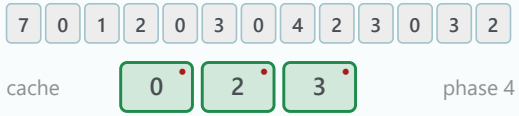
`p = 2` `faults = 9`

`cache = [0, 2, 3]`

`marked = [0, 2, 3]`

The Marking Algorithm: Code Trace

```
1  int marking(int[] requests, int k) {
2    Set<Integer> cache = new HashSet<>();
3    Set<Integer> marked = new HashSet<>();
4    int faults = 0;
5    for (int p : requests) {
6      if (!cache.contains(p)) {
7        faults++;
8        if (cache.size() == k) {
9          if (marked.size() == k) marked.clear(); // new
   phase
10         List<Integer> cand = new ArrayList<>(cache);
11         cand.removeAll(marked);       // unmarked ones
12         int i = rng.nextInt(cand.size());
13         cache.remove(cand.get(i));
14       }
15       cache.add(p);
16     }
17     marked.add(p);
18   }
▶  return faults;
20 }
```



`return 9`. With these draws marking made 9 faults (LRU 9, FIFO 10, OPT 7); another seed may give a different result.

Step 14 / 14

VARIABLES

`faults = 9` `returned = 9`

Try It: Paging Policies

requests `7 0 1 2 0 3 0 4 2 3 0 3 2` k 3 **FIFO** **LRU** **OPT** **Belady example**

Request	7	0	1	2	0	3	0	4	2	3	0	3	2
F1	7	7	7	2	2	2	2	4	4	4	0	0	0
F2		0	0	0	0	0	0	0	0	3	3	3	3
F3			1	1	1	3	3	3	2	2	2	2	2
Result	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓	✓

LRU shown · k = 3 · page faults: FIFO **10**, LRU **9**, OPT **7**

Digits 0–9 separated by spaces (at most 16 requests). Orange cell: newly loaded page · green: hit. Enter = run again.

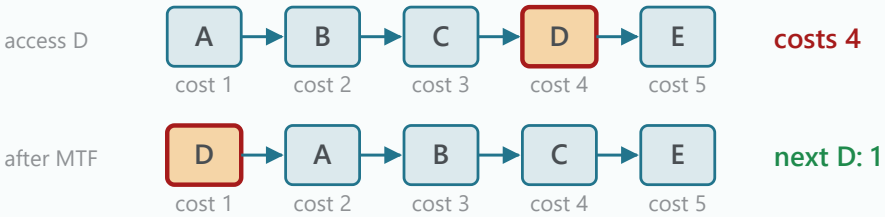
The List Update Problem

- A **linked list** of n elements; accessing the element at position i costs **i** (scanned from the head).
- After an access, the accessed element may be moved toward the front **for free**.
- Requests arrive online: which elements will be popular is unknown.

Move-to-front (MTF)
The accessed element is moved to the **front** of the list.

Transpose
The accessed element swaps places with the **preceding** one.

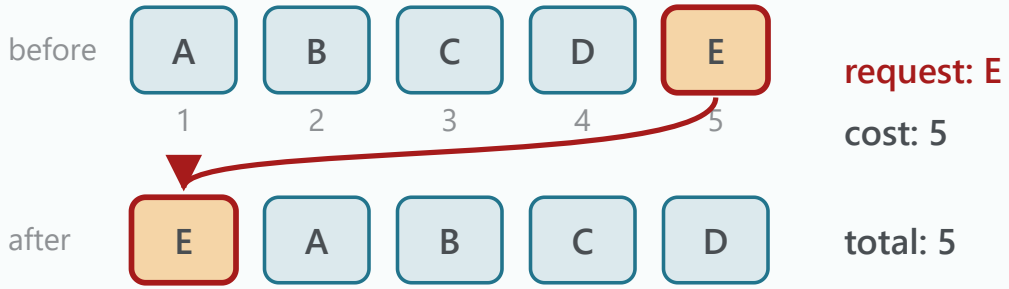
Frequency count
Elements are kept in **decreasing** order of access count.



Applications: self-organizing lists, chains in hash table buckets, data compression with the **MTF transform** (bzip2).

Move-to-Front (MTF) Step by Step

Initial list A B C D E; requests **E D E A D E**.



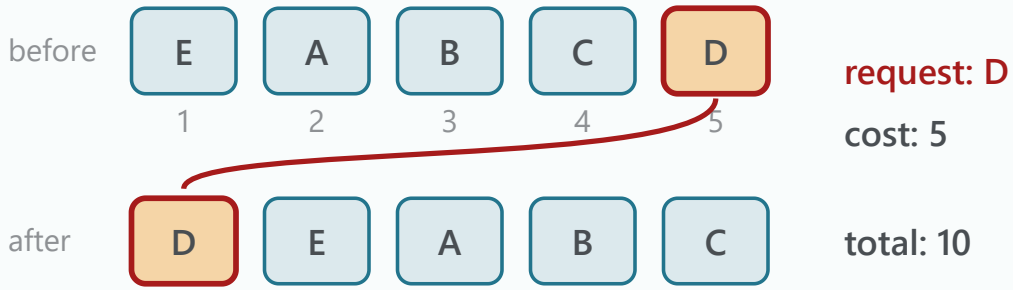
Request **E**: at position 5 → cost 5; **moved to the front**. Total: 5.

Step 1 / 6

Strategy	E	D	E	A	D	E	Total
Move-to-front (MTF)	5	5	2	3	3	3	**21**
No moves	5	4	5	1	4	5	**24**
Transpose	5	5	5	1	5	5	**26**

Move-to-Front (MTF) Step by Step

Initial list A B C D E; requests **E D E A D E**.

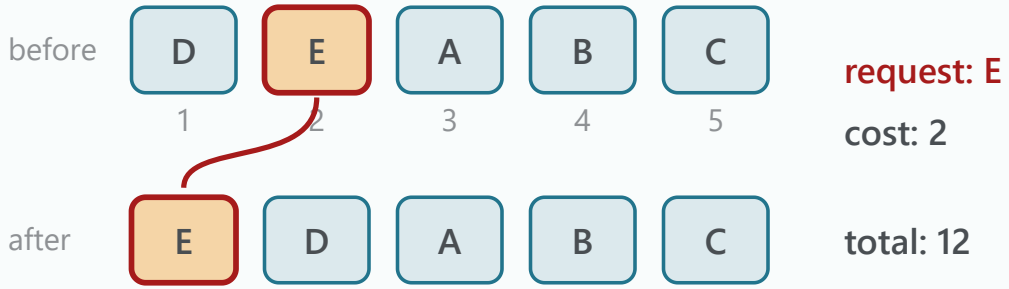


Request **D**: at position 5 → cost 5; **moved to the front**. Total: 10.

Strategy	E	D	E	A	D	E	Total
Move-to-front (MTF)	5	5	2	3	3	3	**21**
No moves	5	4	5	1	4	5	**24**
Transpose	5	5	5	1	5	5	**26**

Move-to-Front (MTF) Step by Step

Initial list A B C D E; requests **E D E A D E**.

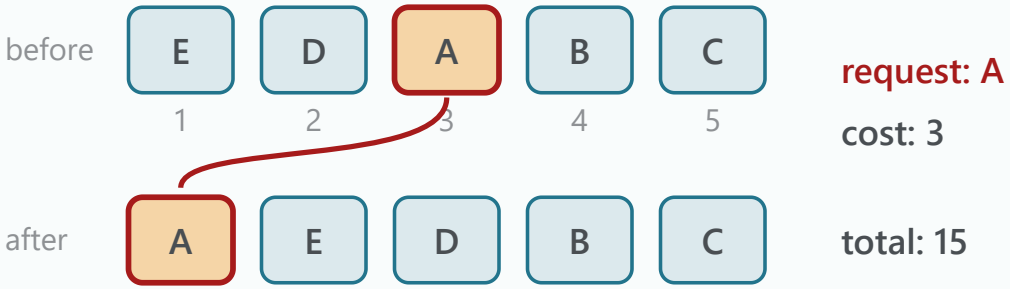


Request **E**: at position 2 → cost 2; **moved to the front**. Total: 12.

Strategy	E	D	E	A	D	E	Total
Move-to-front (MTF)	5	5	2	3	3	3	**21**
No moves	5	4	5	1	4	5	**24**
Transpose	5	5	5	1	5	5	**26**

Move-to-Front (MTF) Step by Step

Initial list A B C D E; requests **E D E A D E**.

