Algorithms for Decision Support

Online Algorithms (3/3)

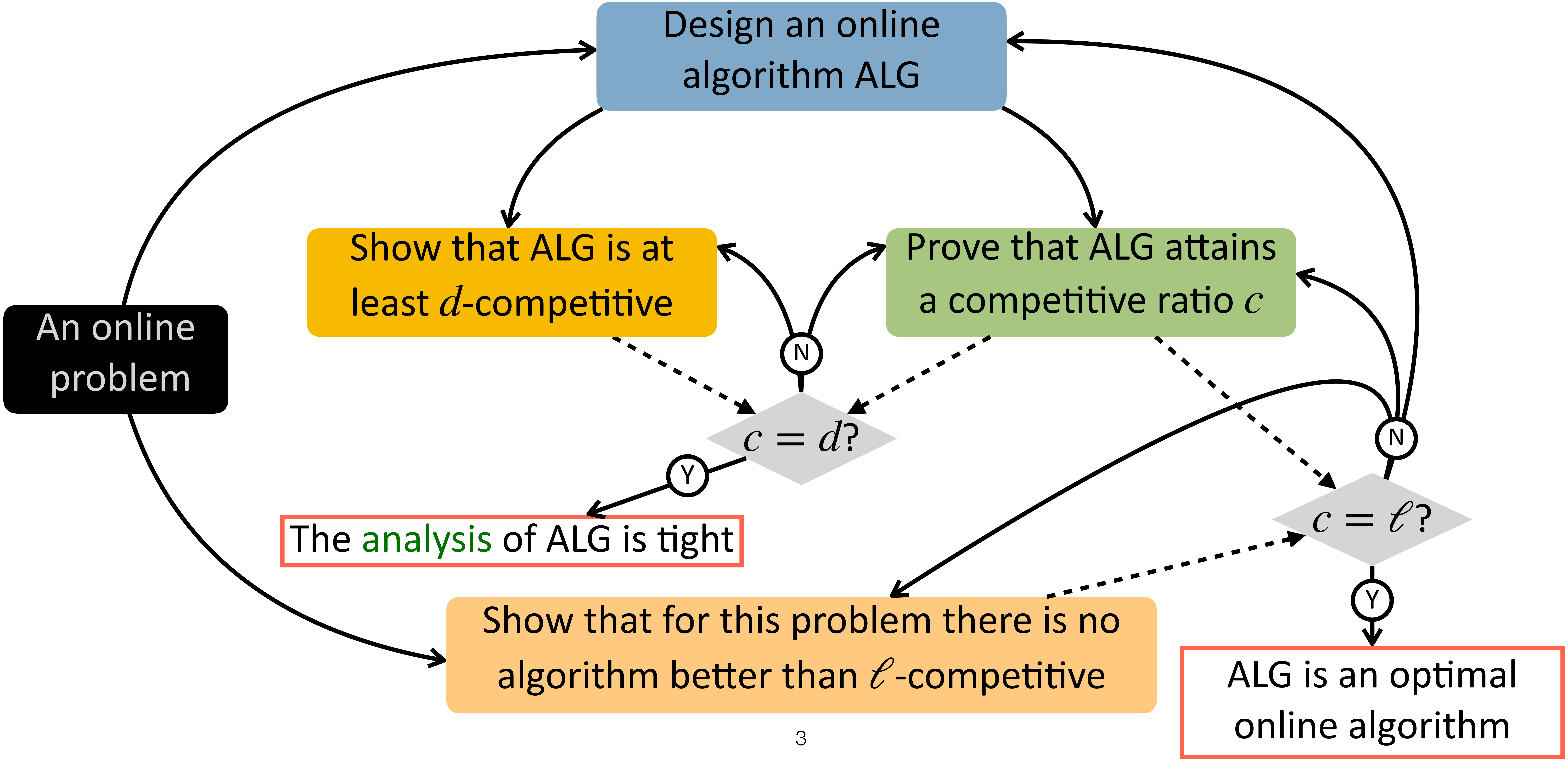
Bin Packing and Paging

Outline

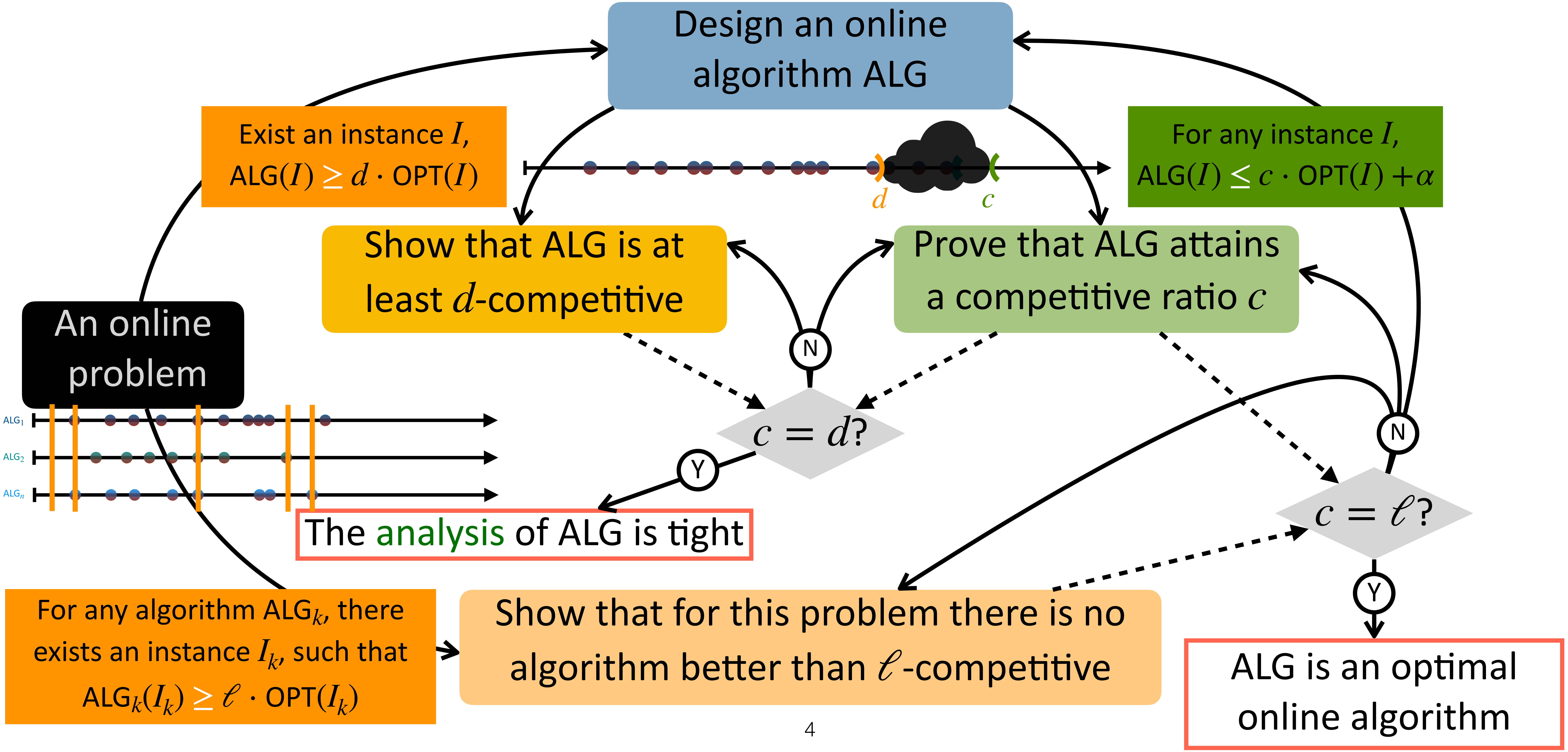
- **Bin Packing** problem
    - Assume that we know the **ALG** cost

- **Paging** problem
    - We know very little about the **ALG** or the **OPT**

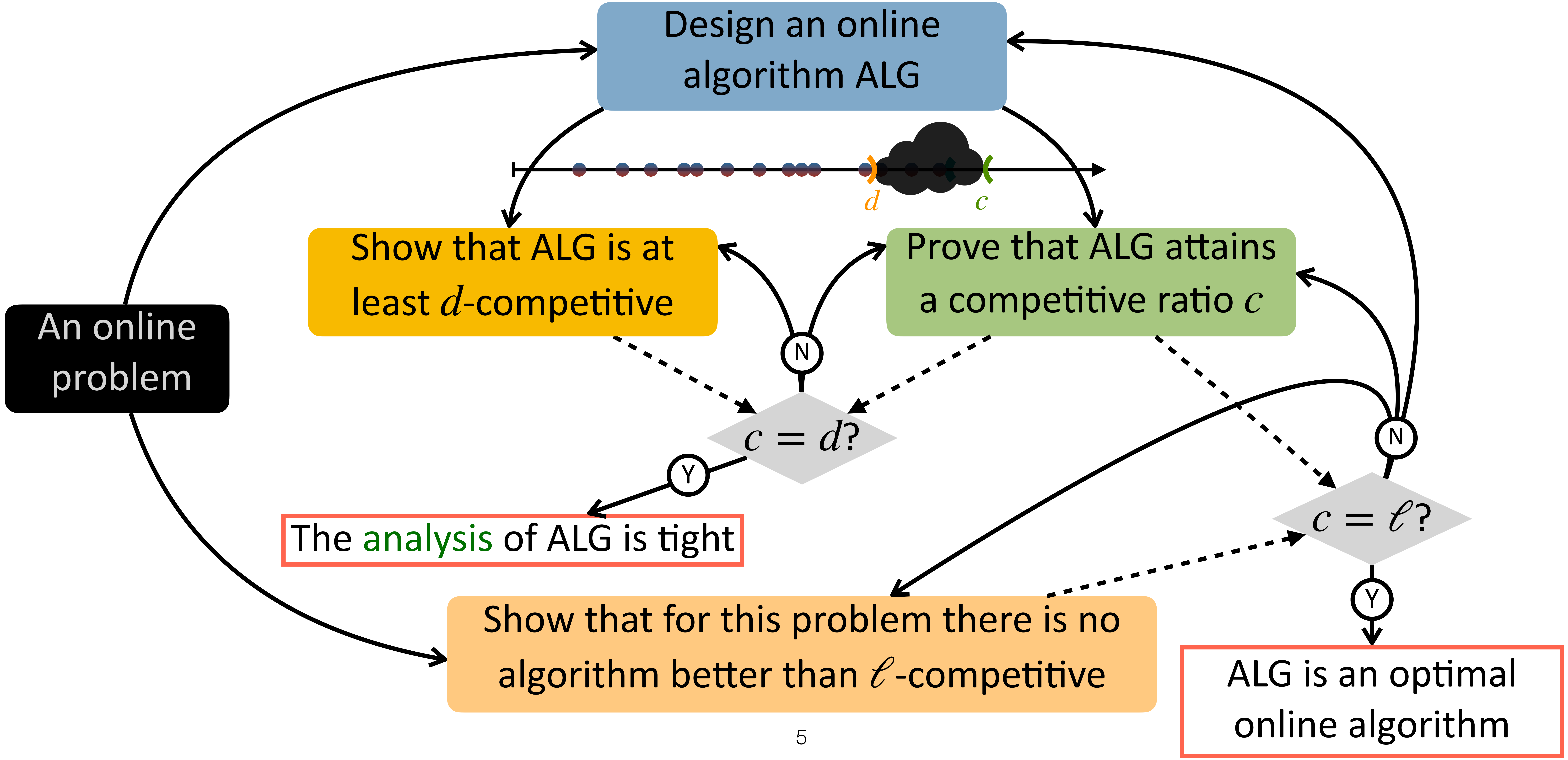
Research cycle of online algorithms



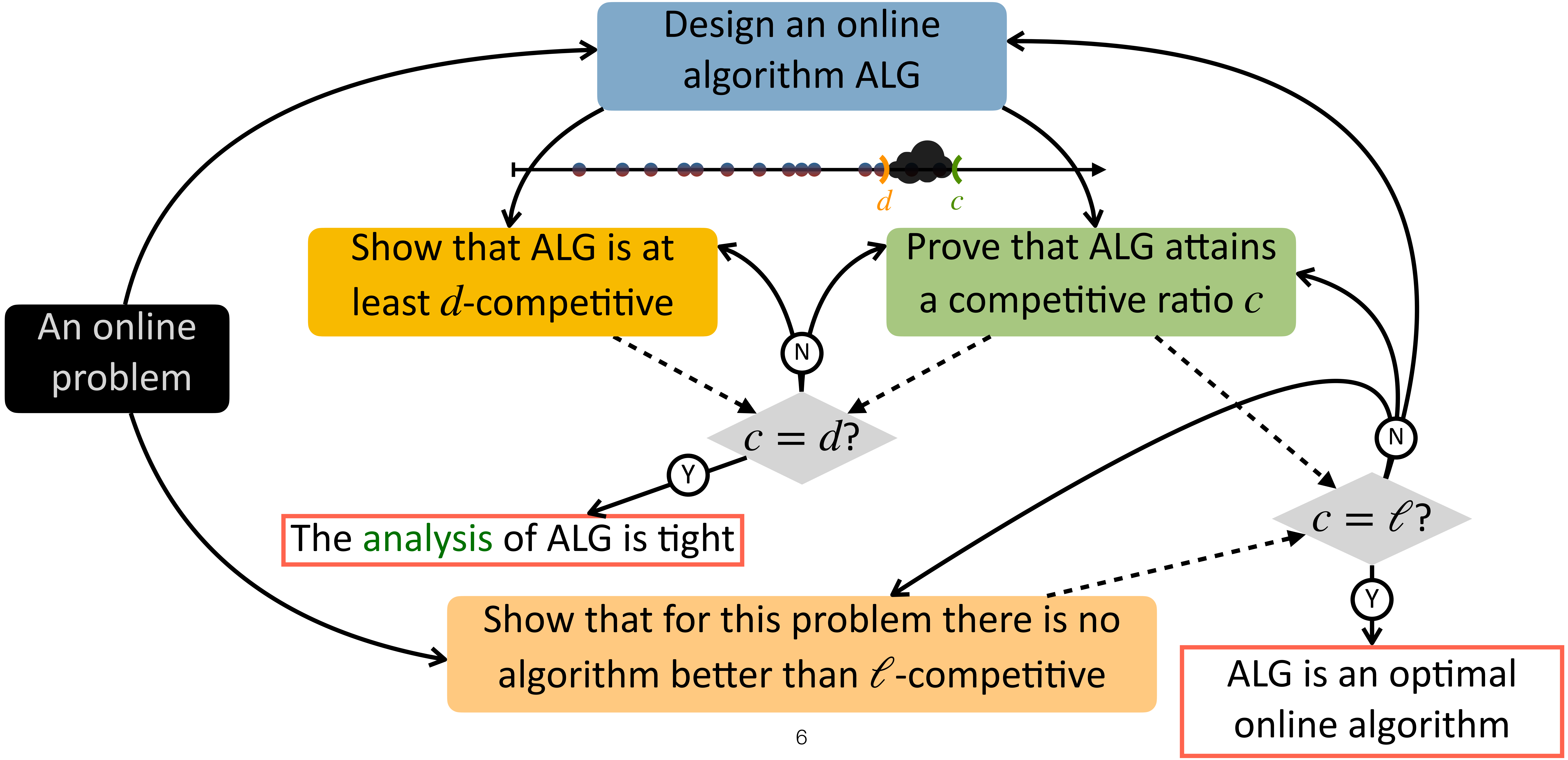
Research cycle of online algorithms



Research cycle of online algorithms

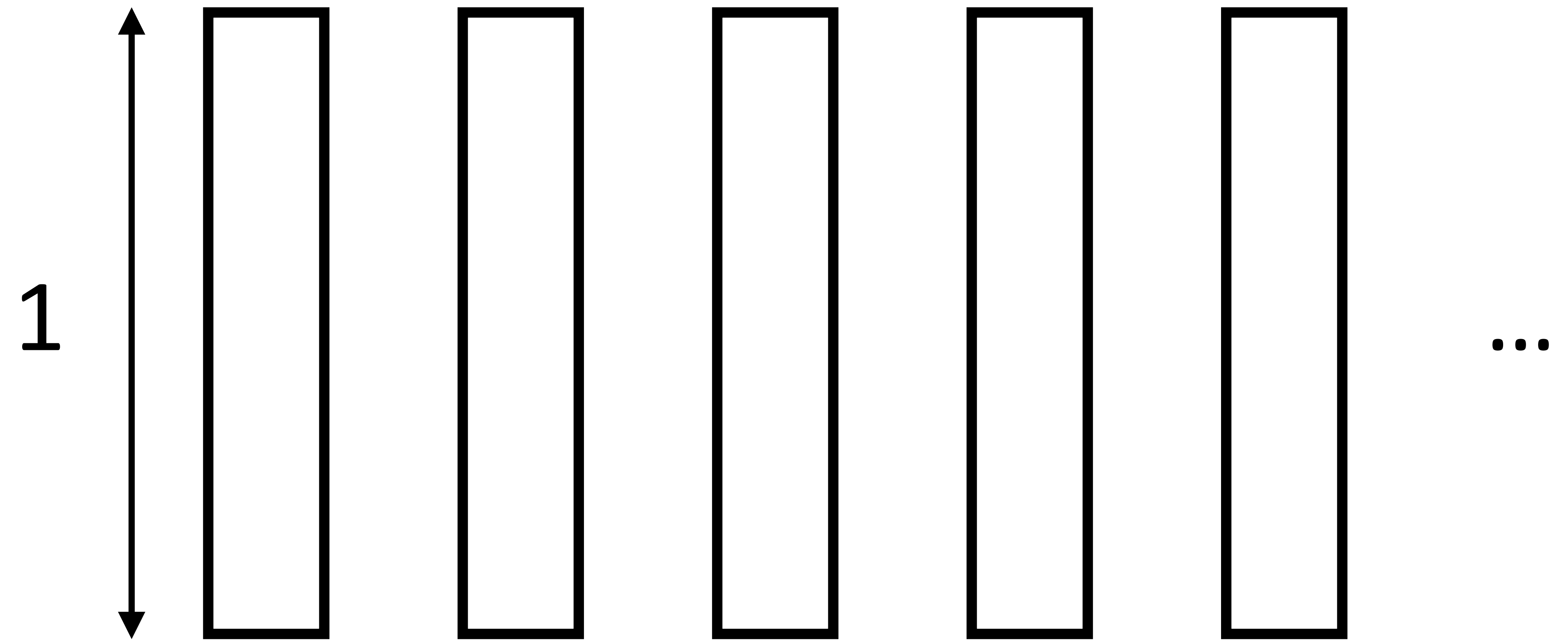


Research cycle of online algorithms



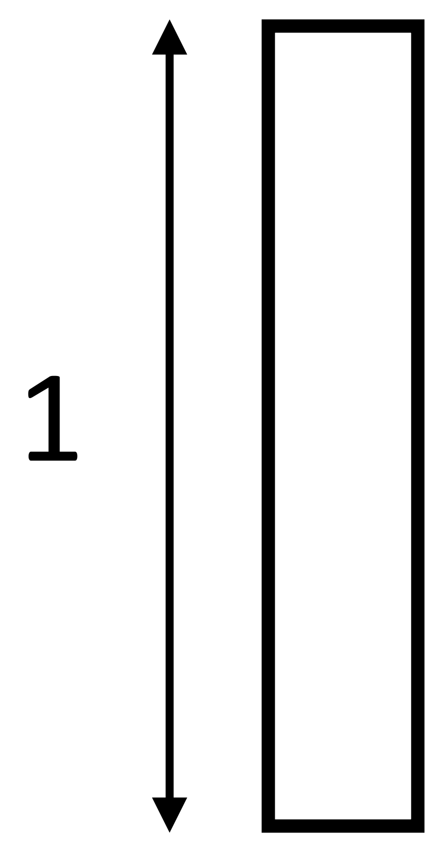
Online Bin Packing Problem

- There are infinite number of **capacity-1** ***bins***



Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The ***items*** i arrive online, each with ***size*** r_i between $(0,1]$