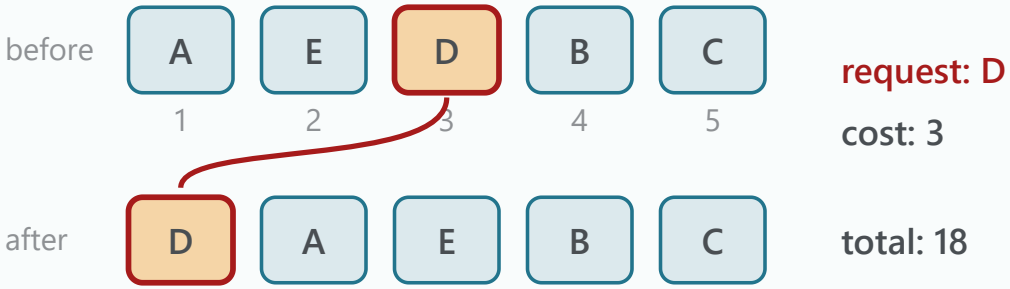


Request **A**: at position 3 → cost 3; **moved to the front**. Total: 15.

Strategy	E	D	E	A	D	E	Total
Move-to-front (MTF)	5	5	2	3	3	3	**21**
No moves	5	4	5	1	4	5	**24**
Transpose	5	5	5	1	5	5	**26**

Move-to-Front (MTF) Step by Step

Initial list A B C D E; requests **E D E A D E**.

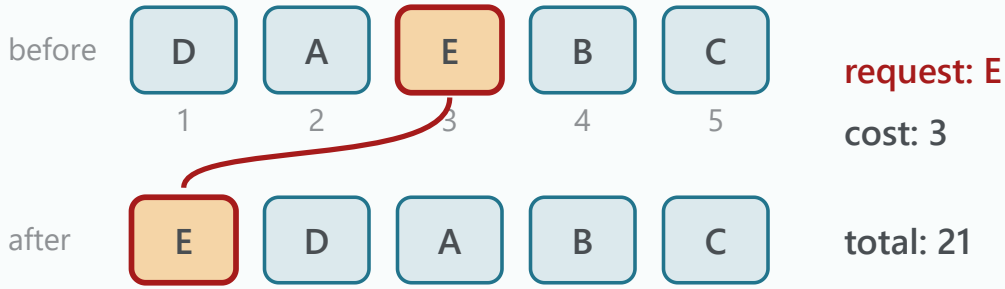


Request **D**: at position 3 → cost 3; **moved to the front**. Total: 18.

Strategy	E	D	E	A	D	E	Total
Move-to-front (MTF)	5	5	2	3	3	3	**21**
No moves	5	4	5	1	4	5	**24**
Transpose	5	5	5	1	5	5	**26**

Move-to-Front (MTF) Step by Step

Initial list A B C D E; requests **E D E A D E**.



Request **E**: at position 3 → cost 3; **moved to the front**. Total: 21. On the same sequence a list that never moves pays 24, and transpose pays 26.

Step 6 / 6

Strategy	E	D	E	A	D	E	Total
Move-to-front (MTF)	5	5	2	3	3	3	**21**
No moves	5	4	5	1	4	5	**24**
Transpose	5	5	5	1	5	5	**26**

Theorem: MTF Is 2-Competitive

- **Sleator and Tarjan (1985):** for every request sequence σ, $MTF(\sigma) \leq 2 \cdot OPT(\sigma) - m$ (m: number of requests; OPT's paid swaps count toward its cost).
- Proof idea: the **potential function** Φ = the number of **inversions** between the MTF list and the OPT list. On every access, MTF's actual cost + $\Delta\Phi \leq 2 \cdot$ (OPT's cost) − 1. This is amortized analysis.

Transpose is not competitive

- Request the last two elements alternately: x_n, x_{n-1}, x_n, … Each access swaps them, but they stay at positions n and n − 1 → ≈ n per request. MTF pays 2 per request after the first two.

Neither is frequency count

- Request x_1 n times, x_2 n − 1 times, …, then the same in reverse order. The counter keeps the old order for a long time; the cost can be a factor $\Theta(n)$ worse.

The Data Stream Model

- Elements x_1, x_2, … arrive one at a time; the stream may be very long or even **infinite**.
- Each element is seen **once** (a single pass); the past cannot be stored in full.
- An answer must be ready at any moment: "what is the average / median / a sample so far?"

Problem	Stored summary	Per element	Memory
Moving average (last k)	Queue + sum	O(1)	O(k)
Running median	Max-heap + min-heap	O(log n)	O(n)
Reservoir sampling	k-element sample	O(1)	O(k)
Pattern matching in a stream	Last m characters + rolling hash	O(1)	O(m)
Running mean	n and μ	O(1)	O(1)

Moving Average

- The average of the **last k elements** of a stream: smoothing sensor data, a 50-day moving average of a stock.
- The naive method sums k elements each time: **O(k)**.
- The smart method: a **sliding window**. Keep the sum; add the new element as it enters and subtract the oldest as it leaves: **O(1)**.

$$\text{sum} \leftarrow \text{sum} - x_{t-k} + x_t$$
$$\text{average} = \text{sum} / \min(t, k)$$

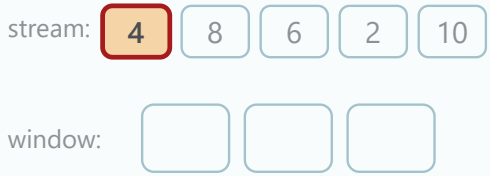
```
class MovingAverage {
  private final int k;
  private final Queue<Integer> window
      = new LinkedList<>();
  private int sum = 0;

  MovingAverage(int k) { this.k = k; }

  double next(int value) {
    if (window.size() == k)
      sum -= window.poll();
    window.offer(value);
    sum += value;
    return sum / (double) window.size();
  }
}
```

Moving Average: Code Trace

```
1 double next(int value) {
▶   if (window.size() == k)
3     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
6   return sum / (double) window.size();
7 }
```



sum = 0

New value 4. The window holds 0 elements (< 3) → nothing to remove.

Step 1 / 17

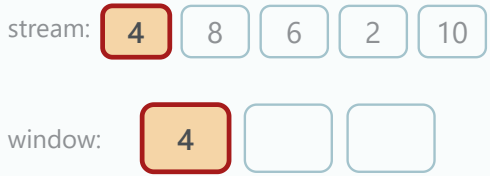
VARIABLES

`k = 3` `value = 4` `sum = 0` `window = []`

OUTPUT

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
3     sum -= window.poll(); // oldest
  ▶ window.offer(value);
  ▶ sum += value;
6   return sum / (double) window.size();
7 }
```



sum = 4

4 joins the queue and is added to the sum: sum = 4.

Step 2 / 17

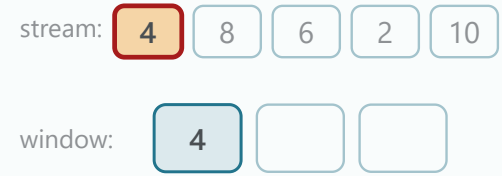
VARIABLES

`k = 3` `value = 4` `sum = 4` `window = [4]`

OUTPUT

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
3     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
▶   return sum / (double) window.size();
7 }
```



sum = 4

Average = 4 / 1 = **4**.

Step 3 / 17

VARIABLES

`k = 3` `value = 4` `sum = 4` `window = [4]`

OUTPUT

```
4.00
```

Moving Average: Code Trace

```
1 double next(int value) {
▶   if (window.size() == k)
3     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
6   return sum / (double) window.size();
7 }
```



sum = 4

New value 8. The window holds 1 element (< 3) → nothing to remove.

Step 4 / 17

VARIABLES

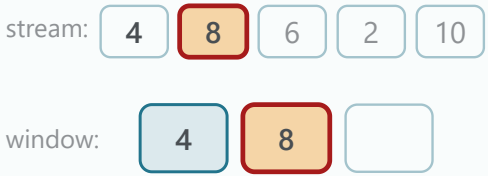
`k = 3` `value = 8` `sum = 4` `window = [4]`

OUTPUT

```
4.00
```

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
3     sum -= window.poll(); // oldest
  ▶ window.offer(value);
  ▶ sum += value;
6   return sum / (double) window.size();
7 }
```



sum = 12

8 joins the queue and is added to the sum: sum = 12.

Step 5 / 17

VARIABLES

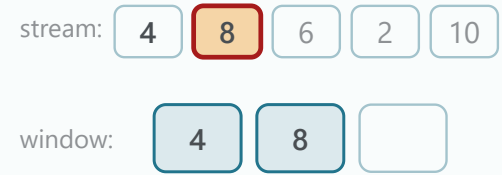
`k = 3` `value = 8` `sum = 12` `window = [4, 8]`

OUTPUT

```
4.00
```

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
3     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
▶   return sum / (double) window.size();
7 }
```



sum = 12

Average = 12 / 2 = **6**.

Step 6 / 17

VARIABLES

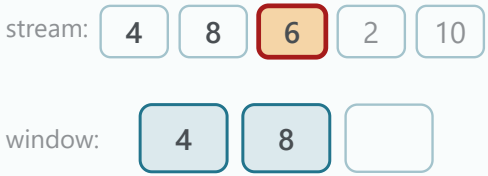
`k = 3` `value = 8` `sum = 12` `window = [4, 8]`

OUTPUT

```
4.00 6.00
```

Moving Average: Code Trace

```
1 double next(int value) {
▶   if (window.size() == k)
3     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
6   return sum / (double) window.size();
7 }
```



sum = 12

New value 6. The window holds 2 elements (< 3) → nothing to remove.