Online Bin Packing Problem

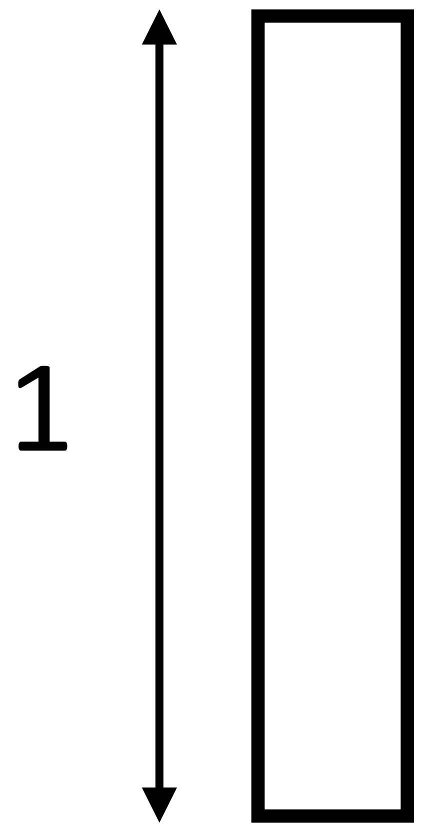
- There are infinite number of capacity-1 *bins*
- The ***items*** i arrive online, each with ***size*** r_i between $(0,1]$

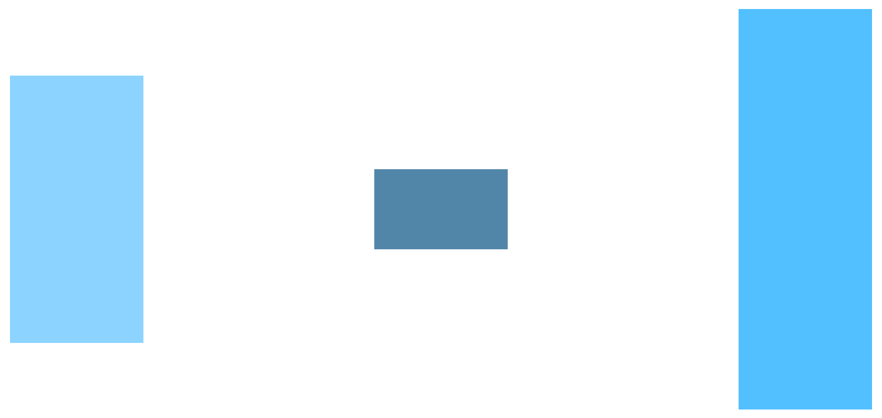
Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The ***items*** i arrive online, each with ***size*** r_i between $(0,1]$

Online Bin Packing Problem

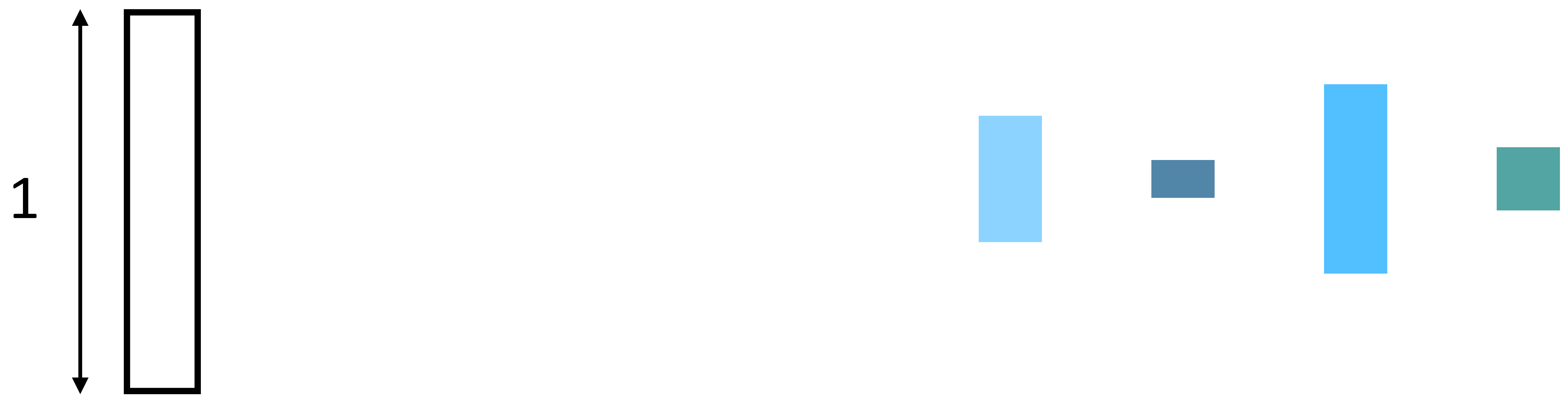
- There are infinite number of capacity-1 *bins*
- The ***items*** i arrive online, each with ***size*** r_i between $(0,1]$





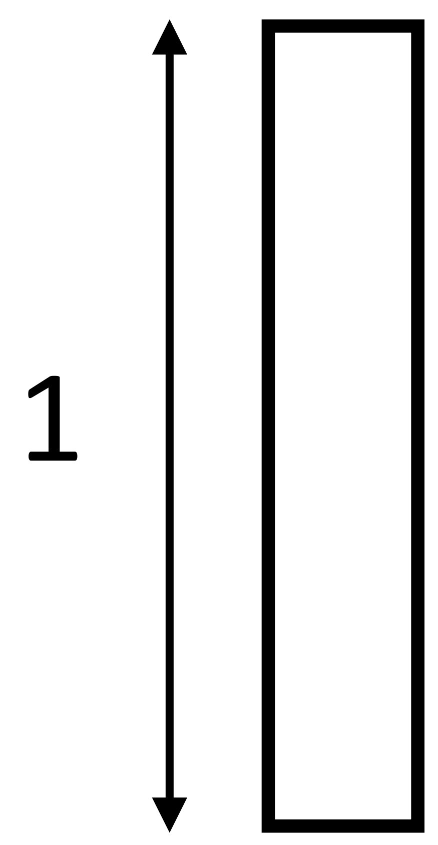
Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The ***items*** i arrive online, each with ***size*** r_i between $(0,1]$



Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The *items* i arrive online, each with *size* r_i between $(0,1]$
- Once an item arrives, we have to put it into a bin without exceeding the bin capacity

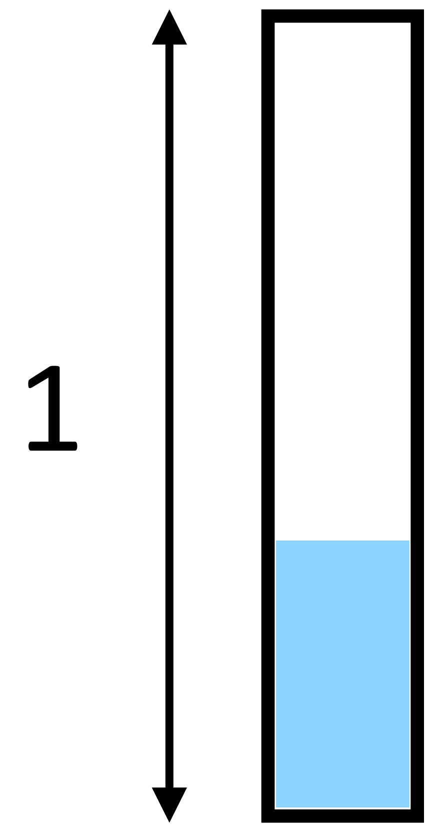


Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The *items* i arrive online, each with *size* r_i between $(0,1]$
- Once an item arrives, we have to put it into a bin without exceeding the bin capacity

Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The *items* i arrive online, each with *size* r_i between $(0,1]$
- Once an item arrives, we have to put it into a bin without exceeding the bin capacity

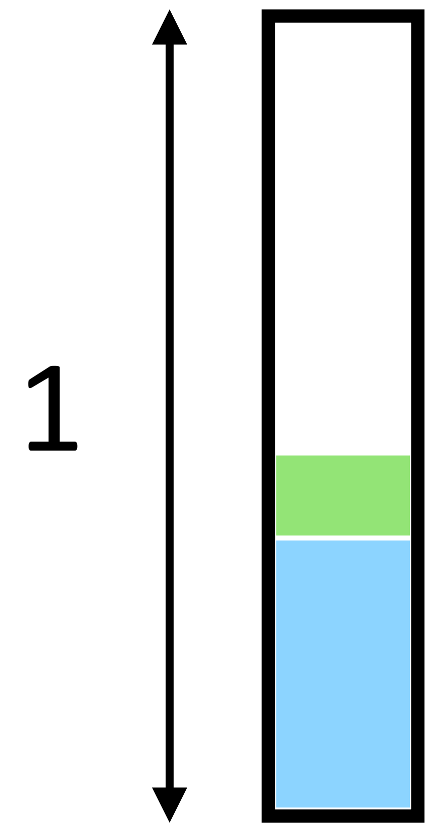


Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The *items* i arrive online, each with *size* r_i between $(0,1]$
- Once an item arrives, we have to put it into a bin without exceeding the bin capacity

Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The *items* i arrive online, each with *size* r_i between $(0,1]$
- Once an item arrives, we have to put it into a bin without exceeding the bin capacity

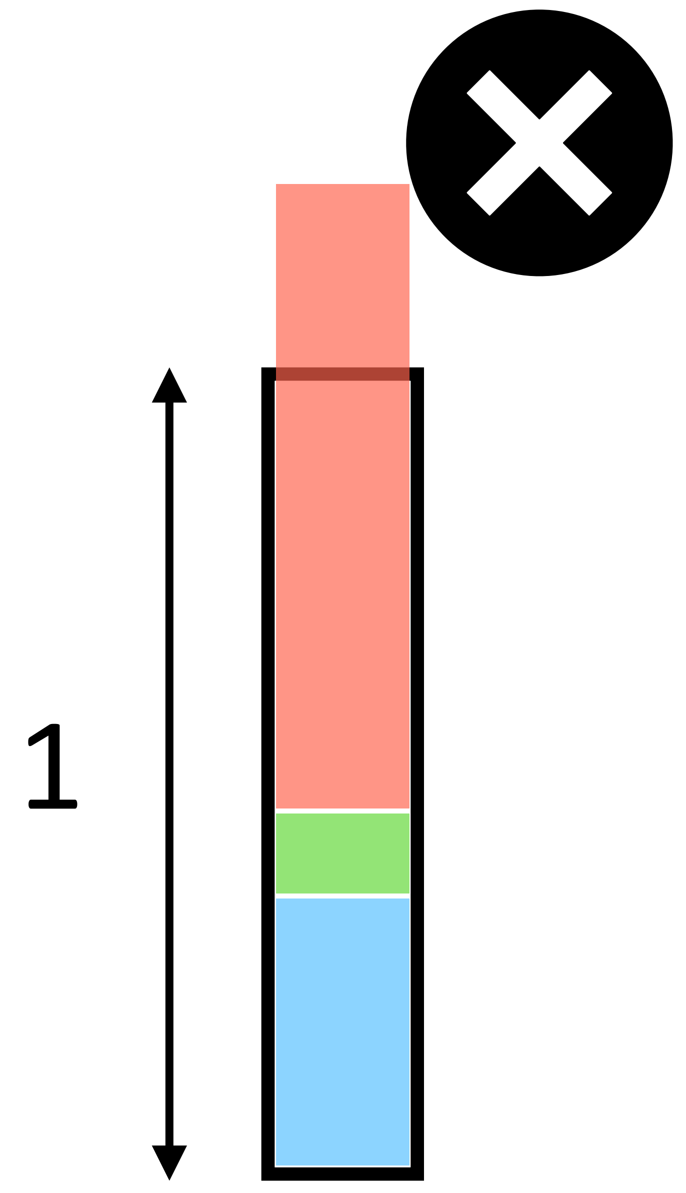


Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The *items* i arrive online, each with *size* r_i between $(0,1]$
- Once an item arrives, we have to put it into a bin without exceeding the bin capacity

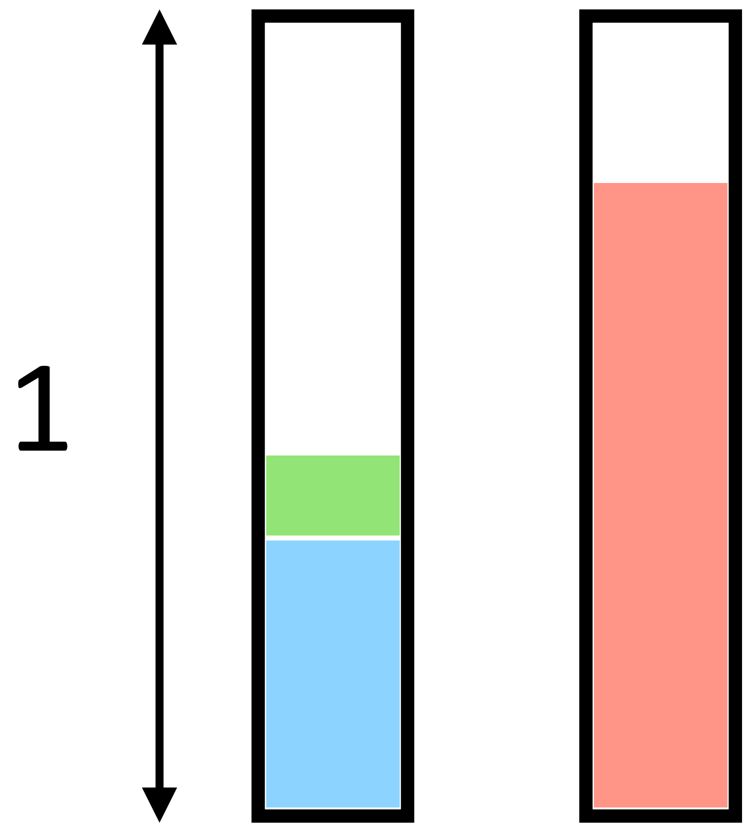
Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The *items* i arrive online, each with *size* r_i between $(0,1]$
- Once an item arrives, we have to put it into a bin without exceeding the bin capacity



Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The *items* i arrive online, each with *size* r_i between $(0,1]$
- Once an item arrives, we have to put it into a bin without exceeding the bin capacity (we may need to *open* a new bin)

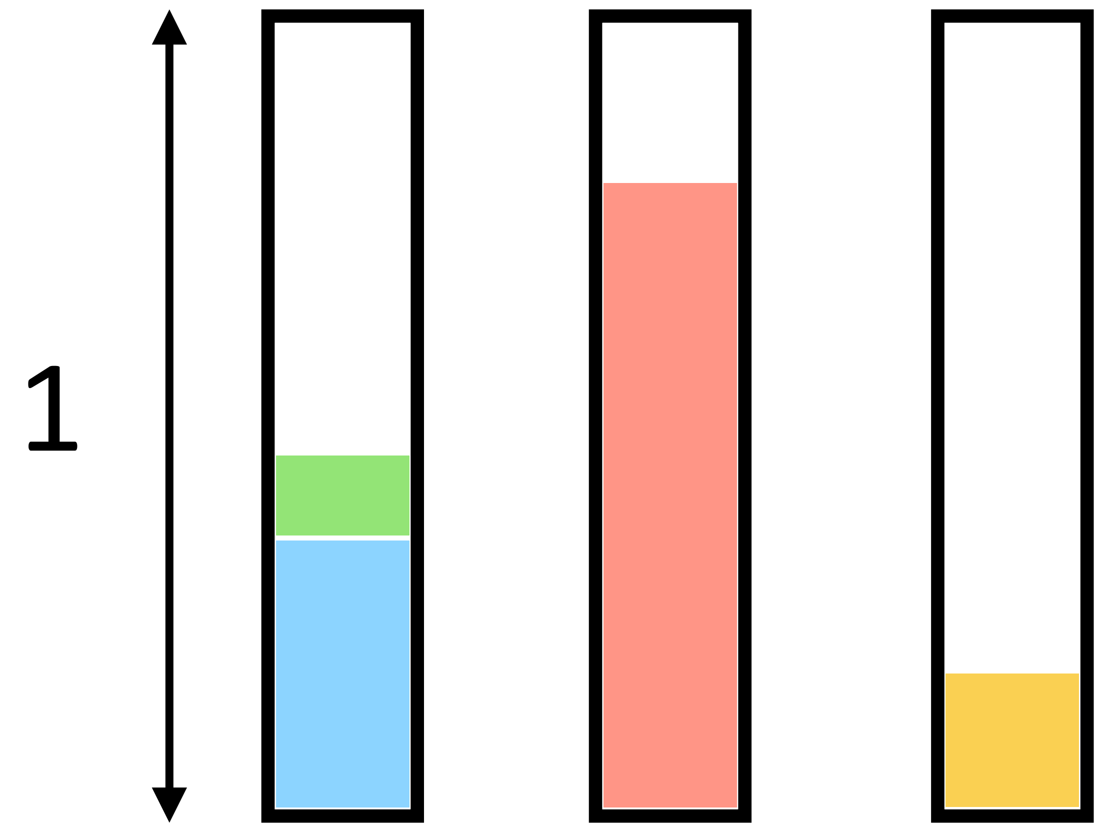


Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The *items* i arrive online, each with *size* r_i between $(0,1]$
- Once an item arrives, we have to put it into a bin without exceeding the bin capacity (we may need to *open* a new bin)

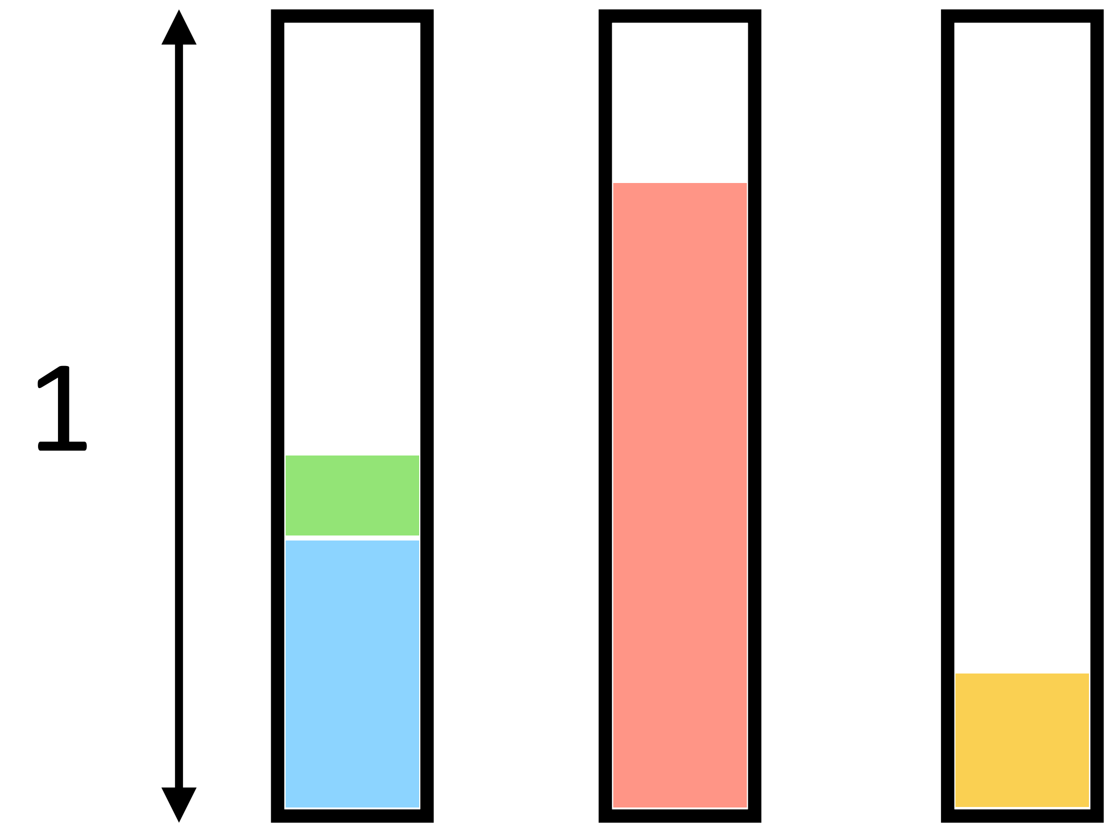
Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The *items* i arrive online, each with *size* r_i between $(0,1]$
- Once an item arrives, we have to put it into a bin without exceeding the bin capacity (we may need to *open* a new bin)



Online Bin Packing Problem

- There are infinite number of capacity-1 *bins*
- The *items* i arrive online, each with *size* r_i between $(0,1]$
- Once an item arrives, we have to put it into a bin without exceeding the bin capacity (we may need to *open* a new bin)
- The objective is to **put all items in a minimum number of bins**



Outline

- **Bin Packing** problem
  - Assume that we know the **ALG** cost

- **Paging** problem
  - We know very little about the **ALG** or the **OPT**

FirstFit Algorithm

FirstFit:

Once an item arrives:

- Scan through all (opened) bins and put it into the first bin that can accommodate it
- If there is no such a bin, open a new bin and put it in

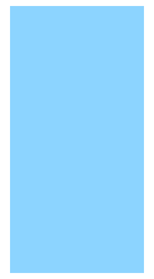
FirstFit Algorithm

FirstFit:

Once an item arrives:

Scan through all (opened) bins and put it into the first bin that can accommodate it

If there is no such a bin, open a new bin and put it in



FirstFit Algorithm

FirstFit:

Once an item arrives:

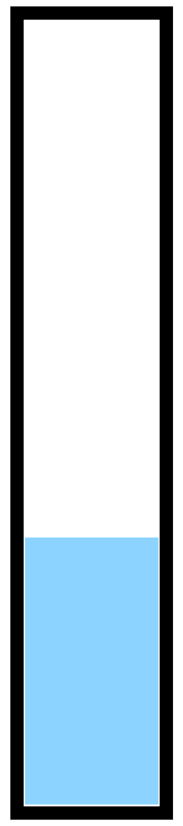
- Scan through all (opened) bins and put it into the first bin that can accommodate it
- If there is no such a bin, open a new bin and put it in

FirstFit Algorithm

FirstFit:

Once an item arrives:

- Scan through all (opened) bins and put it into the first bin that can accommodate it
- If there is no such a bin, open a new bin and put it in



FirstFit Algorithm

FirstFit:

Once an item arrives:

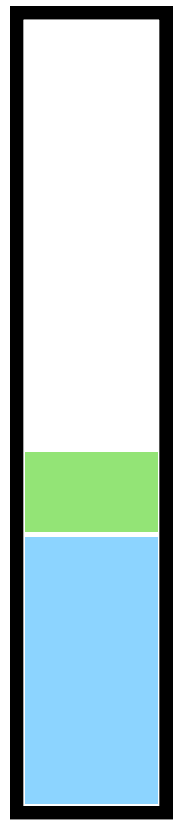
- Scan through all (opened) bins and put it into the first bin that can accommodate it
- If there is no such a bin, open a new bin and put it in

FirstFit Algorithm

FirstFit:

Once an item arrives:

- Scan through all (opened) bins and put it into the first bin that can accommodate it
- If there is no such a bin, open a new bin and put it in



FirstFit Algorithm

FirstFit:

Once an item arrives:

Scan through all (opened) bins and put it into the first bin that can accommodate it

If there is no such a bin, open a new bin and put it in

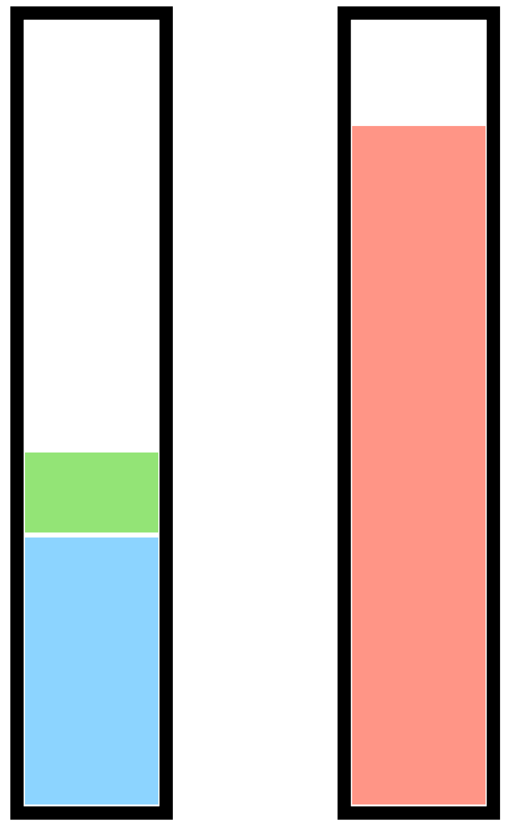
FirstFit Algorithm

FirstFit:

Once an item arrives:

Scan through all (opened) bins and put it into the first bin that can accommodate it

If there is no such a bin, open a new bin and put it in



FirstFit Algorithm

FirstFit:

Once an item arrives:

Scan through all (opened) bins and put it into the first bin that can accommodate it

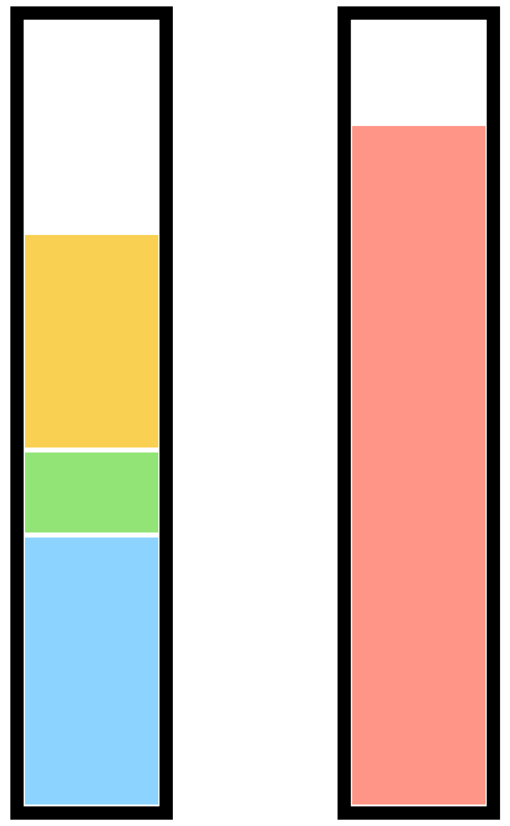
If there is no such a bin, open a new bin and put it in

FirstFit Algorithm

FirstFit:

Once an item arrives:

- Scan through all (opened) bins and put it into the first bin that can accommodate it
- If there is no such a bin, open a new bin and put it in



FirstFit Algorithm

FirstFit:

Once an item arrives:

Scan through all (opened) bins and put it into the first bin that can accommodate it

If there is no such a bin, open a new bin and put it in

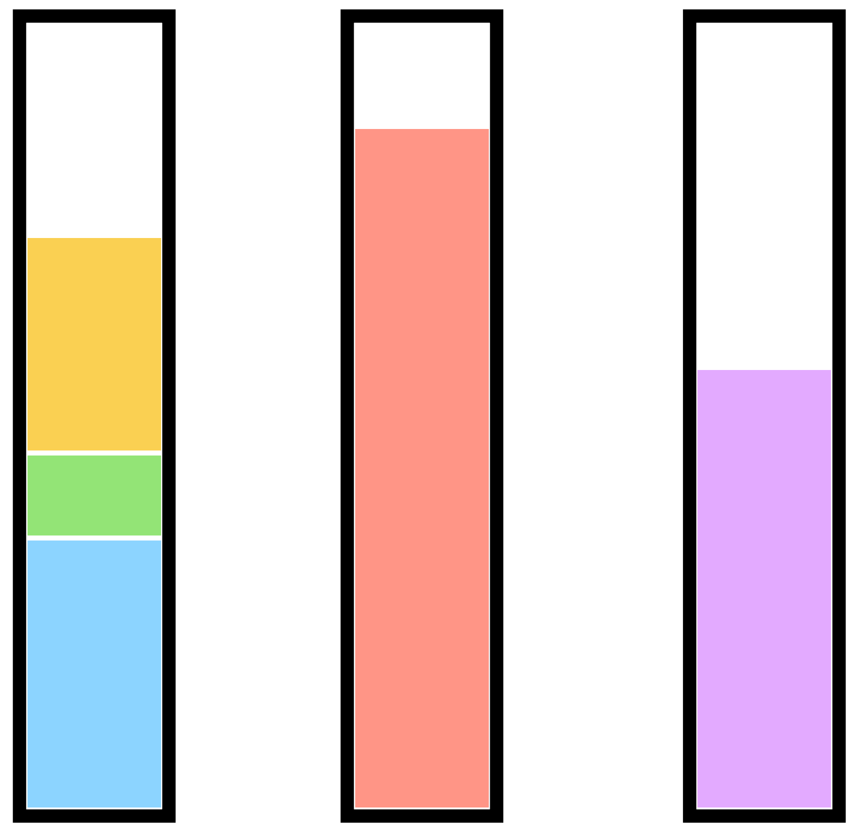
FirstFit Algorithm

FirstFit:

Once an item arrives:

Scan through all (opened) bins and put it into the first bin that can accommodate it

If there is no such a bin, open a new bin and put it in

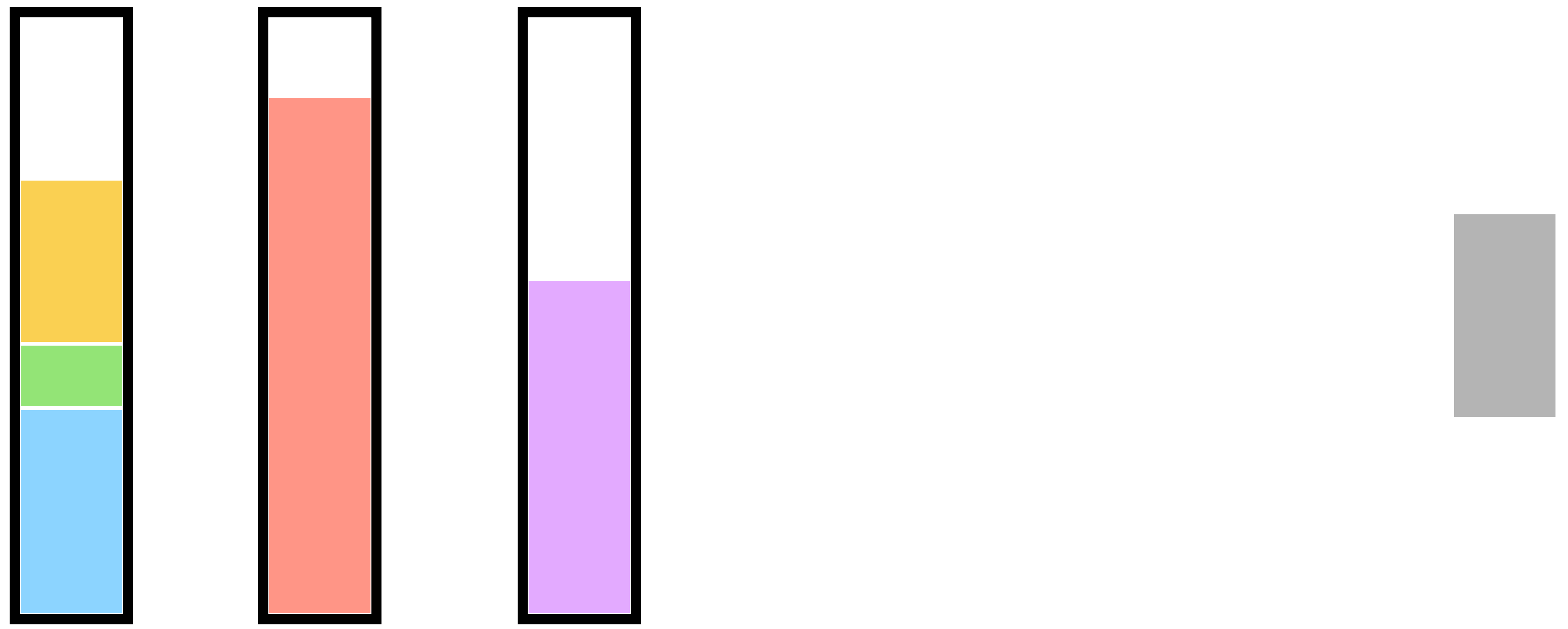


FirstFit Algorithm

FirstFit:

Once an item arrives:

- Scan through all (opened) bins and put it into the first bin that can accommodate it
- If there is no such a bin, open a new bin and put it in

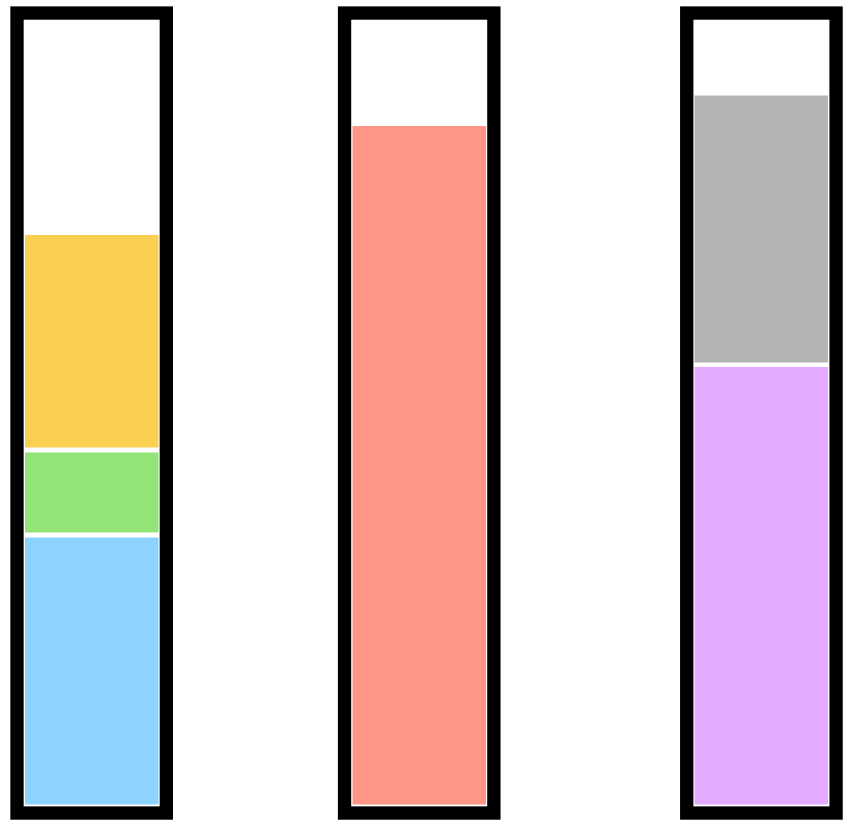


FirstFit Algorithm

FirstFit:

Once an item arrives:

- Scan through all (opened) bins and put it into the first bin that can accommodate it
- If there is no such a bin, open a new bin and put it in



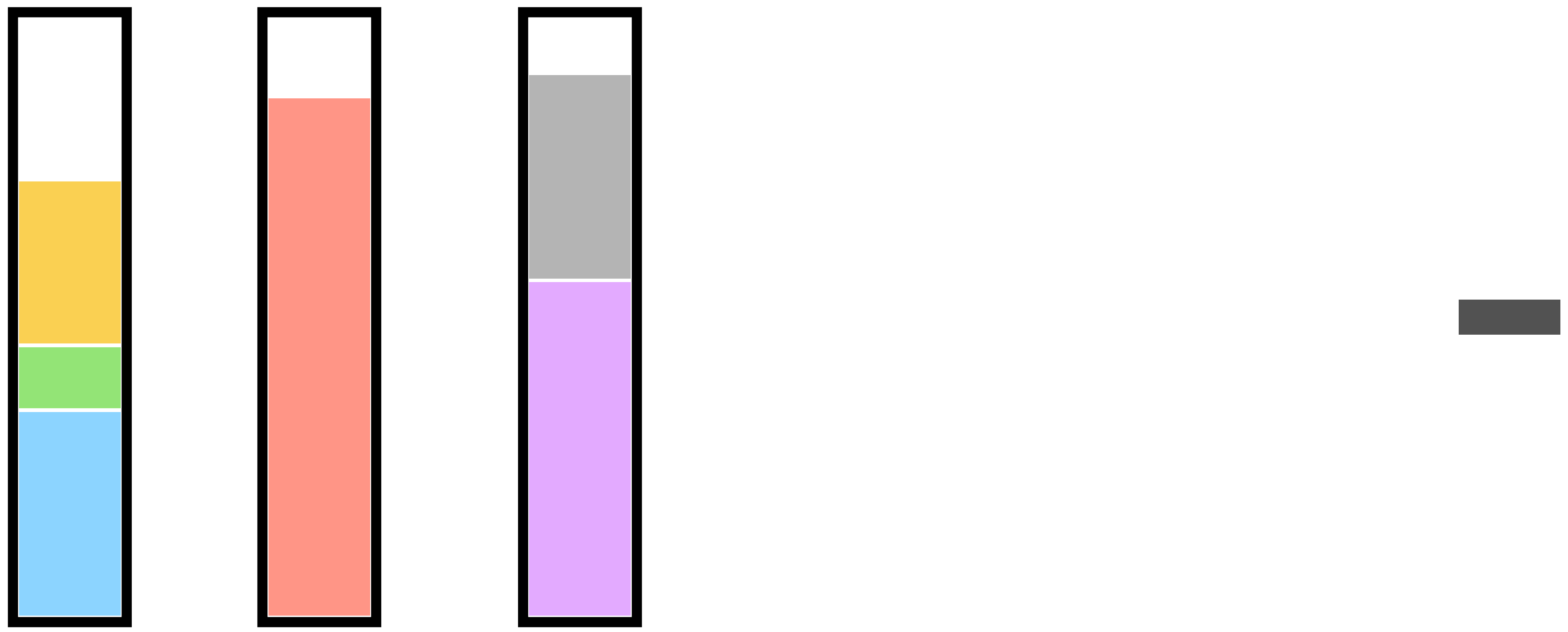
FirstFit Algorithm

FirstFit:

Once an item arrives:

  Scan through all (opened) bins and put it into the first bin that can accommodate it

  If there is no such a bin, open a new bin and put it in



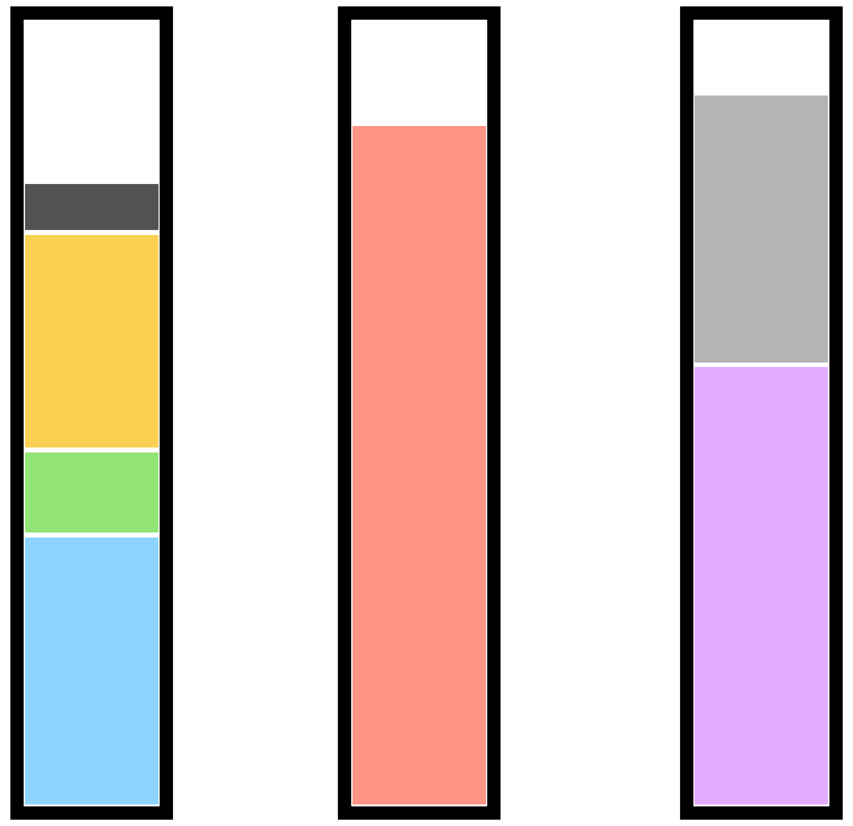
FirstFit Algorithm

FirstFit:

Once an item arrives:

Scan through all (opened) bins and put it into the first bin that can accommodate it

If there is no such a bin, open a new bin and put it in



Outline

- **Bin Packing** problem
  - Assume that we know the **ALG** cost
- **Paging** problem
  - We know very little about the **ALG** or the **OPT**