Step 7 / 17

VARIABLES

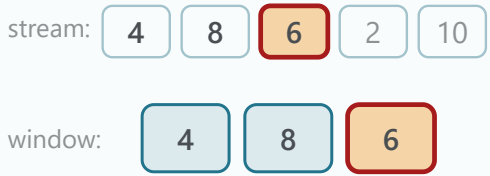
`k = 3` `value = 6` `sum = 12` `window = [4, 8]`

OUTPUT

```
4.00 6.00
```

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
3     sum -= window.poll(); // oldest
▶   window.offer(value);
▶   sum += value;
6   return sum / (double) window.size();
7 }
```



sum = 18

6 joins the queue and is added to the sum: sum = 18.

Step 8 / 17

VARIABLES

`k = 3` `value = 6` `sum = 18`

`window = [4, 8, 6]`

OUTPUT

```
4.00 6.00
```

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
3     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
▶   return sum / (double) window.size();
7 }
```





sum = 18

Average = 18 / 3 = **6**.

Step 9 / 17

VARIABLES

`k = 3` `value = 6` `sum = 18`

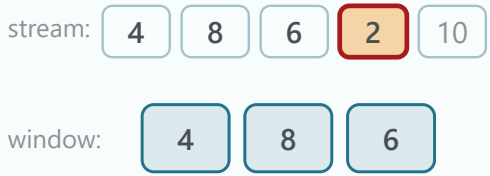
`window = [4, 8, 6]`

OUTPUT

```
4.00 6.00 6.00
```

Moving Average: Code Trace

```
1 double next(int value) {
▶   if (window.size() == k)
3     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
6   return sum / (double) window.size();
7 }
```



sum = 18

New value 2. The window is full (size 3 = k) → the oldest must go.

Step 10 / 17

VARIABLES

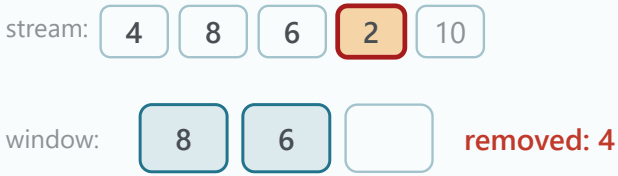
`k = 3` `value = 2` `sum = 18` `window = [4, 8, 6]`

OUTPUT

```
4.00 6.00 6.00
```

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
▶     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
6   return sum / (double) window.size();
7 }
```



The oldest element **4** leaves the queue and is subtracted from the sum: sum = 14.

Step 11 / 17

VARIABLES

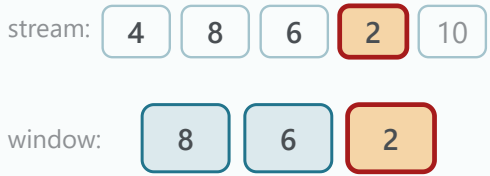
`k = 3` `value = 2` `sum = 14` `window = [8, 6]`

OUTPUT

```
4.00 6.00 6.00
```

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
3     sum -= window.poll(); // oldest
▶   window.offer(value);
▶   sum += value;
6   return sum / (double) window.size();
7 }
```



sum = 16

2 joins the queue and is added to the sum: sum = 16.

Step 12 / 17

VARIABLES

`k = 3` `value = 2` `sum = 16`

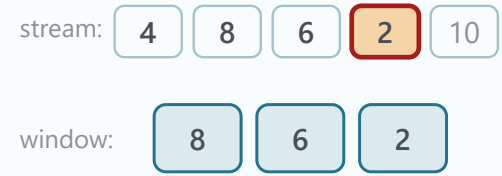
`window = [8, 6, 2]`

OUTPUT

```
4.00 6.00 6.00
```

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
3     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
▶   return sum / (double) window.size();
7 }
```



sum = 16

Average = 16 / 3 = **5.33**.

Step 13 / 17

VARIABLES

`k = 3` `value = 2` `sum = 16`

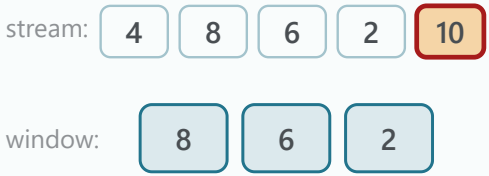
`window = [8, 6, 2]`

OUTPUT

```
4.00 6.00 6.00 5.33
```

Moving Average: Code Trace

```
1 double next(int value) {
▶   if (window.size() == k)
3     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
6   return sum / (double) window.size();
7 }
```



sum = 16

New value 10. The window is full (size 3 = k) → the oldest must go.

Step 14 / 17

VARIABLES

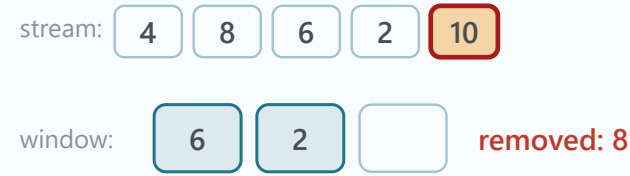
`k = 3` `value = 10` `sum = 16` `window = [8, 6, 2]`

OUTPUT

```
4.00 6.00 6.00 5.33
```

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
▶     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
6   return sum / (double) window.size();
7 }
```



The oldest element **8** leaves the queue and is subtracted from the sum: sum = 8.

Step 15 / 17

VARIABLES

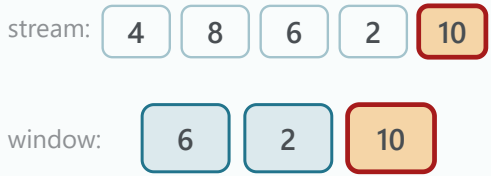
`k = 3` `value = 10` `sum = 8` `window = [6, 2]`

OUTPUT

```
4.00 6.00 6.00 5.33
```

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
3     sum -= window.poll(); // oldest
▶   window.offer(value);
▶   sum += value;
6   return sum / (double) window.size();
7 }
```



sum = 18

10 joins the queue and is added to the sum: sum = 18.

Step 16 / 17

VARIABLES

`k = 3` `value = 10` `sum = 18`

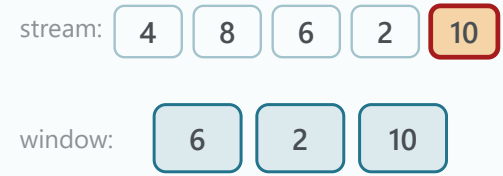
`window = [6, 2, 10]`

OUTPUT

```
4.00 6.00 6.00 5.33
```

Moving Average: Code Trace

```
1 double next(int value) {
2   if (window.size() == k)
3     sum -= window.poll(); // oldest
4   window.offer(value);
5   sum += value;
▶   return sum / (double) window.size();
7 }
```



Average = 18 / 3 = **6**.

Step 17 / 17

VARIABLES

`k = 3` `value = 10` `sum = 18` `window = [6, 2, 10]`

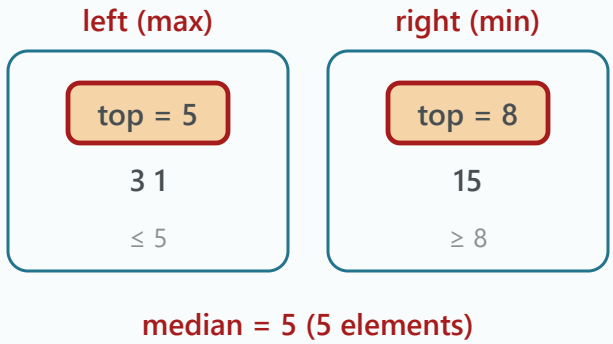
OUTPUT

```
4.00 6.00 6.00 5.33 6.00
```

DATA STREAMS

Running Median

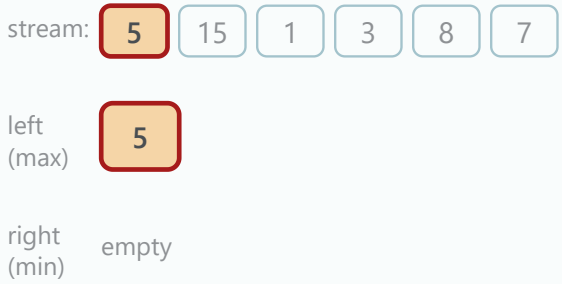
- After each new element we want **the median so far**.
- Sorting every time: O(n log n). Inserting into a sorted array: O(n).
- **Two heaps**: the smaller half in a **max-heap** (left), the larger half in a **min-heap** (right).
- Invariants: every element of left ≤ every element of right; |left| = |right| or |left| = |right| + 1.
- The median is read from the heap tops in **O(1)**; insertion takes **O(log n)**.



With an odd count the median is the top of left; with an even count it is the average of the two tops. A max-heap in Java: `new PriorityQueue<>((a, b) -> b - a)`.

Running Median: Code Trace

```
1  void add(int x) {
▶    if (left.isEmpty() || x <= left.peek())
▶      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



x = 5: left is empty → goes into the **left** heap.