$\text{FirstFit} \leq 2 \cdot \text{OPT} + 1$

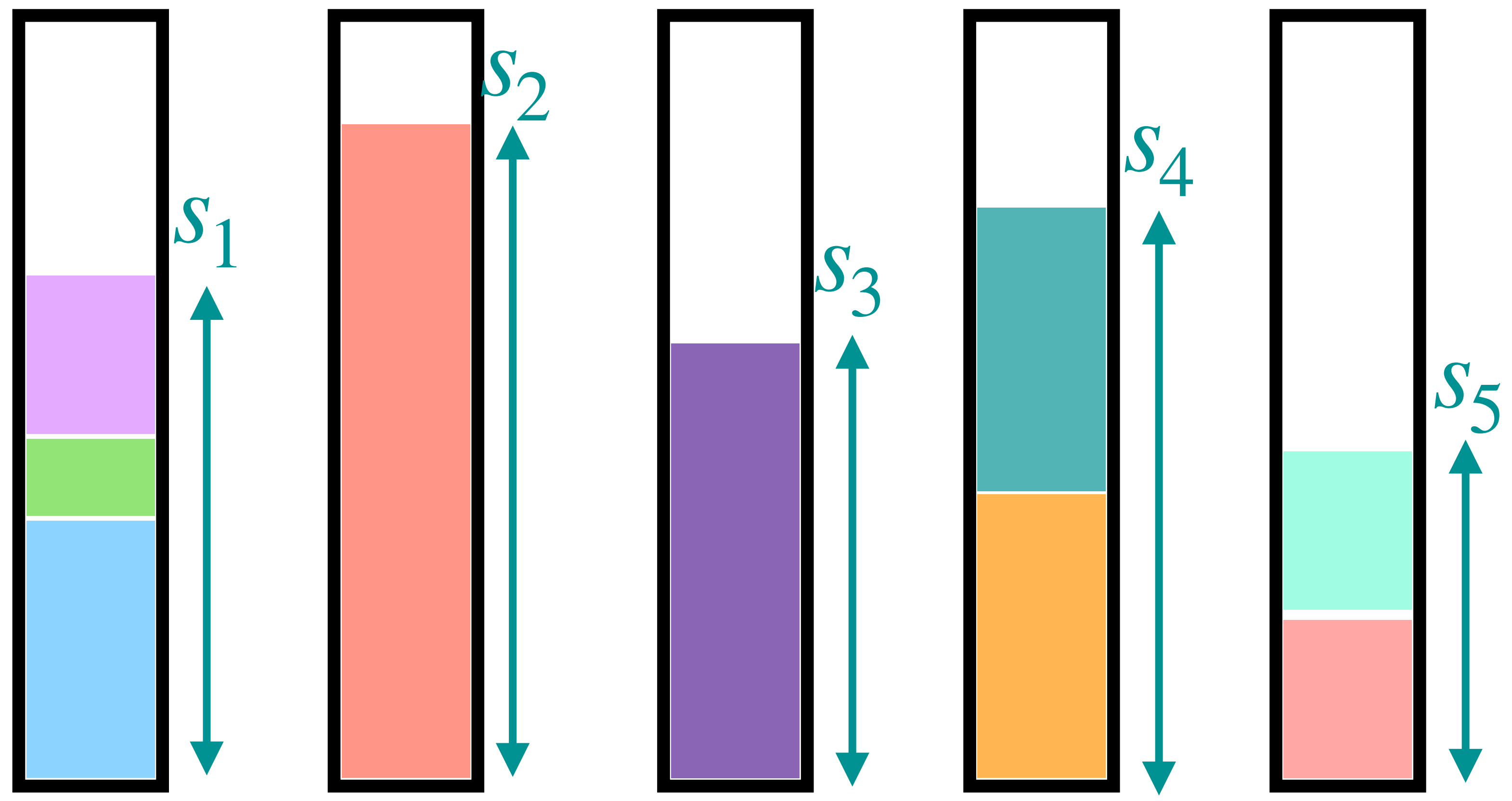
<Proof idea>

- Observation: There is at most one bin at least *half empty*

FirstFit $\leq 2 \cdot \mathrm{OPT} + 1$

<Proof idea>

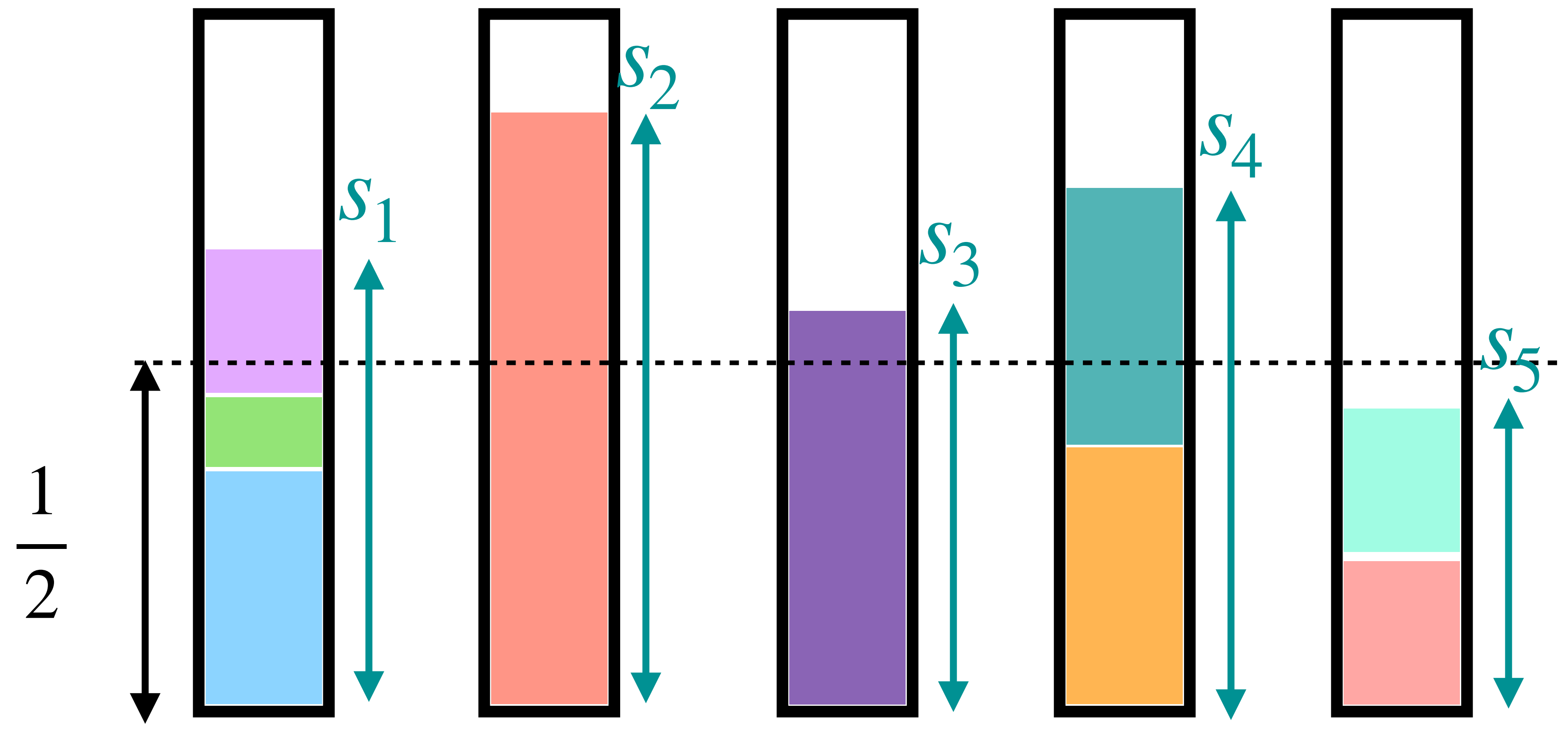
- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)



FirstFit $\leq 2 \cdot \mathrm{OPT} + 1$

<Proof idea>

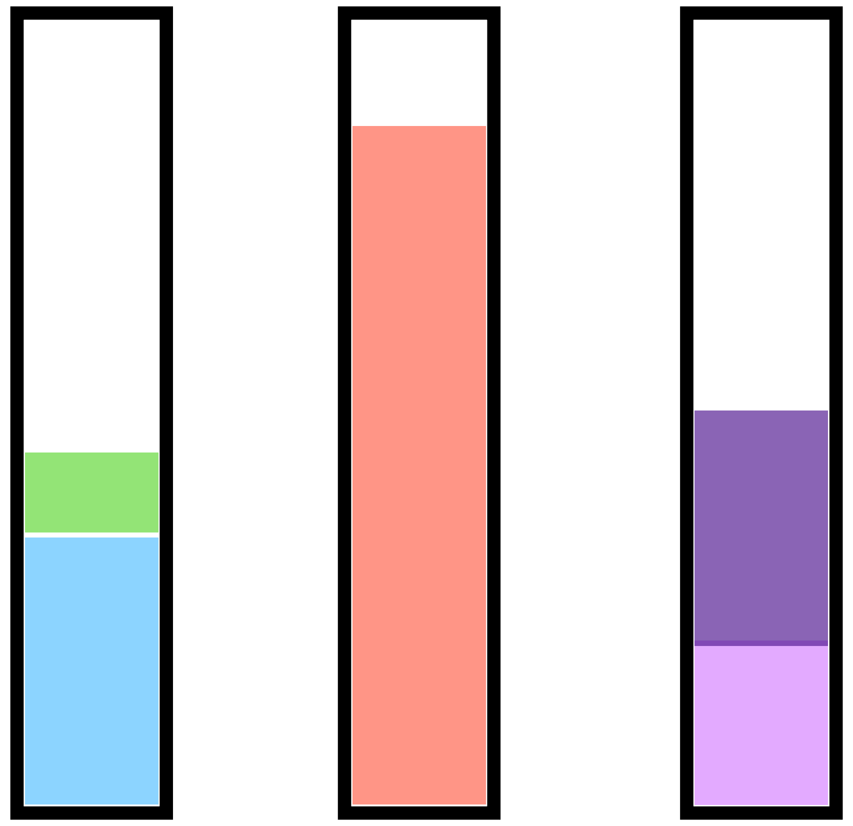
- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)



FirstFit $\leq 2 \cdot \text{OPT} + 1$

<Proof idea>

- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)

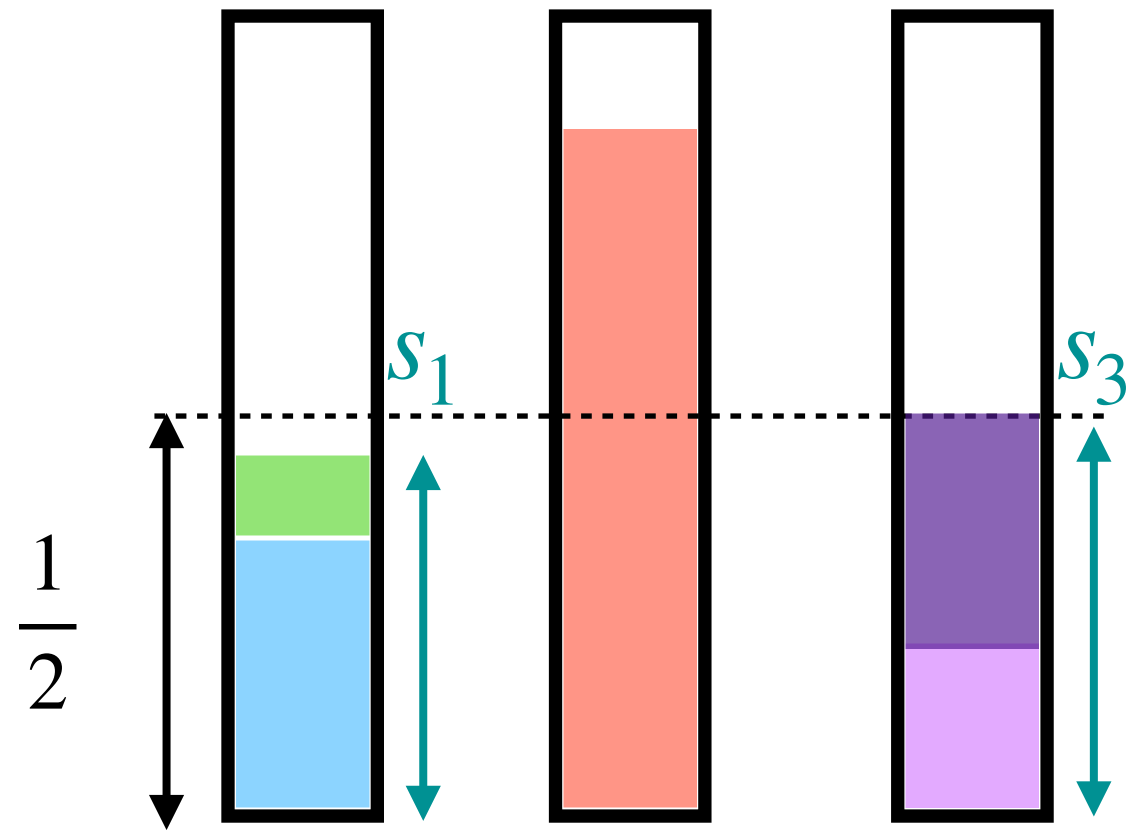


Prove by contradiction: Assume that there are two bins with size less than $1/2$

FirstFit $\leq 2 \cdot \text{OPT} + 1$

<Proof idea>

- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)

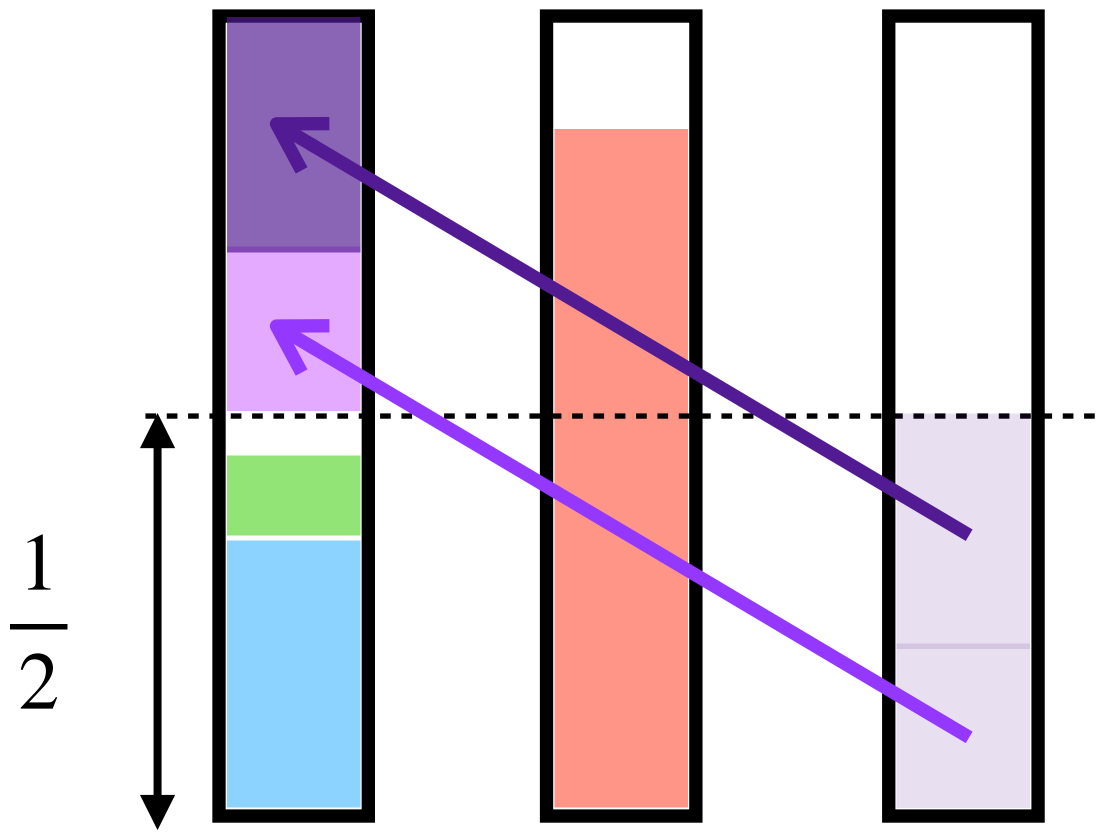


Prove by contradiction: Assume that there are two bins with size less than $1/2$

FirstFit $\leq 2 \cdot \text{OPT} + 1$

<Proof idea>

- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)



Prove by contradiction: Assume that there are two bins with size less than $1/2$

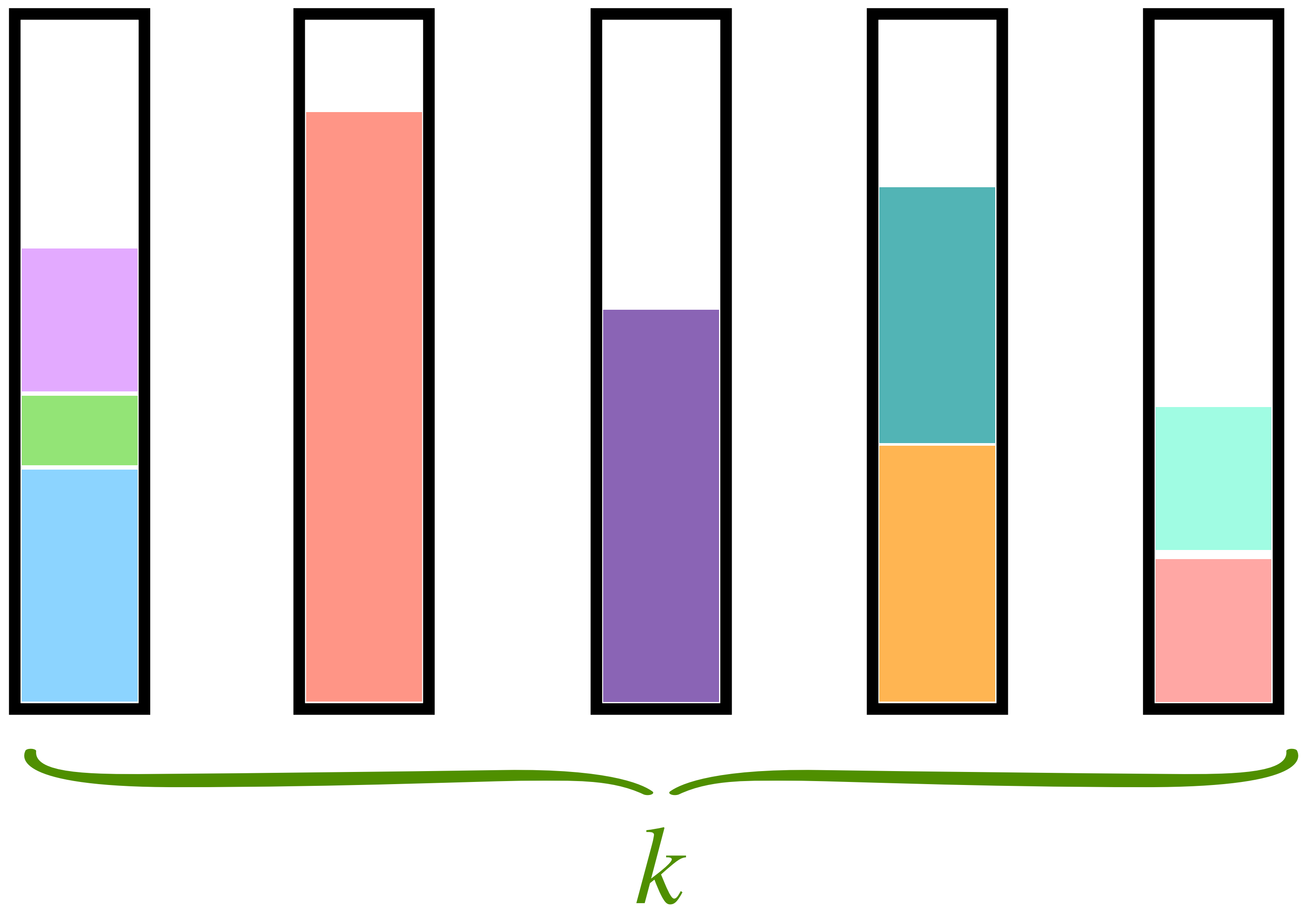
According to the FirstFit algorithm,
the items in the second bing can be put in the first (half-empty) bin

FirstFit $\leq 2 \cdot \text{OPT} + 1$

<Proof idea>

- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)

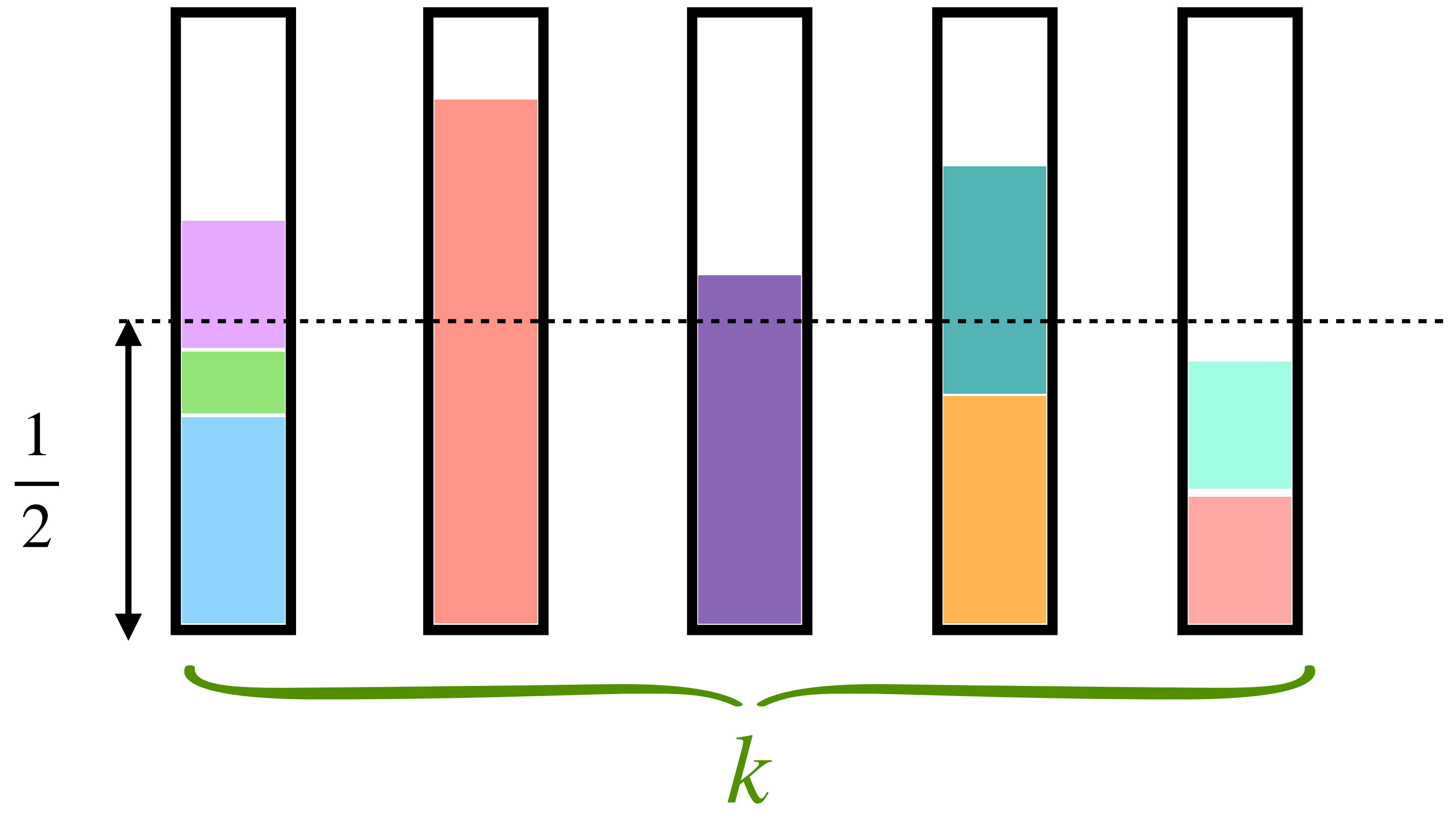
  - Let k be the number of bins opened by **FirstFit**.



FirstFit $\leq 2 \cdot \text{OPT} + 1$

<Proof idea>

- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)
  - Let k be the number of bins opened by **FirstFit**. Total size of all items $\geq (k-1) \cdot \frac{1}{2}$

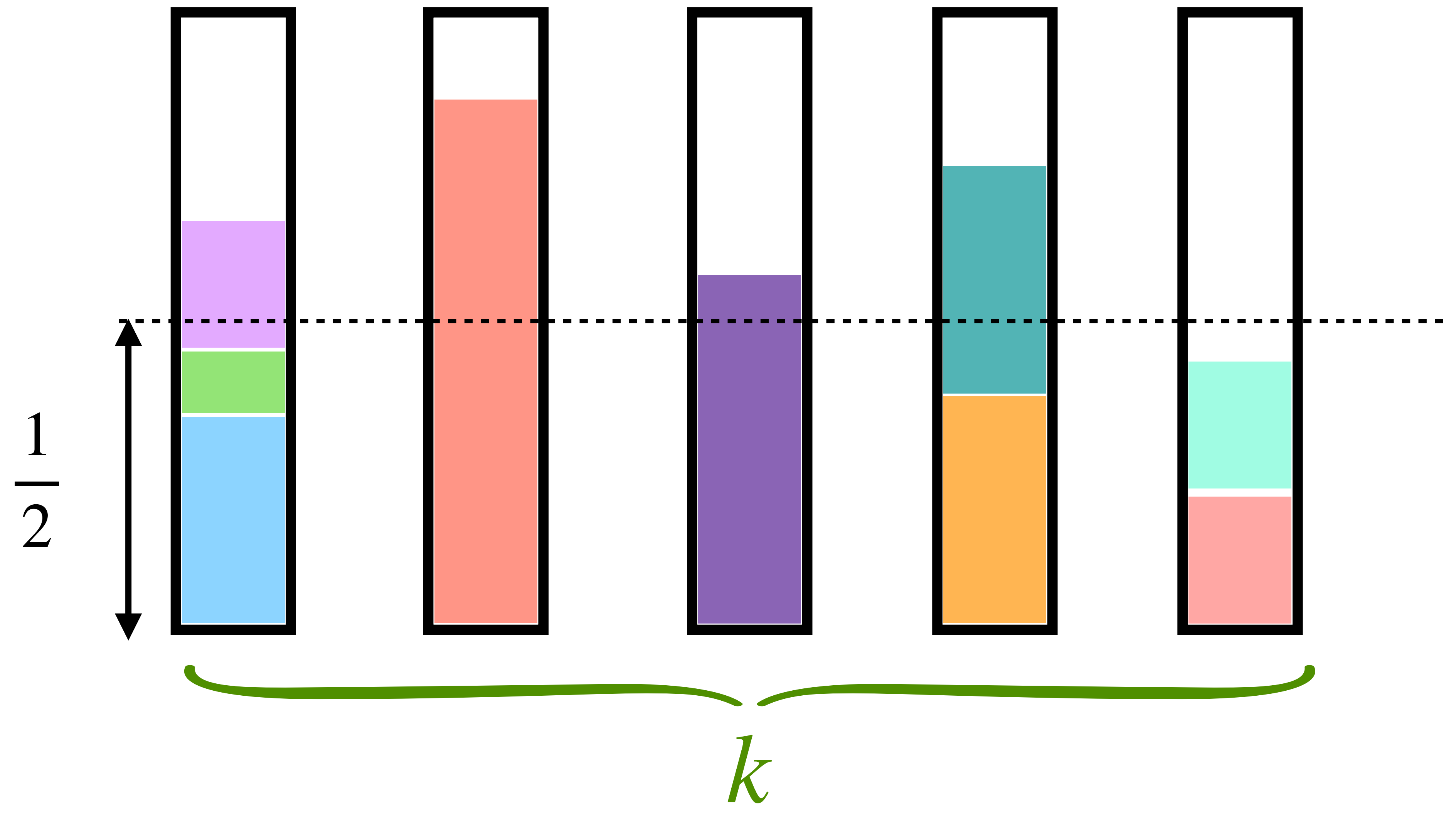


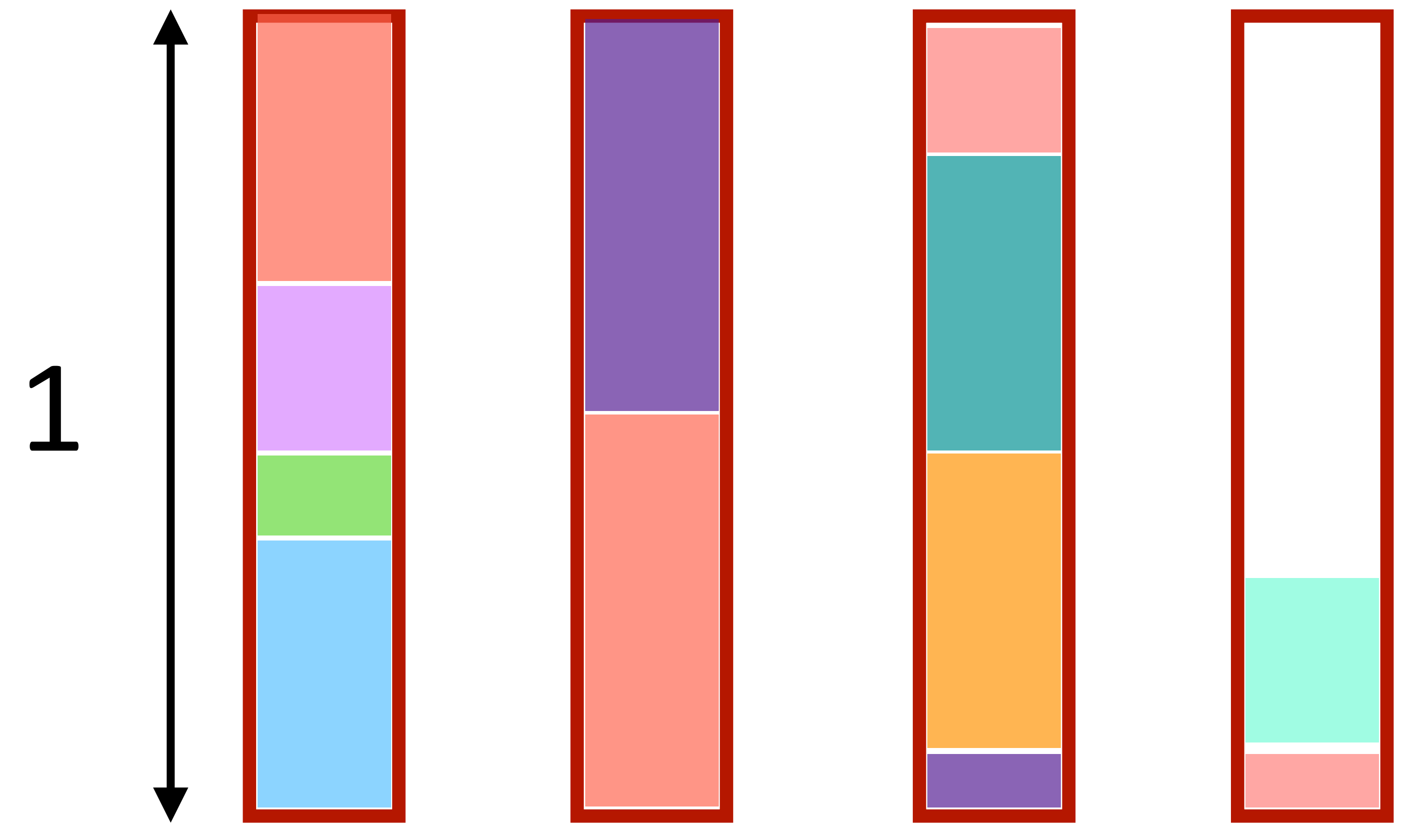
FirstFit $\leq 2 \cdot \text{OPT} + 1$

<Proof idea>

- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)
  - Let k be the number of bins opened by **FirstFit**. Total size of all items $\geq (k-1) \cdot \frac{1}{2}$

Even when the optimal algorithm has superpower to cut the items, it needs Total size of bins to accommodate all the items



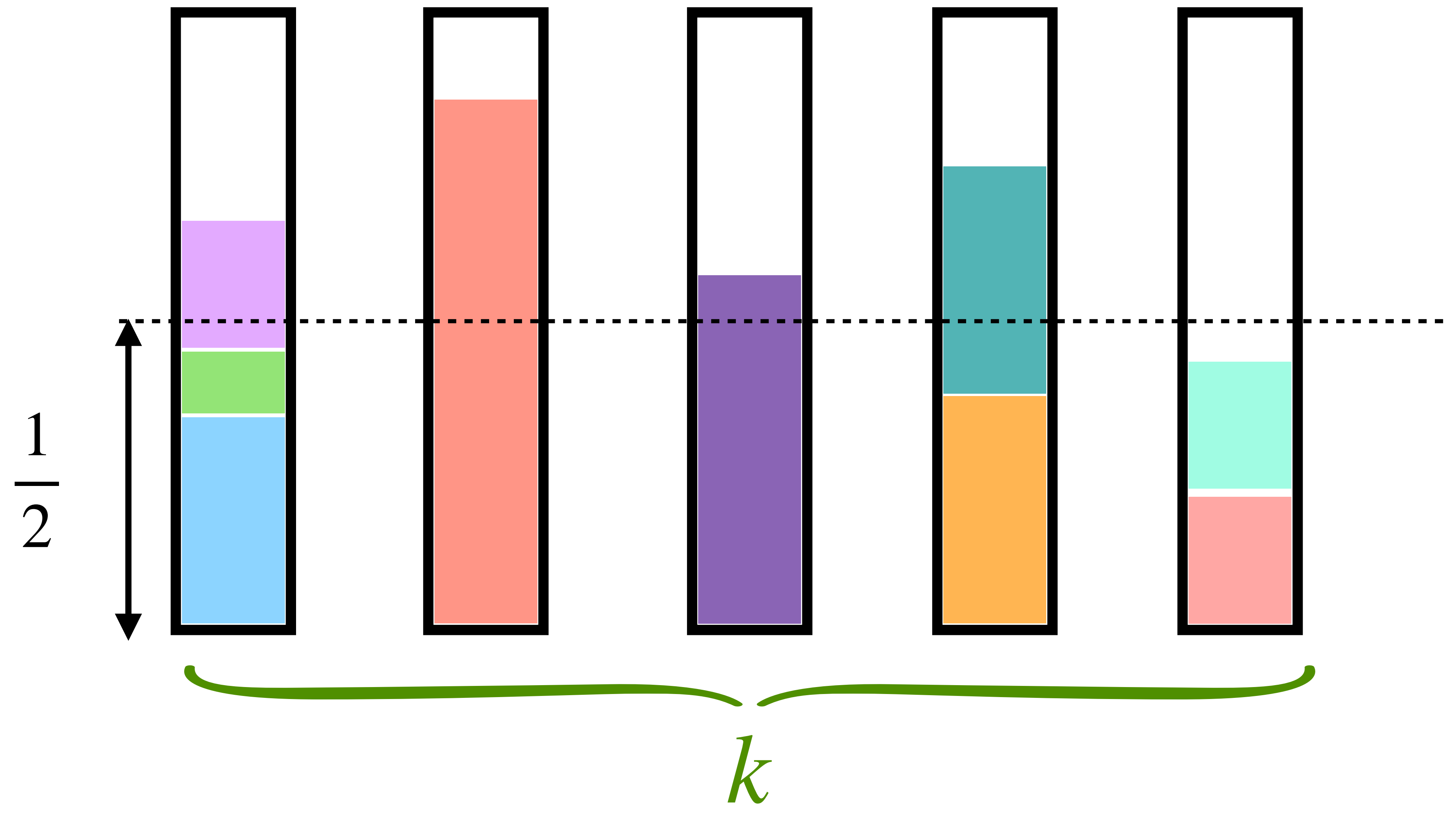


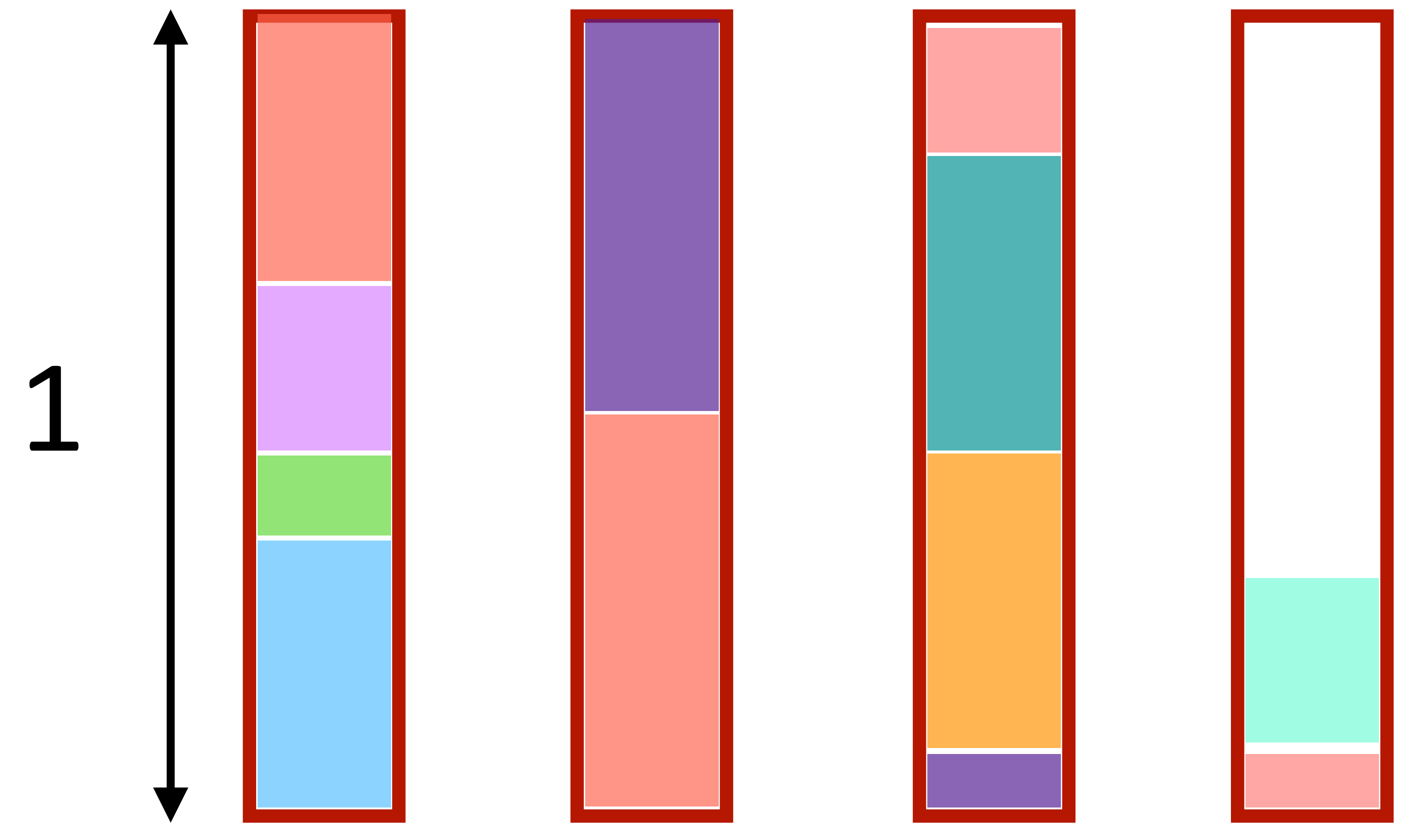
FirstFit $\leq 2 \cdot \text{OPT} + 1$

<Proof idea>

- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)
    - Let k be the number of bins opened by **FirstFit**. Total size of all items $\geq (k-1) \cdot \frac{1}{2}$
- OPT $\geq$ (total size of all items)/1

Even when the optimal algorithm has superpower to cut the items, it needs Total size of bins to accommodate all the items