Step 1 / 18

VARIABLES

`x = 5` `left = [5]` `right = []` `median = –`

OUTPUT

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
▶    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
▶    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



Sizes 1 and 0: balanced, no move.

Step 2 / 18

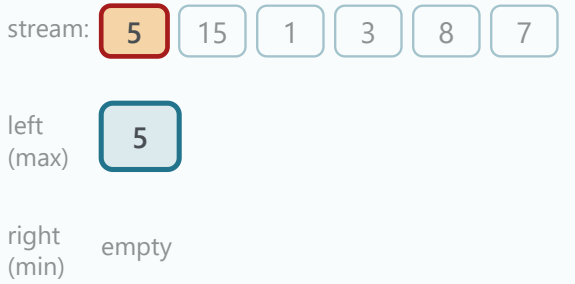
VARIABLES

`x = 5` `left = [5]` `right = []` `median = –`

OUTPUT

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
▶    if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
▶    return left.peek();
15 }
```



Odd count: median = top of left = **5**.

Step 3 / 18

VARIABLES

`x = 5` `left = [5]` `right = []`

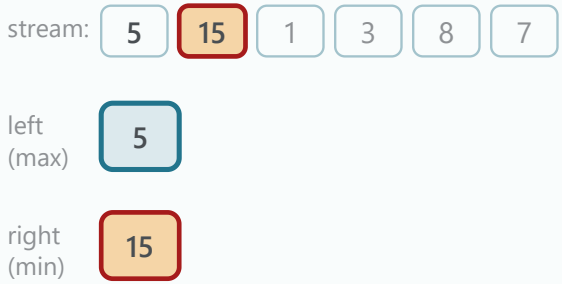
`median = 5`

OUTPUT

```
5.0
```

Running Median: Code Trace

```
1  void add(int x) {
▶    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
▶    else
▶      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



x = 15: 15 > 5 → goes into the **right** heap.

Step 4 / 18

VARIABLES

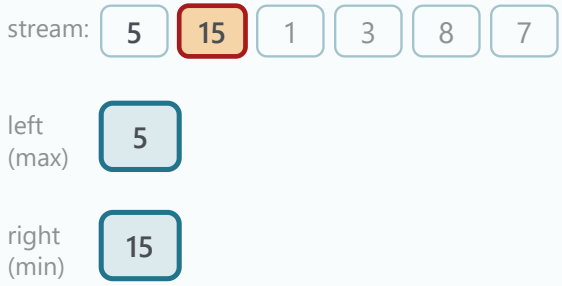
`x = 15` `left = [5]` `right = [15]` `median = –`

OUTPUT

```
5.0
```

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
▶    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
▶    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



Sizes 1 and 1: balanced, no move.

Step 5 / 18

VARIABLES

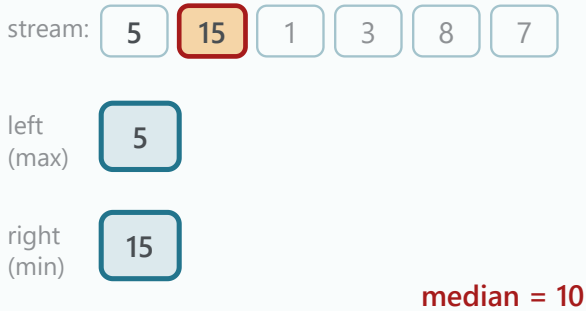
`x = 15` `left = [5]` `right = [15]` `median = –`

OUTPUT

```
5.0
```

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
▶      return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



Even count: median = (5 + 15) / 2 = **10**.

Step 6 / 18

VARIABLES

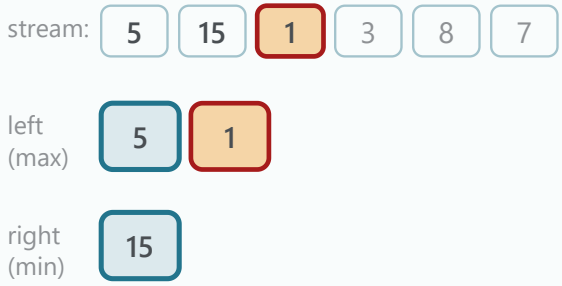
`x = 15` `left = [5]` `right = [15]` `median = 10`

OUTPUT

```
5.0 10.0
```

Running Median: Code Trace

```
1  void add(int x) {
▶    if (left.isEmpty() || x <= left.peek())
▶      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



x = 1: 1 ≤ 5 → goes into the **left** heap.

Step 7 / 18

VARIABLES

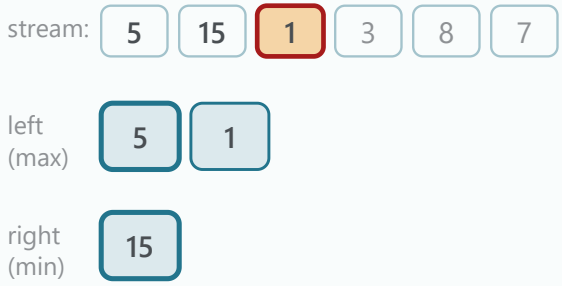
`x = 1` `left = [5, 1]` `right = [15]` `median = –`

OUTPUT

```
5.0 10.0
```

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
▶    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
▶    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



Sizes 2 and 1: balanced, no move.

Step 8 / 18

VARIABLES

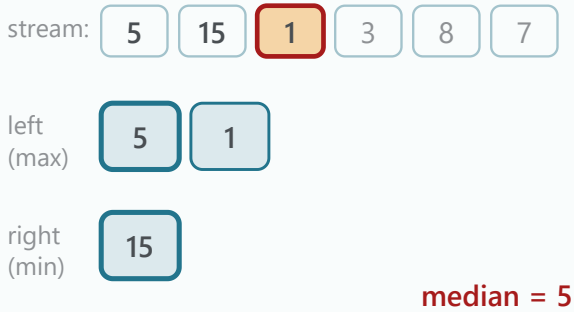
`x = 1` `left = [5, 1]` `right = [15]` `median = –`

OUTPUT

```
5.0 10.0
```

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
▶    if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
▶    return left.peek();
15 }
```



Odd count: median = top of left = **5**.

Step 9 / 18

VARIABLES

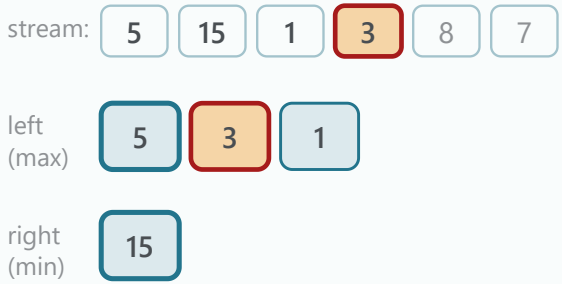
`x = 1` `left = [5, 1]` `right = [15]` `median = 5`

OUTPUT

```
5.0 10.0 5.0
```

Running Median: Code Trace

```
1  void add(int x) {
▶    if (left.isEmpty() || x <= left.peek())
▶      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



x = 3: 3 ≤ 5 → goes into the **left** heap.

Step 10 / 18

VARIABLES

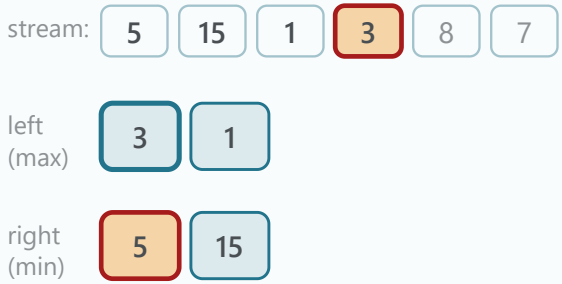
`x = 3` `left = [5, 3, 1]` `right = [15]` `median = –`

OUTPUT

```
5.0 10.0 5.0
```

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
▶    if (left.size() > right.size() + 1)
▶      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



Out of balance (|left| = 3, |right| = 1): the top of left, **5**, moves to the right heap.

Step 11 / 18

VARIABLES

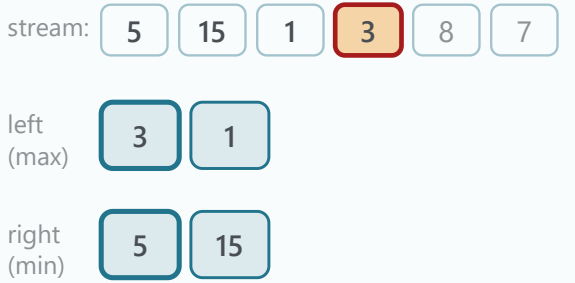
`x = 3` `left = [3, 1]` `right = [5, 15]` `median = –`

OUTPUT

```
5.0 10.0 5.0
```

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
▶      return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



Even count: median = (3 + 5) / 2 = **4**.

Step 12 / 18

VARIABLES

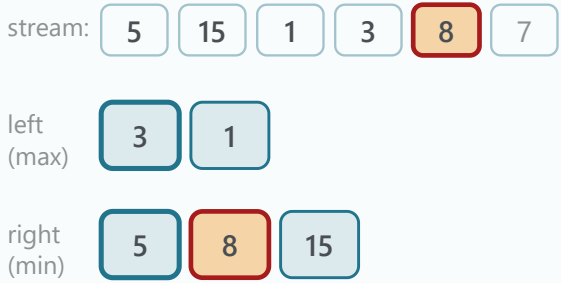
`x = 3` `left = [3, 1]` `right = [5, 15]` `median = 4`

OUTPUT

```
5.0 10.0 5.0 4.0
```

Running Median: Code Trace

```
1  void add(int x) {
▶    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
▶    else
▶      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



x = 8: 8 > 3 → goes into the **right** heap.