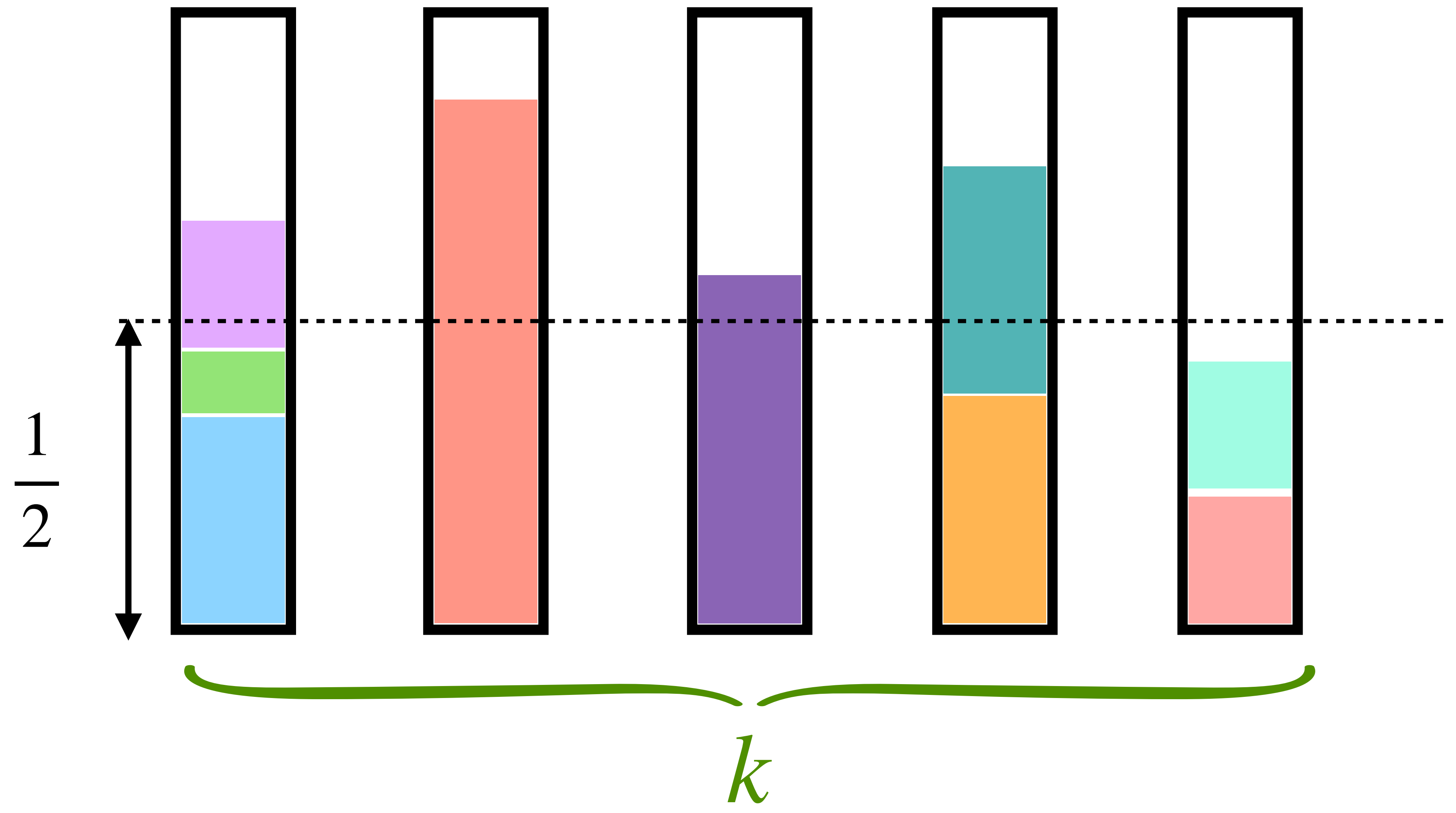


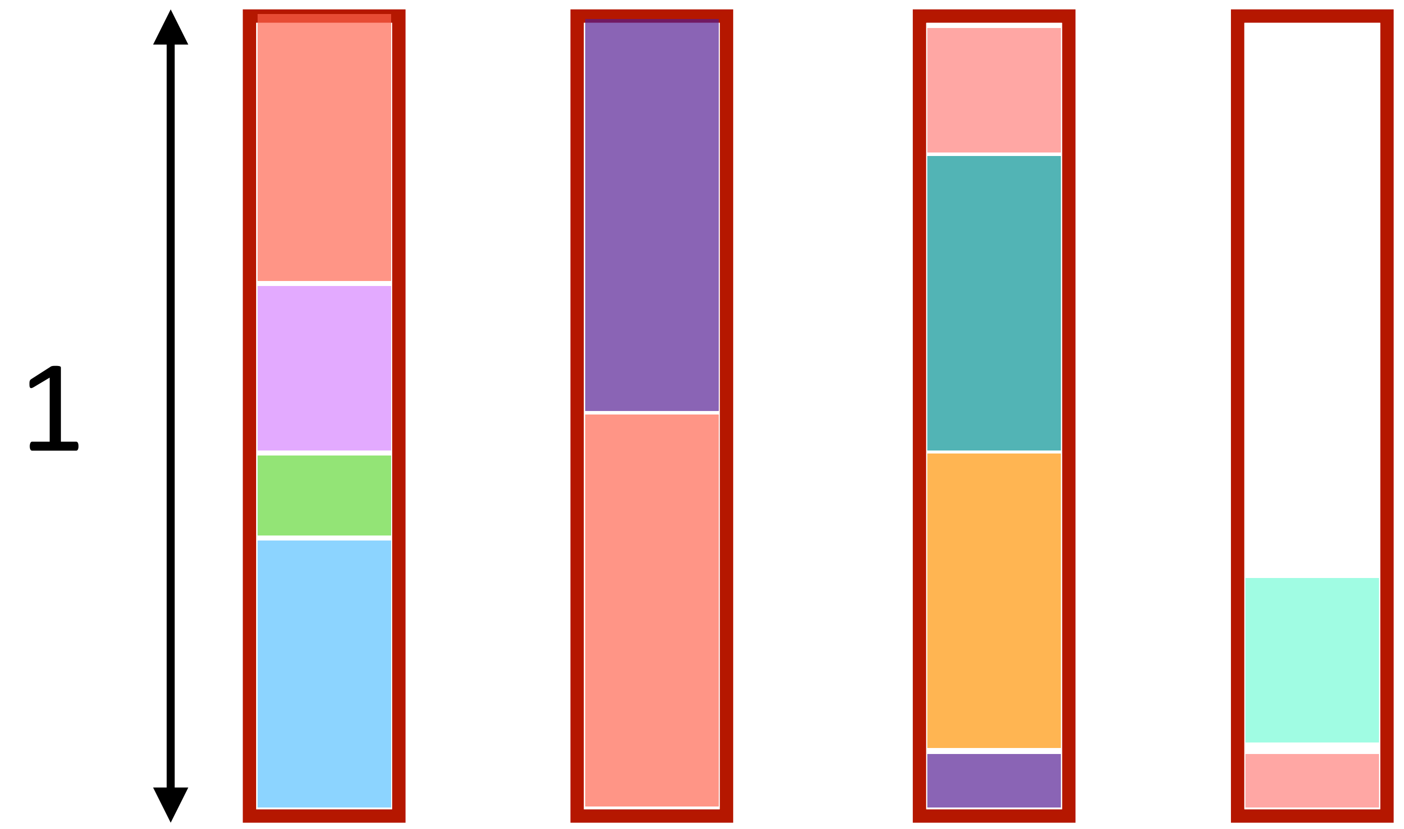


FirstFit $\leq 2 \cdot \text{OPT} + 1$

<Proof idea>

- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)
  - Let k be the number of bins opened by **FirstFit**. Total size of all items $\geq (k-1) \cdot \dfrac{1}{2}$
- OPT $\geq$ (total size of all items)/1 $\geq (k-1)/2$

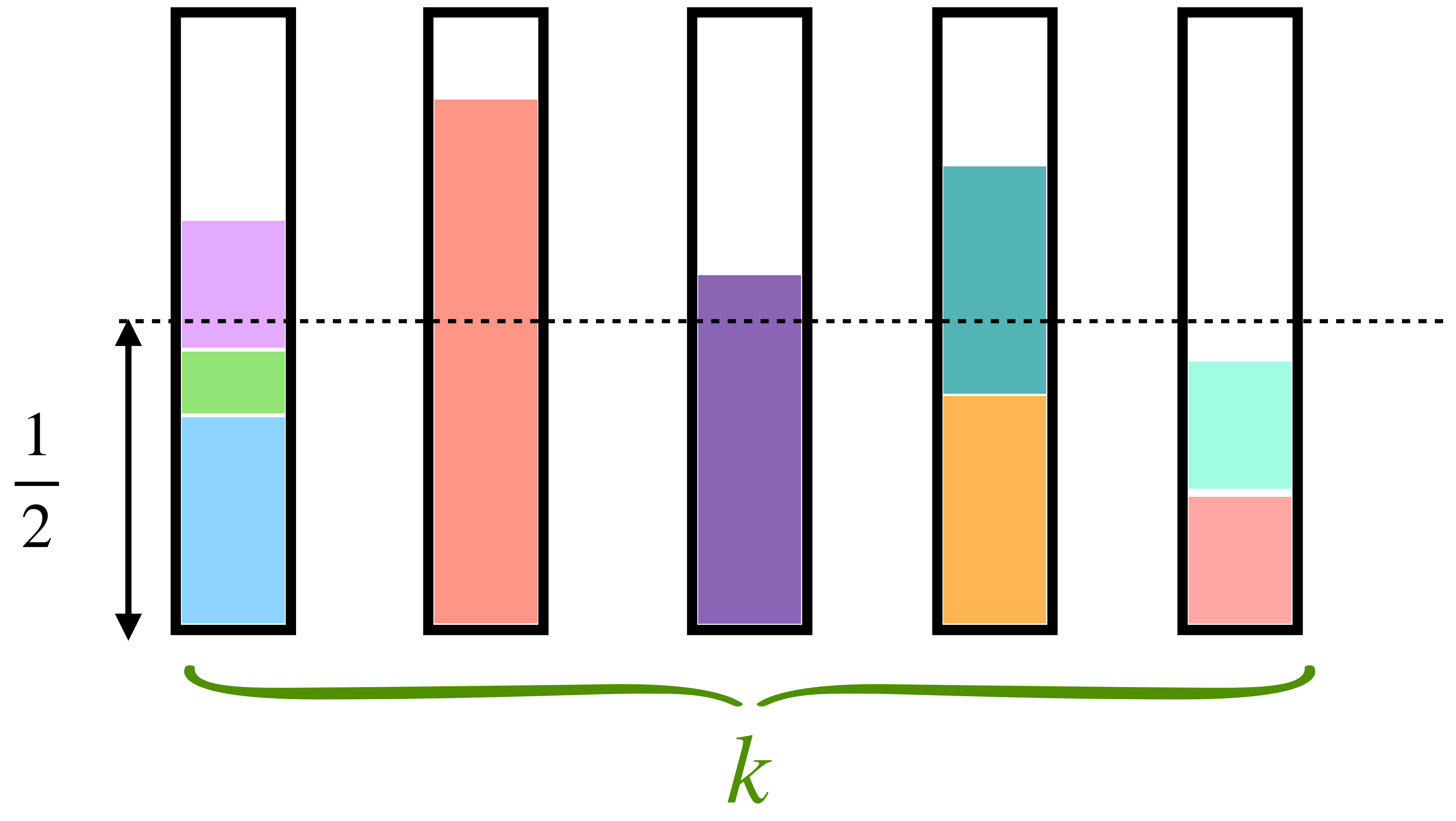




FirstFit $\leq 2 \cdot$ OPT $+ 1$

<Proof idea>

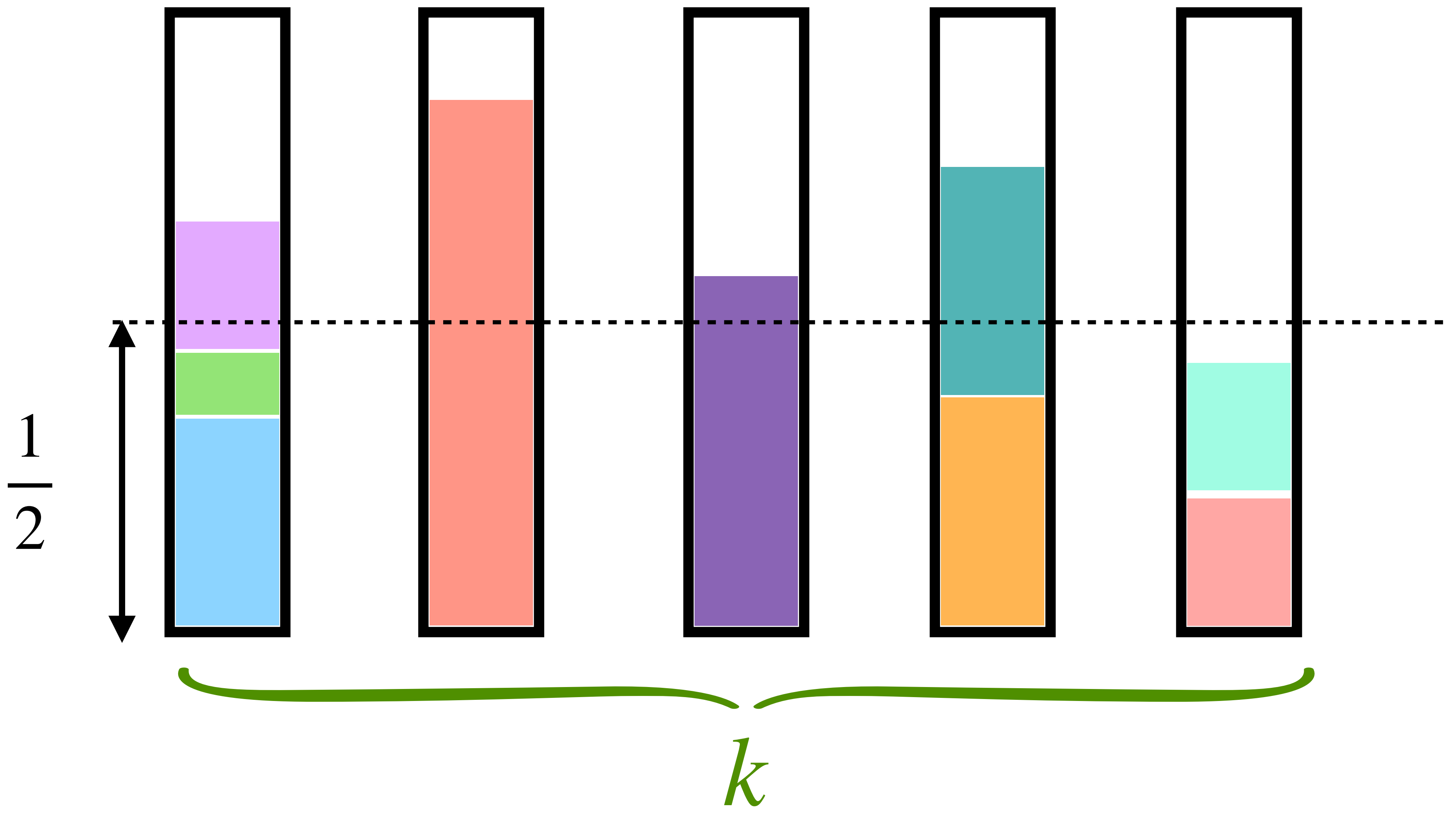
- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)
  - Let k be the number of bins opened by **FirstFit**. Total size of all items $\geq (k-1) \cdot \frac{1}{2}$
- OPT $\geq$ (total size of all items)/1 $\geq (k-1)/2 \iff$ **FirstFit** $= k \leq 2 \cdot$ OPT $+1$

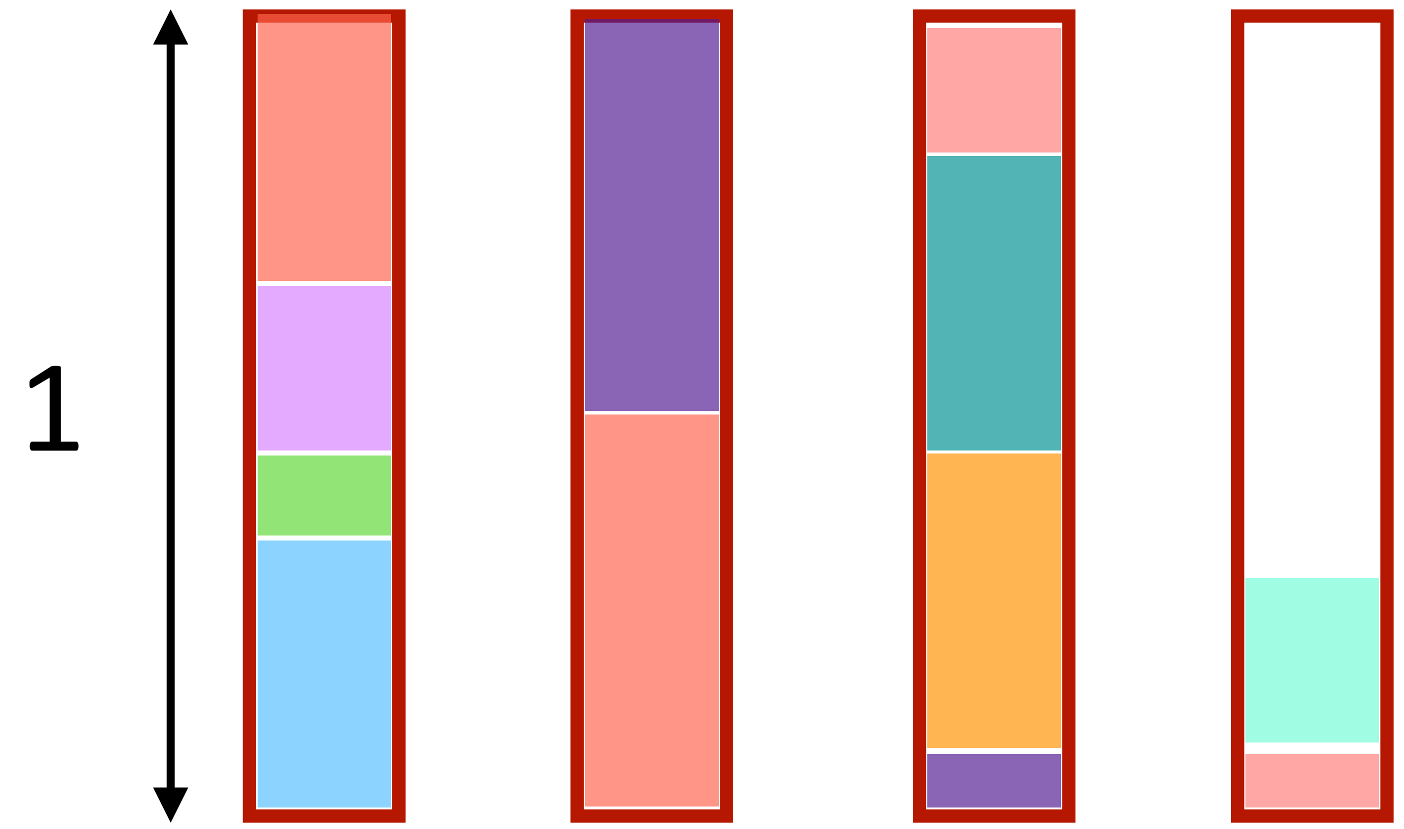


FirstFit $\leq 2 \cdot$ OPT $+ 1$

<Proof idea>

- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)
  - Let k be the number of bins opened by **FirstFit**. Total size of all items $\geq (k-1) \cdot \dfrac{1}{2}$
- OPT $\geq$ (total size of all items)/1 $\geq (k-1)/2 \iff$ **FirstFit** $= k \leq 2 \cdot$ OPT $+1$



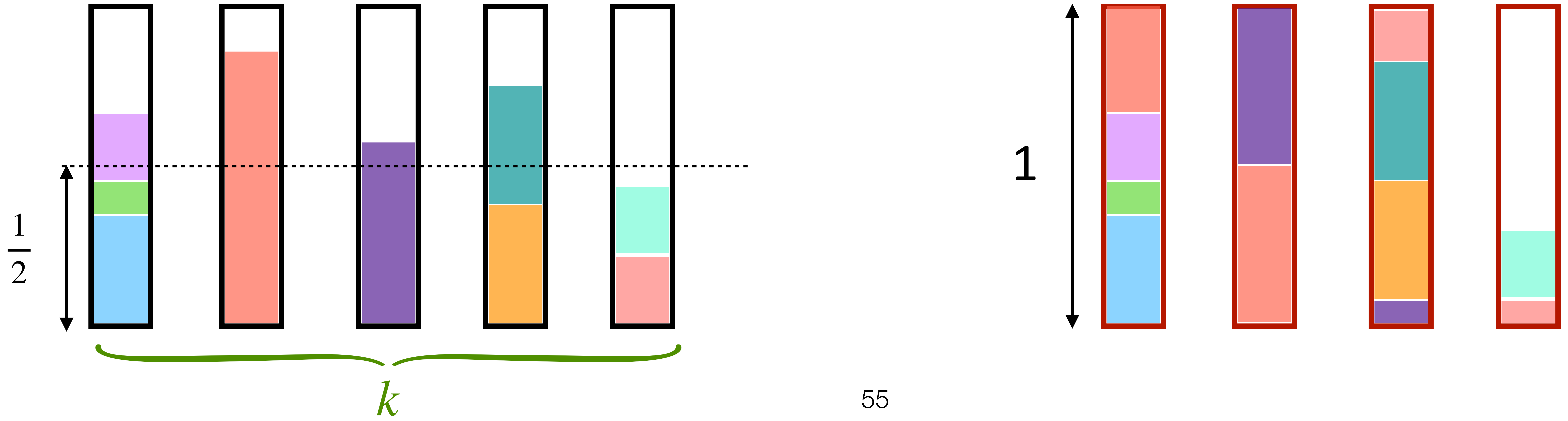


Even when the optimal algorithm has superpower to cut the items, it needs Total size of bins to accommodate all the items

FirstFit $\leq 2 \cdot \text{OPT} + 1$

<Proof idea>

- Observation: There is at most one bin at least *half empty* (that is, let s_j be the total size of the items in bin B_j. There is at most one bin B_j such that $s_j \leq 1/2$)
  - Let k be the number of bins opened by **FirstFit**. Total size of all items $\geq (k-1) \cdot \frac{1}{2}$
- OPT $\geq$ (total size of all items)/1 $\geq (k-1)/2 \iff$ **FirstFit** $= k \leq 2 \cdot$ OPT $+1$



$\square$

What Happened

- To analyze the competitive ratio of FirstFit, we assume that it takes k bins

- By the property of FirstFit, at most one bin is half-empty
  - The total size of all items is bounded by below, so is the number of bins optimal solution needs

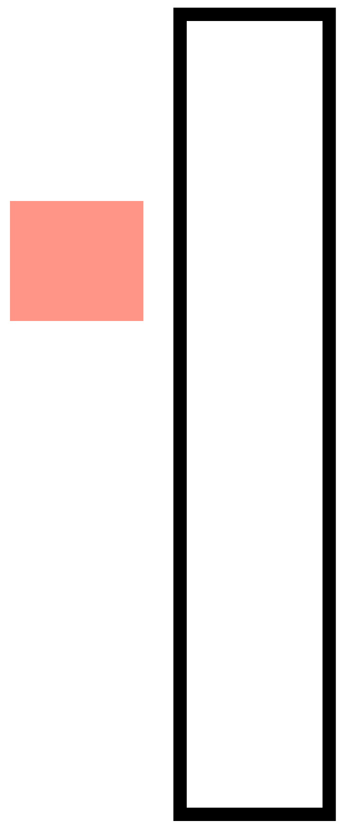
Outline

- **Bin Packing** problem
  - Assume that we know the **ALG** cost

- **Paging** problem
  - We know very little about the **ALG** or the **OPT**

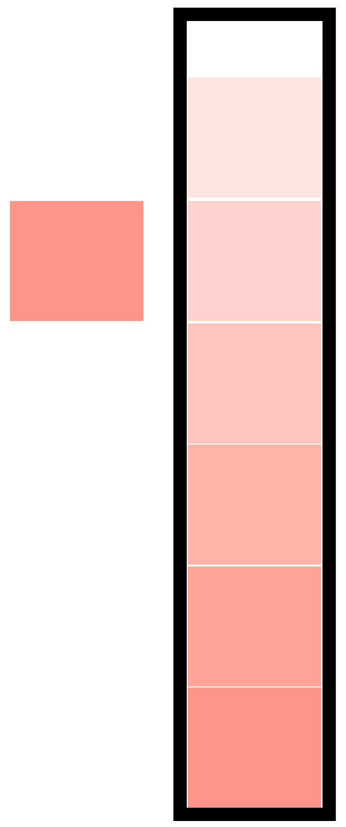
FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$



FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$

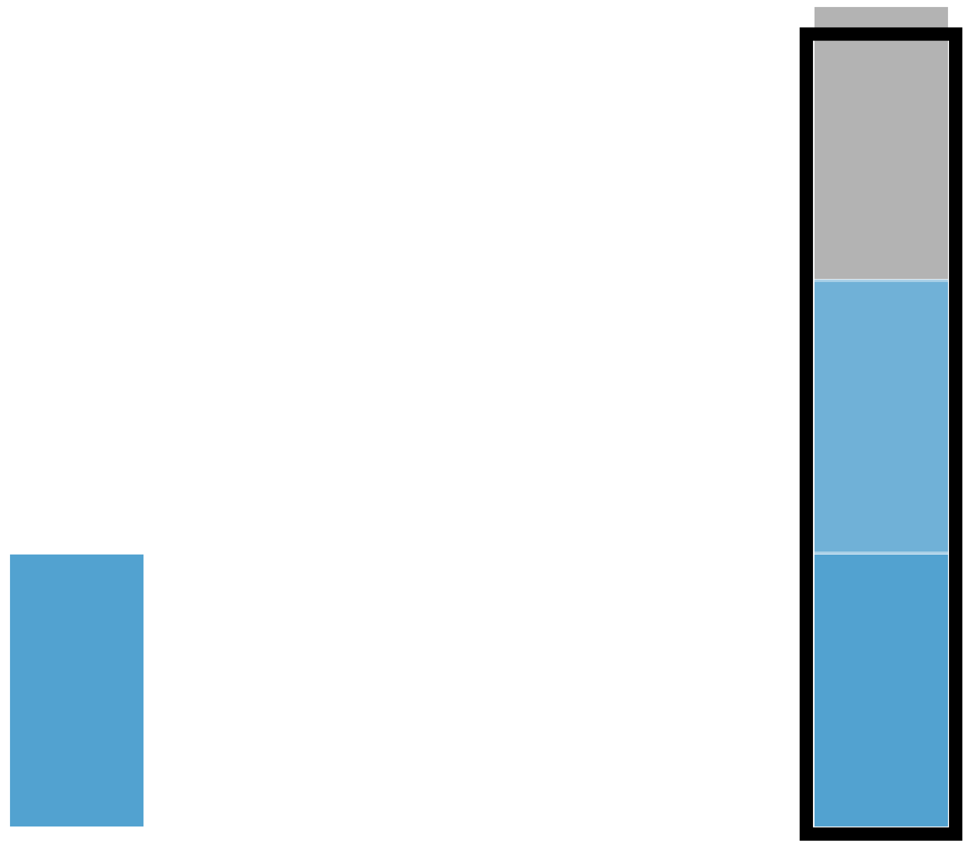


FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$

FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$

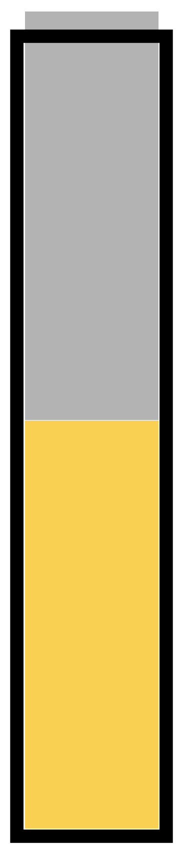


FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$
  - 6k items, each with size $\frac{1}{2} + \epsilon$

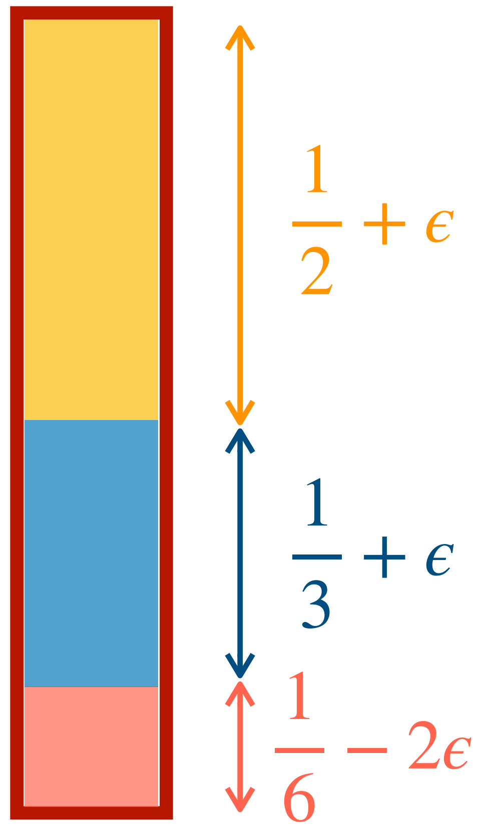
FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$
  - 6k items, each with size $\frac{1}{2} + \epsilon$



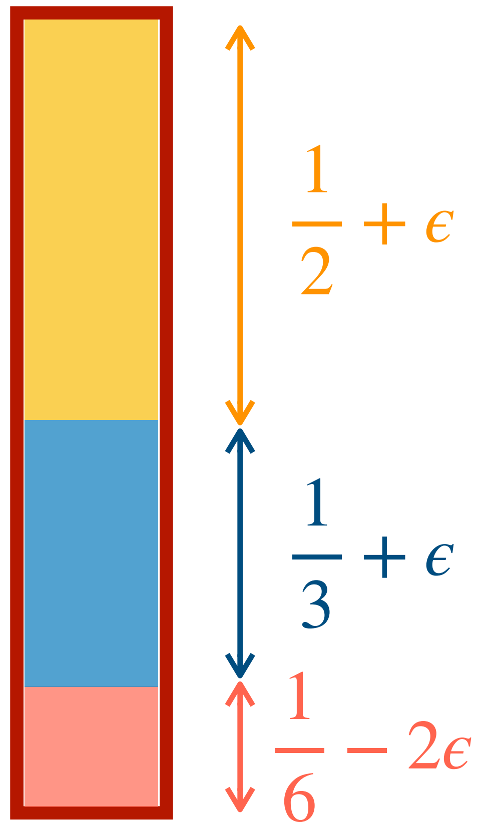
FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$
  - 6k items, each with size $\frac{1}{2} + \epsilon$



FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$
  - 6k items, each with size $\frac{1}{2} + \epsilon$

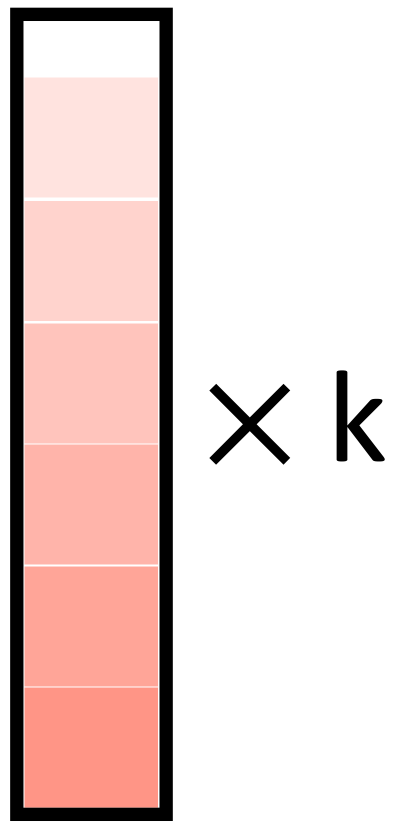


$\times$ 6k

OPT = 6k

FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$

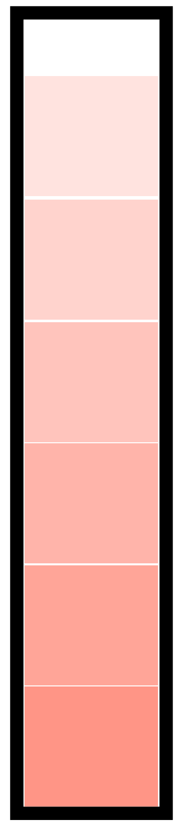


FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$

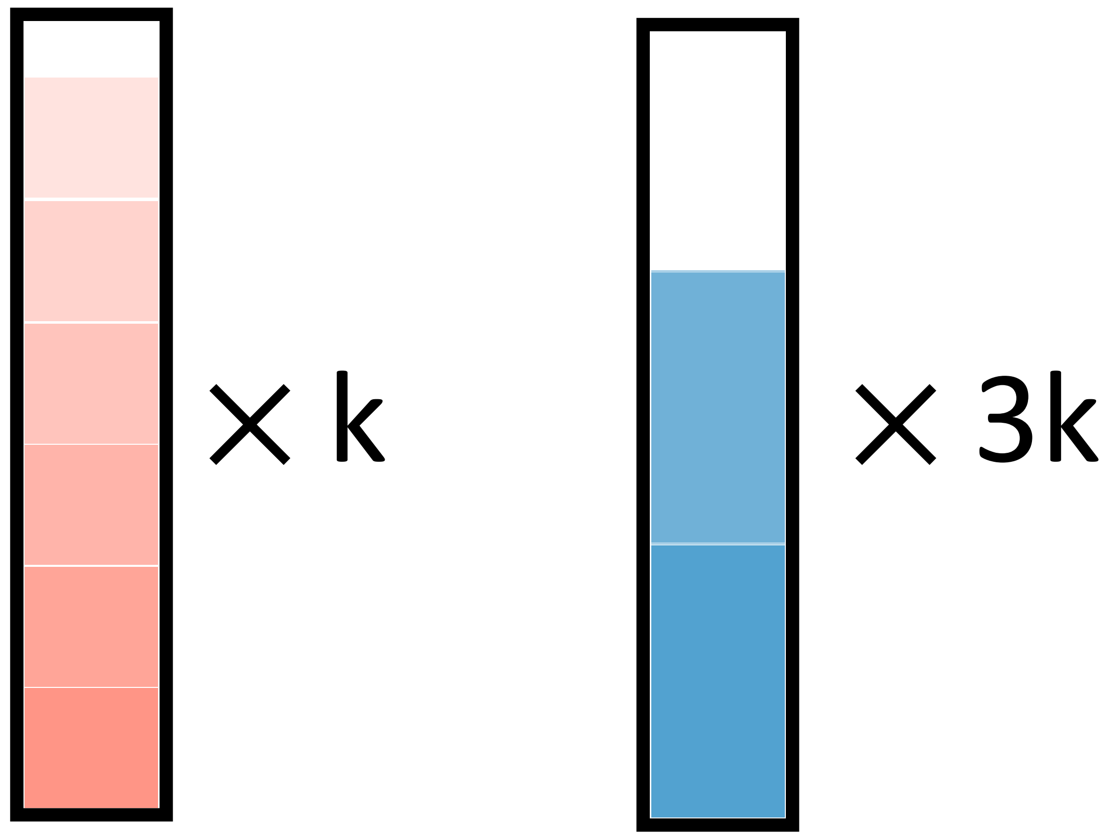
FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$


$\times$ k

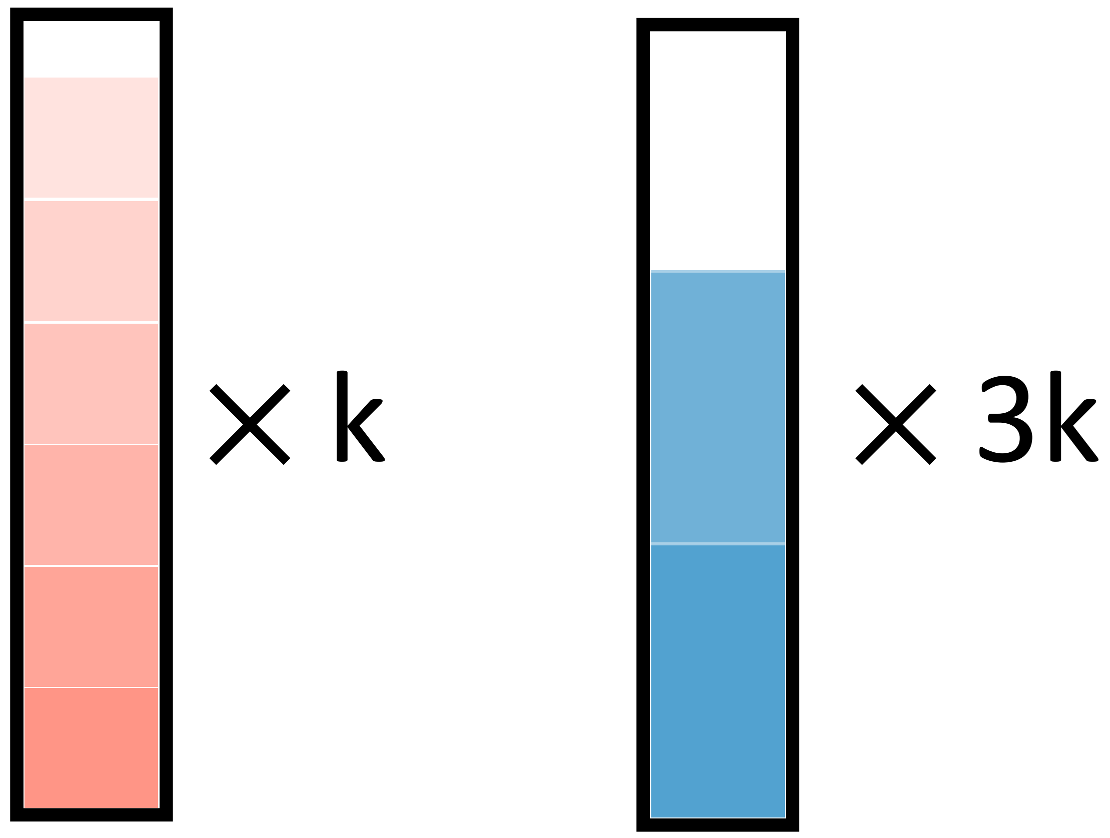
FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$