Step 13 / 18

VARIABLES

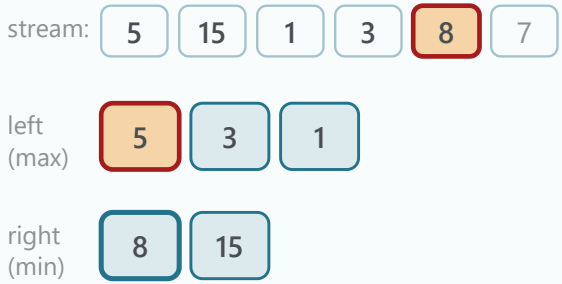
`x = 8` `left = [3, 1]` `right = [5, 8, 15]` `median = –`

OUTPUT

```
5.0 10.0 5.0 4.0
```

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
▶    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
▶    else if (right.size() > left.size())
▶      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



Out of balance (|left| = 2, |right| = 3): the top of right, **5**, moves to the left heap.

Step 14 / 18

VARIABLES

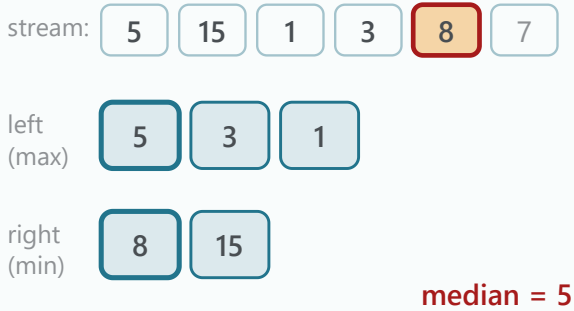
`x = 8` `left = [5, 3, 1]` `right = [8, 15]` `median = –`

OUTPUT

```
5.0 10.0 5.0 4.0
```

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
▶    if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
▶    return left.peek();
15 }
```



Odd count: median = top of left = **5**.

Step 15 / 18

VARIABLES

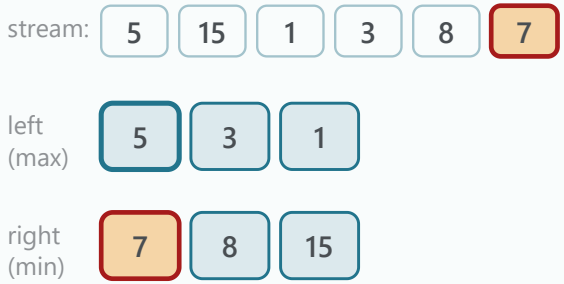
`x = 8` `left = [5, 3, 1]` `right = [8, 15]` `median = 5`

OUTPUT

```
5.0 10.0 5.0 4.0 5.0
```

Running Median: Code Trace

```
1  void add(int x) {
▶    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
▶    else
▶      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



x = 7: 7 > 5 → goes into the **right** heap.

Step 16 / 18

VARIABLES

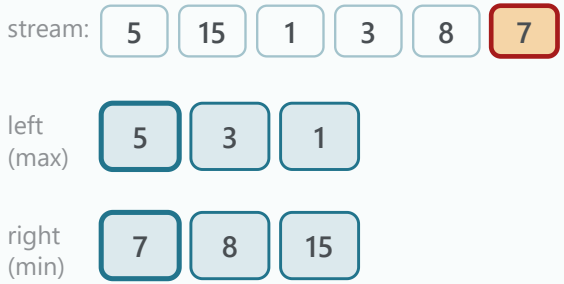
`x = 7` `left = [5, 3, 1]` `right = [7, 8, 15]` `median = –`

OUTPUT

```
5.0 10.0 5.0 4.0 5.0
```

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
▶    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
▶    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
13     return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



Sizes 3 and 3: balanced, no move.

Step 17 / 18

VARIABLES

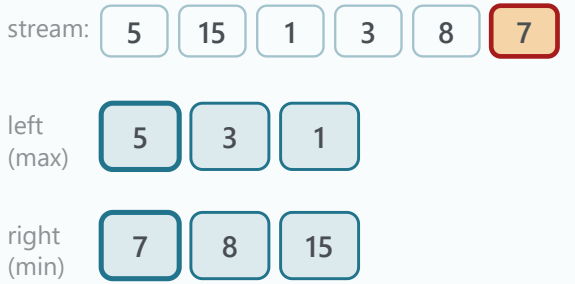
`x = 7` `left = [5, 3, 1]` `right = [7, 8, 15]` `median = –`

OUTPUT

```
5.0 10.0 5.0 4.0 5.0
```

Running Median: Code Trace

```
1  void add(int x) {
2    if (left.isEmpty() || x <= left.peek())
3      left.offer(x);          // smaller half
4    else
5      right.offer(x);         // larger half
6    if (left.size() > right.size() + 1)
7      right.offer(left.poll());
8    else if (right.size() > left.size())
9      left.offer(right.poll());
10 }
11 double median() {
12   if (left.size() == right.size())
▶      return (left.peek() + right.peek()) / 2.0;
14   return left.peek();
15 }
```



Even count: median = (5 + 7) / 2 = **6**.

Step 18 / 18

VARIABLES

`x = 7` `left = [5, 3, 1]` `right = [7, 8, 15]` `median = 6`

OUTPUT

```
5.0 10.0 5.0 4.0 5.0 6.0
```

Reservoir Sampling

- Maintain a sample of k elements from a stream of **unknown** length, with every element **equally likely** to be chosen.
- **Algorithm R** (Vitter, 1985): put the first k elements into the reservoir (if the stream is shorter than k, return them all). Then for the i-th element (0-based, $i \geq k$), pick $j = \text{random}[0, i]$; if **$j < k$**, set $\text{reservoir}[j] \leftarrow x_i$.
- The i-th element enters with probability **$k / (i + 1)$**; if it does, it replaces an element at a random slot.

Proof (after n elements, each is in the reservoir with probability k / n): x_i enters with $k/(i+1)$. At each later step $t > i$ it survives with probability $1 - (k/(t+1)) \cdot (1/k) = t/(t+1)$. The product:

$$k/(i+1) \cdot (i+1)/(i+2) \cdot \ldots \cdot (n-1)/n = \mathbf{k / n}$$

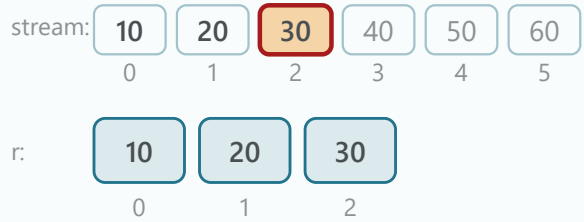
2/3	·	3/4	·	4/5	=	2/5
enters		t = 3 stays		t = 4 stays		k / n

Example: k = 2, n = 5, element x_2.

The same holds for the first k elements: they enter with probability 1, and the product $k/(k+1) \cdot \ldots \cdot (n-1)/n = k/n$. Memory O(k), O(1) per element.

Reservoir Sampling: Code Trace

```
1  List<Integer> reservoir(int[] stream, int k) {
▶    List<Integer> r = new ArrayList<>();
▶    for (int i = 0; i < Math.min(k,
   stream.length); i++)
▶      r.add(stream[i]);
5    for (int i = k; i < stream.length; i++) {
6      int j = rng.nextInt(i + 1); // 0..i
7      if (j < k)
8        r.set(j, stream[i]);
9    }
10   return r;
11 }
```



The first k = 3 elements (10, 20, 30) go straight into the reservoir.

Step 1 / 8

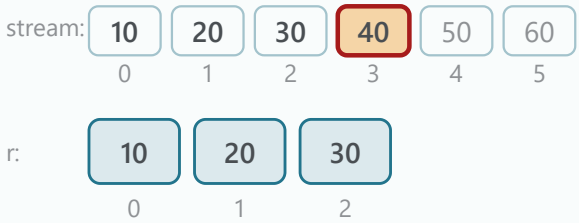
VARIABLES

`k = 3` `i = –` `j = –` `prob = –`

`r = [10, 20, 30]`

Reservoir Sampling: Code Trace

```
1  List<Integer> reservoir(int[] stream, int k) {
2    List<Integer> r = new ArrayList<>();
3    for (int i = 0; i < Math.min(k,
   stream.length); i++)
4      r.add(stream[i]);
▶    for (int i = k; i < stream.length; i++) {
▶      int j = rng.nextInt(i + 1); // 0..i
7      if (j < k)
8        r.set(j, stream[i]);
9    }
10   return r;
11 }
```



i = 3 (value 40): j = random[0, 3] → **j = 1** (fixed for this trace).

Step 2 / 8

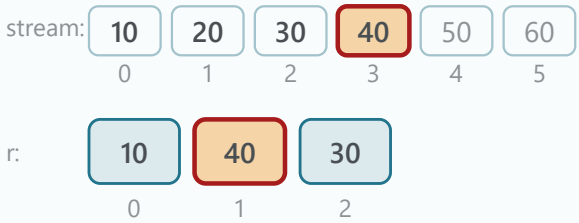
VARIABLES

`k = 3` `i = 3` `j = 1` `prob = 3/4`

`r = [10, 20, 30]`

Reservoir Sampling: Code Trace

```
1  List<Integer> reservoir(int[] stream, int k) {
2    List<Integer> r = new ArrayList<>();
3    for (int i = 0; i < Math.min(k,
   stream.length); i++)
4      r.add(stream[i]);
5    for (int i = k; i < stream.length; i++) {
6      int j = rng.nextInt(i + 1); // 0..i
▶      if (j < k)
▶        r.set(j, stream[i]);
9    }
10   return r;
11 }
```



j = 1 < 3 → r[1] = 40 (20 leaves). 40 entered with probability 3/4.

Step 3 / 8

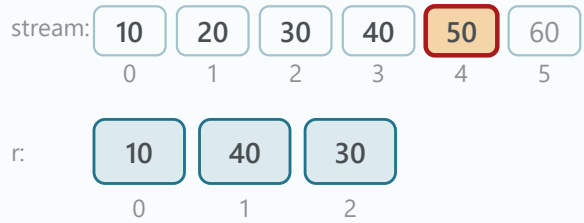
VARIABLES

`k = 3` `i = 3` `j = 1` `prob = 3/4`

`r = [10, 40, 30]`

Reservoir Sampling: Code Trace

```
1  List<Integer> reservoir(int[] stream, int k) {
2    List<Integer> r = new ArrayList<>();
3    for (int i = 0; i < Math.min(k,
   stream.length); i++)
4      r.add(stream[i]);
▶    for (int i = k; i < stream.length; i++) {
▶      int j = rng.nextInt(i + 1); // 0..i
7      if (j < k)
8        r.set(j, stream[i]);
9    }
10   return r;
11 }
```



i = 4 (value 50): j = random[0, 4] → **j = 4** (fixed for this trace).

Step 4 / 8

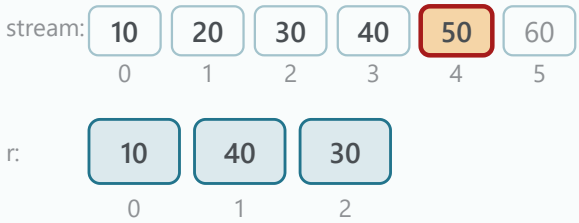
VARIABLES

`k = 3` `i = 4` `j = 4` `prob = 3/5`

`r = [10, 40, 30]`

Reservoir Sampling: Code Trace

```
1  List<Integer> reservoir(int[] stream, int k) {
2    List<Integer> r = new ArrayList<>();
3    for (int i = 0; i < Math.min(k,
   stream.length); i++)
4      r.add(stream[i]);
5    for (int i = k; i < stream.length; i++) {
6      int j = rng.nextInt(i + 1); // 0..i
▶      if (j < k)
8        r.set(j, stream[i]);
9    }
10   return r;
11 }
```



j = 4 ≥ 3 → 50 does **not enter**; the reservoir is unchanged.

Step 5 / 8

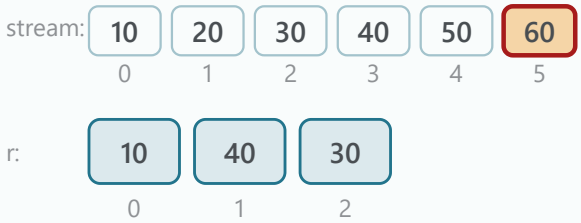
VARIABLES

`k = 3` `i = 4` `j = 4` `prob = 3/5`

`r = [10, 40, 30]`

Reservoir Sampling: Code Trace

```
1  List<Integer> reservoir(int[] stream, int k) {
2    List<Integer> r = new ArrayList<>();
3    for (int i = 0; i < Math.min(k,
   stream.length); i++)
4      r.add(stream[i]);
▶    for (int i = k; i < stream.length; i++) {
▶      int j = rng.nextInt(i + 1); // 0..i
7      if (j < k)
8        r.set(j, stream[i]);
9    }
10   return r;
11 }
```



i = 5 (value 60): j = random[0, 5] → **j = 0** (fixed for this trace).

Step 6 / 8

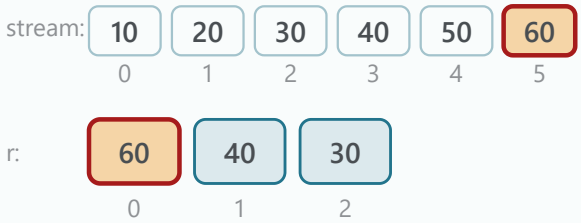
VARIABLES

`k = 3` `i = 5` `j = 0` `prob = 3/6`

`r = [10, 40, 30]`

Reservoir Sampling: Code Trace

```
1  List<Integer> reservoir(int[] stream, int k) {
2    List<Integer> r = new ArrayList<>();
3    for (int i = 0; i < Math.min(k,
   stream.length); i++)
4      r.add(stream[i]);
5    for (int i = k; i < stream.length; i++) {
6      int j = rng.nextInt(i + 1); // 0..i
▶      if (j < k)
▶        r.set(j, stream[i]);
9    }
10   return r;
11 }
```



j = 0 < 3 → r[0] = 60 (10 leaves). 60 entered with probability 3/6.

Step 7 / 8

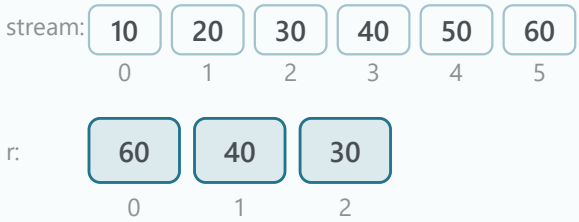
VARIABLES

`k = 3` `i = 5` `j = 0` `prob = 3/6`

`r = [60, 40, 30]`

Reservoir Sampling: Code Trace

```
1  List<Integer> reservoir(int[] stream, int k) {
2    List<Integer> r = new ArrayList<>();
3    for (int i = 0; i < Math.min(k,
   stream.length); i++)
4      r.add(stream[i]);
5    for (int i = k; i < stream.length; i++) {
6      int j = rng.nextInt(i + 1); // 0..i
7      if (j < k)
8        r.set(j, stream[i]);
9    }
▶    return r;
11 }
```



Stream done: the sample is [60, 40, 30]. Other values of j give other results; each element is chosen with probability 3/6 = 1/2.

Step 8 / 8

VARIABLES

`k = 3` `i = 6` `j = –` `prob = –`

`r = [60, 40, 30]`

Pattern Matching in a Stream: Rolling Hash

```
1  void search(String stream, String pattern) {
▶    int m = pattern.length(), B = 101, h = 0, hp = 0;
▶    int pow = 1;                          // B^(m-1)
▶    for (int i = 0; i < m - 1; i++) pow *= B;
▶    for (int i = 0; i < m; i++)
▶      hp = hp * B + pattern.charAt(i);
7    for (int t = 0; t < stream.length(); t++) {
8      char c = stream.charAt(t);
9      if (t >= m) h -= stream.charAt(t - m) * pow;
10     h = h * B + c;           // add incoming
11     if (t >= m - 1 && h == hp
12         && stream.startsWith(pattern, t - m + 1))
13       System.out.println("match: " + (t - m + 1));
14   }
15 }
```



h = 0 hp = 6631 ("AB")

Pattern "AB", m = 2, B = 101. pow = 101^{m-1} = 101, hp = 65 · 101 + 66 = **6631**. The stream "CABDAB" arrives one character at a time; for each new character $h_{new} = (h_{old} - c_{out} \cdot B^{m-1}) \cdot B + c_{in}$: **O(1)** instead of O(m) from scratch.

Step 1 / 7

VARIABLES

`m = 2` `pow = 101` `hp = 6631` `h = 0`

OUTPUT

Pattern Matching in a Stream: Rolling Hash

```
1  void search(String stream, String pattern) {
2    int m = pattern.length(), B = 101, h = 0, hp = 0;
3    int pow = 1;                              // B^(m-1)
4    for (int i = 0; i < m - 1; i++) pow *= B;
5    for (int i = 0; i < m; i++)
6      hp = hp * B + pattern.charAt(i);
7    for (int t = 0; t < stream.length(); t++) {
▶      char c = stream.charAt(t);
9      if (t >= m) h -= stream.charAt(t - m) * pow;
▶      h = h * B + c;           // add incoming
11     if (t >= m - 1 && h == hp
12         && stream.startsWith(pattern, t - m + 1))
13       System.out.println("match: " + (t - m + 1));
14   }
15 }
```



h = 67 hp = 6631 ("AB")

0·101 + 67

t = 0, c = C (67): the window is not full yet; h = 67.

Step 2 / 7

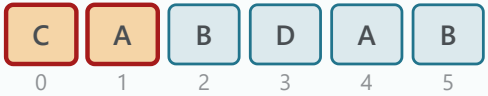
VARIABLES

t = 0 c = C h = 67 hp = 6631

OUTPUT

Pattern Matching in a Stream: Rolling Hash

```
1  void search(String stream, String pattern) {
2    int m = pattern.length(), B = 101, h = 0, hp = 0;
3    int pow = 1;                              // B^(m-1)
4    for (int i = 0; i < m - 1; i++) pow *= B;
5    for (int i = 0; i < m; i++)
6      hp = hp * B + pattern.charAt(i);
7    for (int t = 0; t < stream.length(); t++) {
▶      char c = stream.charAt(t);
9      if (t >= m) h -= stream.charAt(t - m) * pow;
▶      h = h * B + c;           // add incoming
11     if (t >= m - 1 && h == hp
12         && stream.startsWith(pattern, t - m + 1))
13       System.out.println("match: " + (t - m + 1));
14   }
15 }
```



h = 6832 hp = 6631 ("AB")

67·101 + 65

t = 1: the window is full: "CA". h = 6832 ≠ hp → no match.

Step 3 / 7

VARIABLES

t = 1 c = A h = 6832 hp = 6631

OUTPUT

Pattern Matching in a Stream: Rolling Hash

```
1  void search(String stream, String pattern) {
2    int m = pattern.length(), B = 101, h = 0, hp = 0;
3    int pow = 1;                              // B^(m-1)
4    for (int i = 0; i < m - 1; i++) pow *= B;
5    for (int i = 0; i < m; i++)
6      hp = hp * B + pattern.charAt(i);
7    for (int t = 0; t < stream.length(); t++) {
▶      char c = stream.charAt(t);
▶      if (t >= m) h -= stream.charAt(t - m) * pow;
▶      h = h * B + c;          // add incoming
▶      if (t >= m - 1 && h == hp
▶          && stream.startsWith(pattern, t - m + 1))
▶        System.out.println("match: " + (t - m + 1));
14   }
15 }
```



h = 6631 hp = 6631 ("AB")

(6832 − 67·101)·101 + 66

t = 2: C leaves (subtract 67 · 101), B enters. h = 6631 = hp and the characters agree → **match: 1**.

Step 4 / 7

VARIABLES

t = 2 c = B h = 6631 hp = 6631

OUTPUT

```
match: 1
```

Pattern Matching in a Stream: Rolling Hash

```
1  void search(String stream, String pattern) {
2    int m = pattern.length(), B = 101, h = 0, hp = 0;
3    int pow = 1;                              // B^(m-1)
4    for (int i = 0; i < m - 1; i++) pow *= B;
5    for (int i = 0; i < m; i++)
6      hp = hp * B + pattern.charAt(i);
7    for (int t = 0; t < stream.length(); t++) {
▶      char c = stream.charAt(t);
▶      if (t >= m) h -= stream.charAt(t - m) * pow;
▶      h = h * B + c;            // add incoming
11     if (t >= m - 1 && h == hp
12         && stream.startsWith(pattern, t - m + 1))
13       System.out.println("match: " + (t - m + 1));
14   }
15 }
```



h = 6734 **hp = 6631 ("AB")**

(6631 − 65·101)·101 + 68

t = 3: A leaves (subtract 65 · 101), D enters. h = 6734 ≠ hp → no match.

Step 5 / 7

VARIABLES

`t = 3` `c = D` `h = 6734` `hp = 6631`

OUTPUT

```
match: 1
```

Pattern Matching in a Stream: Rolling Hash

```
1  void search(String stream, String pattern) {
2    int m = pattern.length(), B = 101, h = 0, hp = 0;
3    int pow = 1;                              // B^(m-1)
4    for (int i = 0; i < m - 1; i++) pow *= B;
5    for (int i = 0; i < m; i++)
6      hp = hp * B + pattern.charAt(i);
7    for (int t = 0; t < stream.length(); t++) {
▶      char c = stream.charAt(t);
▶      if (t >= m) h -= stream.charAt(t - m) * pow;
▶      h = h * B + c;            // add incoming
11     if (t >= m - 1 && h == hp
12         && stream.startsWith(pattern, t - m + 1))
13       System.out.println("match: " + (t - m + 1));
14   }
15 }
```



h = 6933 **hp = 6631 ("AB")**

(6734 − 66·101)·101 + 65

t = 4: B leaves (subtract 66 · 101), A enters. h = 6933 ≠ hp → no match.

Step 6 / 7

VARIABLES

`t = 4` `c = A` `h = 6933` `hp = 6631`

OUTPUT

```
match: 1
```

Pattern Matching in a Stream: Rolling Hash

```
1  void search(String stream, String pattern) {
2    int m = pattern.length(), B = 101, h = 0, hp = 0;
3    int pow = 1;                              // B^(m-1)
4    for (int i = 0; i < m - 1; i++) pow *= B;
5    for (int i = 0; i < m; i++)
6      hp = hp * B + pattern.charAt(i);
7    for (int t = 0; t < stream.length(); t++) {
▶      char c = stream.charAt(t);
▶      if (t >= m) h -= stream.charAt(t - m) * pow;
▶      h = h * B + c;           // add incoming
▶      if (t >= m - 1 && h == hp
▶          && stream.startsWith(pattern, t - m + 1))
▶        System.out.println("match: " + (t - m + 1));
14   }
15 }
```



h = 6631 **hp = 6631 ("AB")**

(6933 − 68·101)·101 + 66

t = 5: D leaves (subtract 68 · 101), B enters. h = 6631 = hp and the characters agree → **match: 4**. Equal hashes are not enough: collisions are possible, so the characters are compared too.

Step 7 / 7

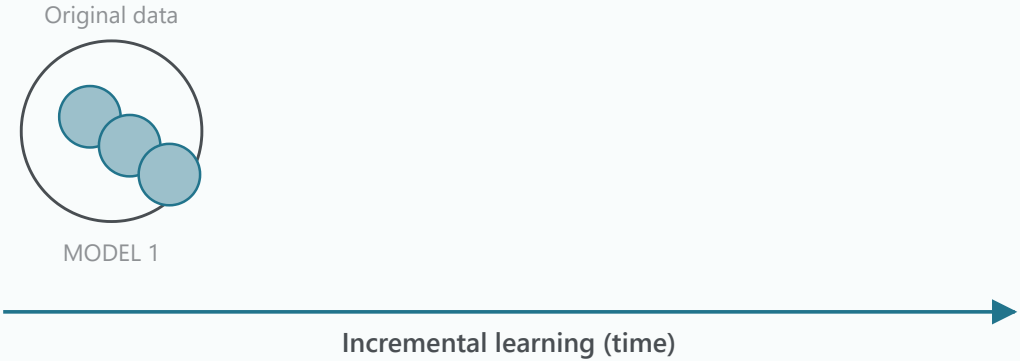
VARIABLES

`t = 5` `c = B` `h = 6631` `hp = 6631`

OUTPUT

```
match: 1
match: 4
```

Incremental Learning



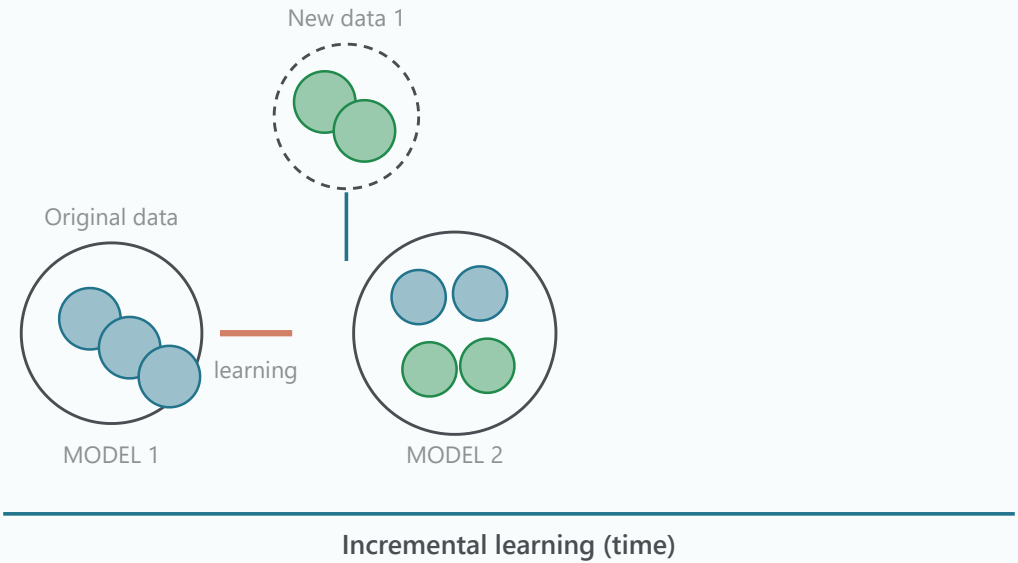
At first the model is trained on the **original data set**: MODEL 1.

Incremental Learning



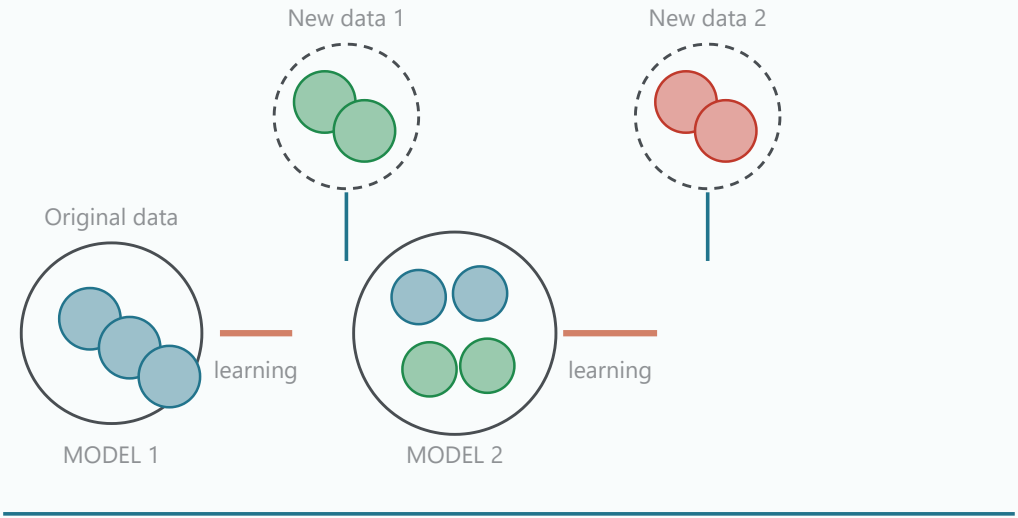
New data 1 arrives. Instead of retraining on all the data, the model is **updated** with the new data only.

Incremental Learning



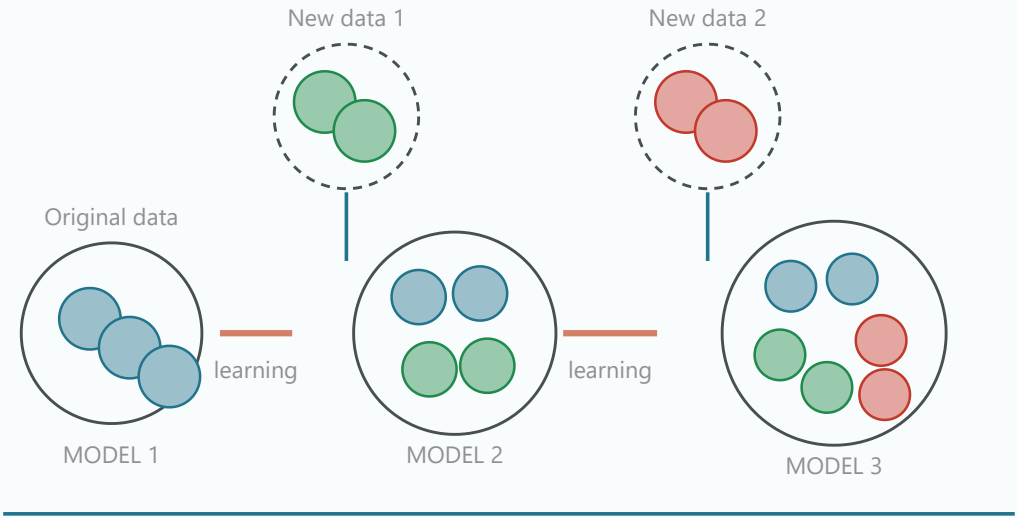
MODEL 2 captures the information in both old and new data; the old data is **not processed again**.

Incremental Learning



New data 2 arrives; the same update is repeated.

Incremental Learning



MODEL 3 carries the information of all three data sets. The model is always **up to date**, and each update costs only as much as the new data.

Incremental Updates and Their Challenges

Running mean

The mean is updated with each new element; no need to store the past:

$$\mu_n = \mu_{n-1} + (x_n - \mu_{n-1}) / n$$

Stream 4, 8, 6: $\mu = 4 \to 4 + (8 - 4)/2 = 6 \to 6 + (6 - 6)/3 =$ **6**.

Online gradient descent

$$w \leftarrow w - \eta \cdot \nabla \ell(w; x_t, y_t)$$

Each example is used once; η is the learning rate.

Challenges

- **Catastrophic forgetting**: old knowledge is lost while adapting to new data.
- **Concept drift**: the data distribution changes over time; old data can mislead.
- **Order effects**: the model is shaped by the order in which data arrives.
- **Noise**: a single outlier can corrupt the model; a small η and a sliding window help.

SUMMARY

Problems and Results

Problem	Online algorithm	Result	Note
Ski rental	Rent for B − 1 days, then buy	2 − 1/B	best deterministic; randomized e/(e−1)
Paging	LRU, FIFO, FWF	k	best deterministic
Paging	LFU, LIFO	unbounded	not competitive
Paging (randomized)	Marking	$2H_k$	lower bound H_k
List update	Move-to-front (MTF)	2	transpose and frequency count are not competitive
Moving average	Queue + sum	O(1)	memory O(k)
Running median	Two heaps	O(log n)	median query O(1)
Reservoir sampling	Algorithm R	O(1)	each element with probability k/n

The numbers in the top five rows are competitive ratios; those below are time per element.

Summary

Topic	Key points
Model	the input arrives piece by piece; decisions are immediate and irrevocable
Competitive ratio	$ALG(\sigma) \le c \cdot OPT(\sigma) + \alpha$ for every σ; worst-case analysis, adversary model
Ski rental	$OPT = \min(d, B)$; break-even achieves $2 - 1/B$; no deterministic strategy does better
Paging	FIFO and LRU are k-competitive; OPT evicts the page needed farthest in the future; Belady's anomaly in FIFO
List update	cost = position; MTF is 2-competitive (potential function: inversions)
Data streams	single pass, little memory: sliding window, two heaps, reservoir, rolling hash
Incremental learning	the model is updated with new data; challenges: forgetting and concept drift

SUMMARY

Common Mistakes

Thinking "online = internet"

An online algorithm works in a model where the input is **revealed over time**; it has nothing to do with network connections.

Confusing the ratio with the average case

The competitive ratio refers to the **worst** input; "usually works well" is not a competitive ratio.

Mixing up FIFO and LRU

In FIFO a hit does **not change** the order; in LRU a hit makes the page the most recent.

Forgetting compulsory faults

The first k distinct pages fault under every policy; even OPT cannot achieve 0 faults.

Wrong range in the reservoir

j must be random[0, i] (i included, 0-based). Writing `nextInt(i)` breaks the probabilities.

Forgetting to rebalance the median heaps

Inserting is not enough; when the size difference exceeds 1, a top must move to the other heap.

SUMMARY

Sample Problems

1. How many page faults does OPT make on the request sequence **1 2 3 4 1 2 5 1 2 3 4 5** with k = 3?
2. In ski rental B = 20. What is the ratio of the break-even strategy? What is the worst-case ratio of the strategy that buys on day 5?
3. On the list A B C D, what is the total MTF cost of the requests **D C D A**?
4. Write down the running medians for the stream 2, 9, 4, 7.
5. In reservoir sampling with k = 2, with what probability does element 9 (0-based) enter the reservoir? If the stream ends after 10 elements, with what probability is this element in the final reservoir?

▶ **Show answers**

Exercises

1. Implement FIFO paging in O(1) time with a `Queue` and a `HashSet`.
2. Implement OPT (Belady) in Java. What is its time complexity?
3. Find the ratio of the "buy on day B/2" strategy for ski rental.
4. Why is adding `remove(x)` to the running median hard?
5. Show that with k = 1, each element of a 3-element stream is chosen with probability 1/3.
6. Find the **maximum** of the last k elements in amortized O(1) time.

▶ **Answers / hints**

References

- A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, 1998.
- D. D. Sleator and R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules", *Communications of the ACM* 28(2), 1985.
- L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer", *IBM Systems Journal* 5(2), 1966.
- A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator and N. E. Young, "Competitive Paging Algorithms", *Journal of Algorithms* 12(4), 1991.
- J. S. Vitter, "Random Sampling with a Reservoir", *ACM Transactions on Mathematical Software* 11(1), 1985.
- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., 2022 (Chapter 27: Online Algorithms).

The End

Chapter 8: Online Algorithms

Next chapter: Chapter 9: Parallel Algorithms