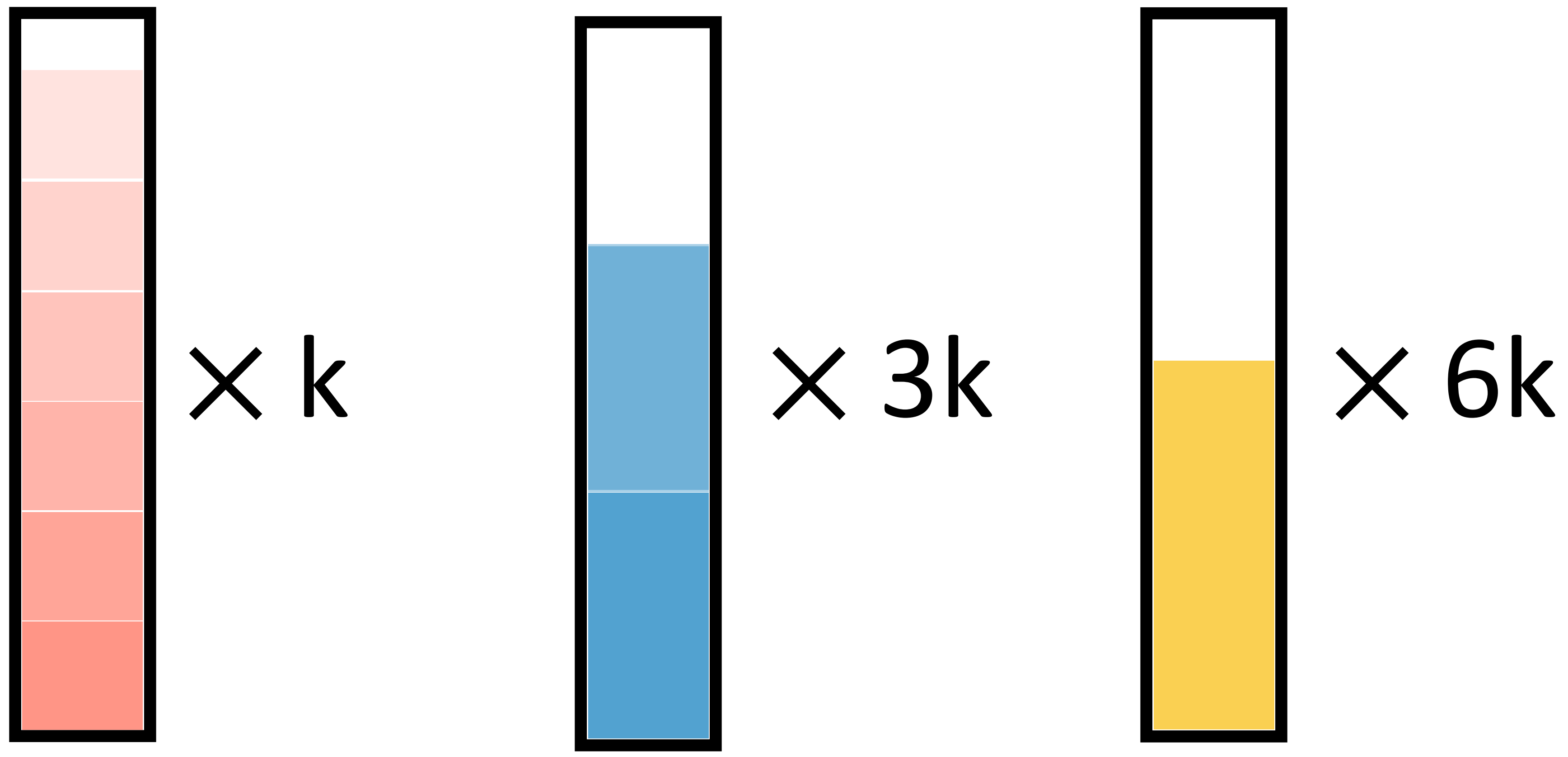
FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$
  - 6k items, each with size $\frac{1}{2} + \epsilon$



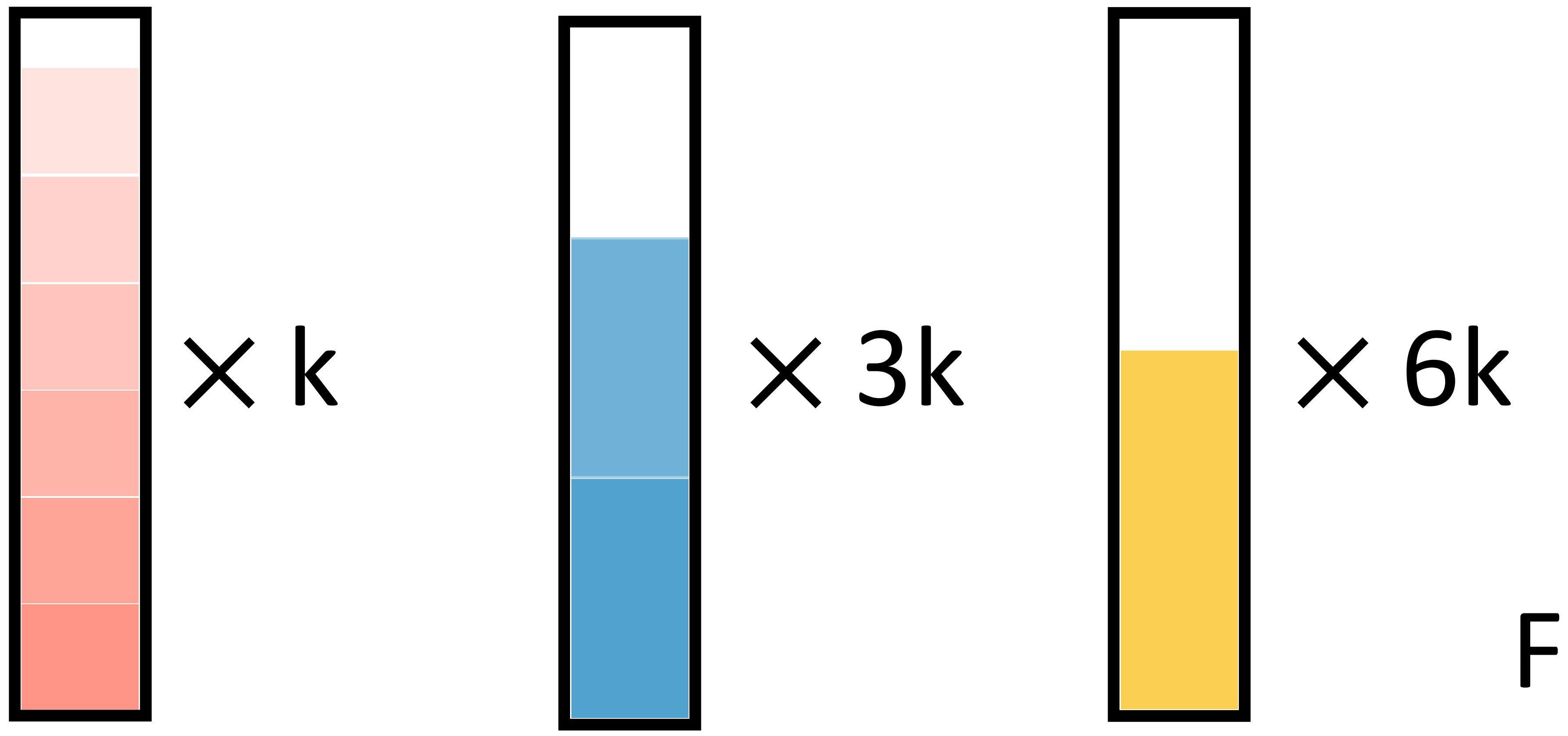
FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$
  - 6k items, each with size $\frac{1}{2} + \epsilon$



FirstFit is at least 1.667-competitive

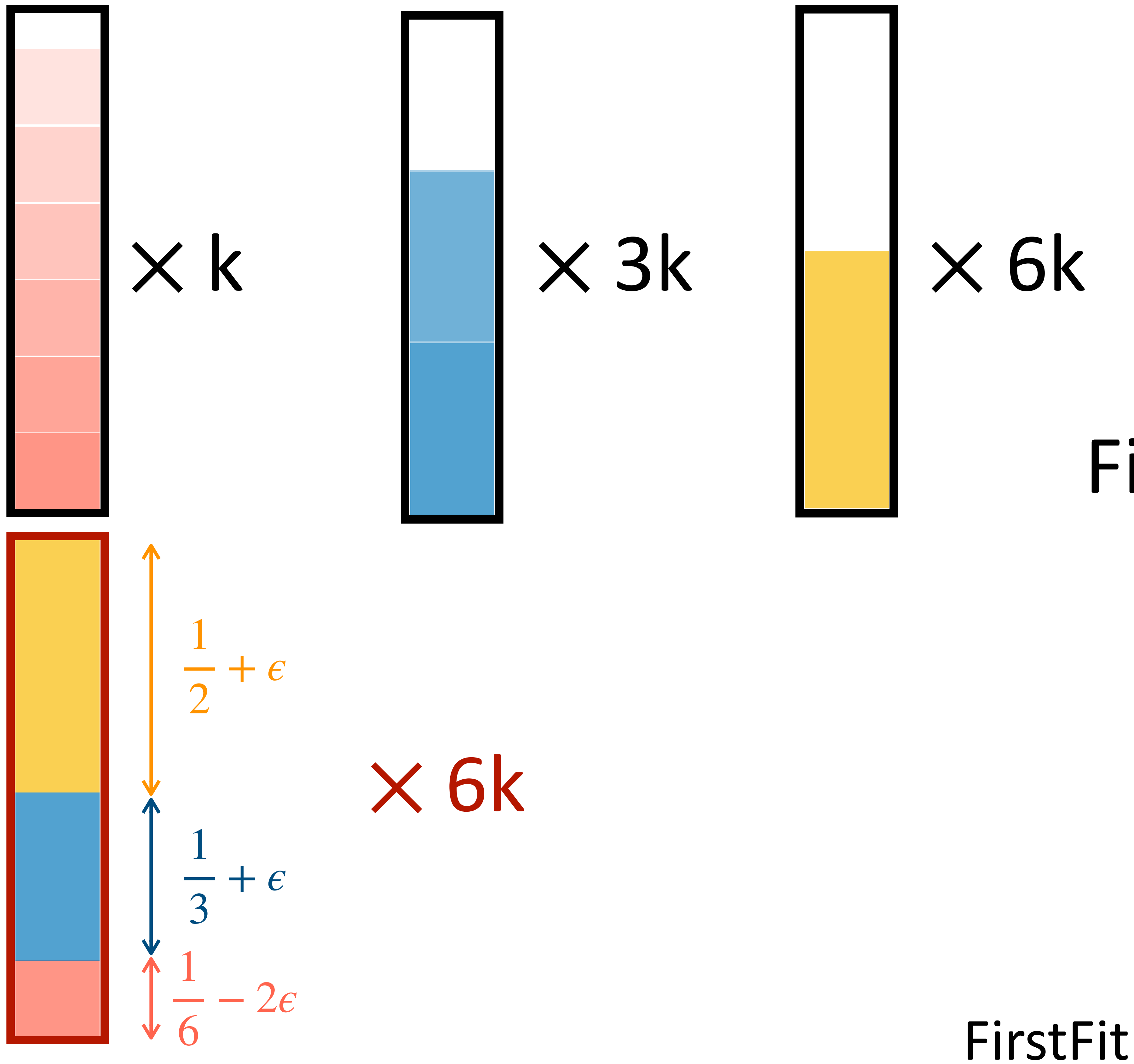
- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$
  - 6k items, each with size $\frac{1}{2} + \epsilon$



FirstFit = 10k

FirstFit is at least 1.667-competitive

- Consider a sequence of requests
  - 6k items, each with size $\frac{1}{6} - 2\epsilon$
  - 6k items, each with size $\frac{1}{3} + \epsilon$
  - 6k items, each with size $\frac{1}{2} + \epsilon$



FirstFit = 10k

OPT = 6k

$$\frac{\text{FirstFit}}{\text{OPT}} = \frac{10}{6} = 1.667$$

Outline

- **Bin Packing** problem
  - Assume that we know the **ALG** cost

- **Paging** problem
  - We know very little about the **ALG** or the **OPT**

Any deterministic online algorithm is at least 1.333-competitive

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea>

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance.

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea>
Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \dots, \frac{1}{2}-\epsilon}_{m}$$

OPT(I) = $\dfrac{m}{2}$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \dots, \frac{1}{2}-\epsilon}_{m}$$

OPT(I) = $\frac{m}{2}$

$\text{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2}$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}$$

OPT(I) = $\frac{m}{2}$

ALG(I) $< \frac{4}{3} \cdot \frac{m}{2}$ = $\frac{2}{3} \cdot m$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}$$

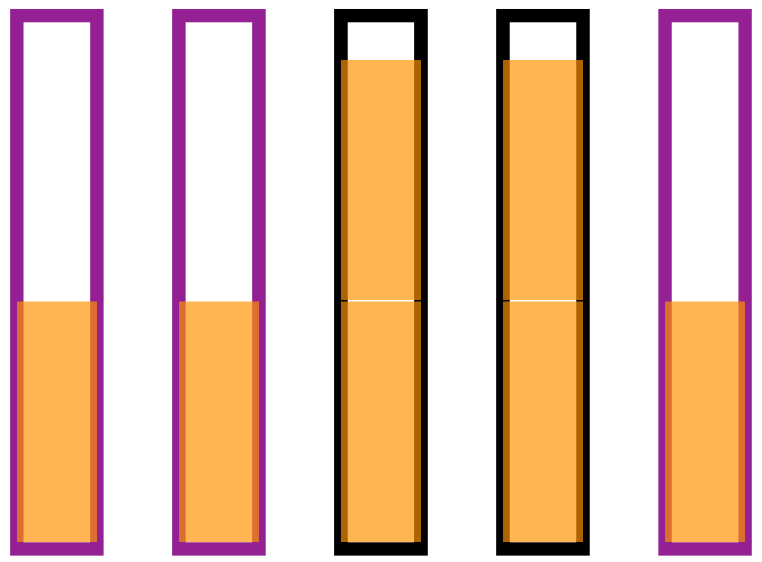
$$\mathrm{OPT}(I) = \frac{m}{2}$$

$$\mathrm{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2} = \frac{2}{3} \cdot m$$

$$= a_1 + a_2$$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)



Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}$$

$$\text{OPT}(I) = \frac{m}{2}$$

$$\boxed{\text{ALG}(I) < \frac{4}{3}\cdot\frac{m}{2}} = \frac{2}{3}\cdot m$$

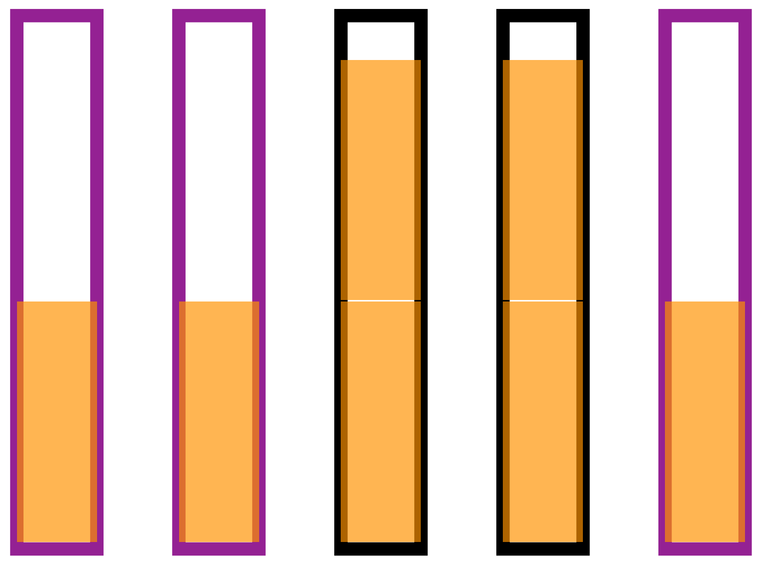
$$= a_1 + a_2$$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

There are m items



Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \dots, \frac{1}{2}-\epsilon}_{m}$$

$$\text{OPT}(I) = \frac{m}{2}$$

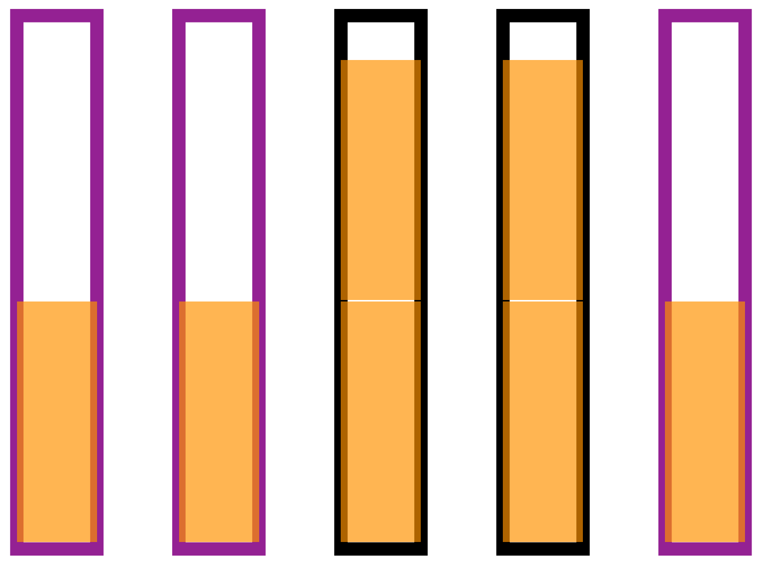
$$\text{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2} = \frac{2}{3} \cdot m$$

$$= a_1 + a_2 = m - a_2$$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$



Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

$$\text{OPT}(I) = \frac{m}{2}$$

$$\text{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2} = \frac{2}{3} \cdot m$$

$$= a_1 + a_2 = m - a_2$$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is $(4/3-\epsilon)$-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is $(4/3-\epsilon)$-competitive for the first half of the instance, it cannot be $(4/3-\epsilon)$-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

$$\mathrm{OPT}(I) = \frac{m}{2}$$

$$\boxed{\mathrm{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2}} = \frac{2}{3} \cdot m$$

$$= a_1 + a_2 = m - a_2$$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

OPT(I) = $\frac{m}{2}$

OPT(I+I) = m

$$\boxed{\text{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2}} = \frac{2}{3} \cdot m$$

$$= a_1 + a_2 = m - a_2$$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

OPT(I) = $\dfrac{m}{2}$

OPT(I+I) = m

$$\boxed{\text{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2}} = \frac{2}{3} \cdot m$$

$$= a_1 + a_2 = m - a_2$$

ALG(I+I) = $a_1 + a_2 + x$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

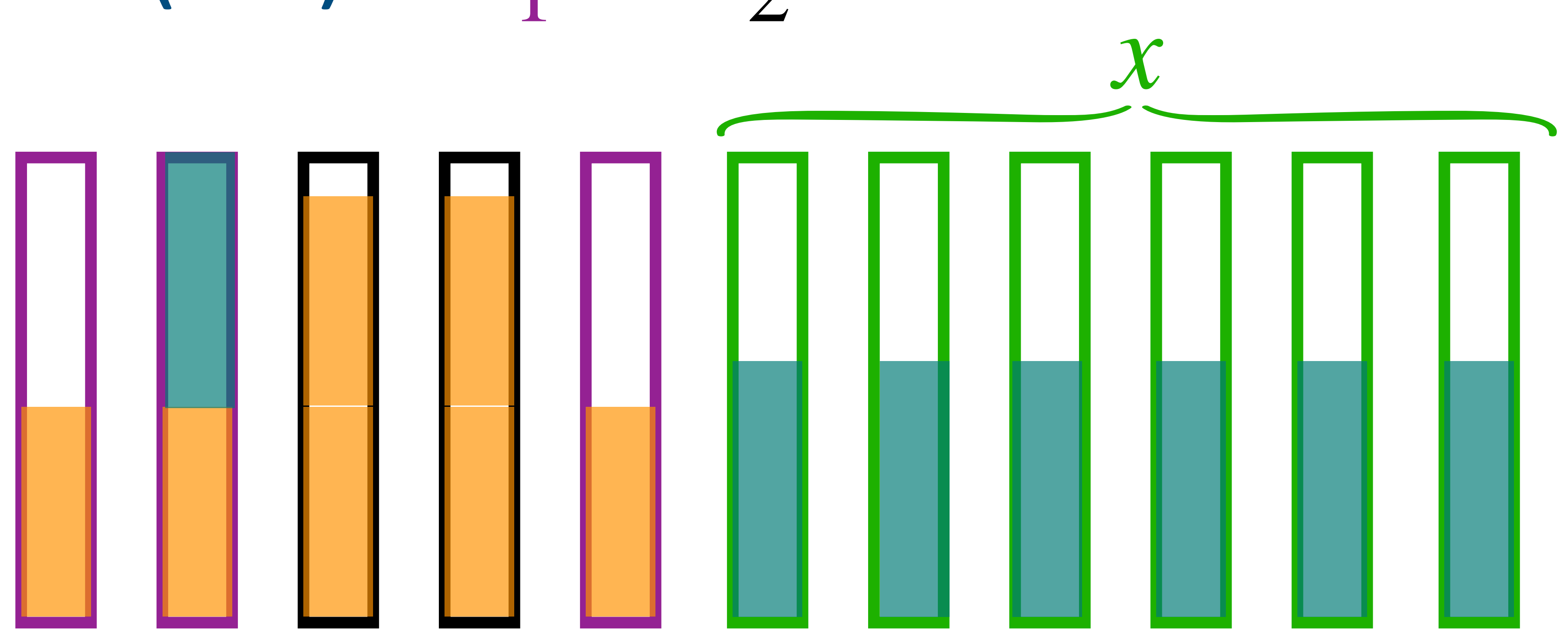
OPT(I) = $\dfrac{m}{2}$

OPT(I+I) = m

$$\text{ALG}(I) < \frac{4}{3}\cdot\frac{m}{2} = \frac{2}{3}\cdot m$$

$$= a_1 + a_2 = m - a_2$$

ALG(I+I) = $a_1 + a_2 + x$



a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

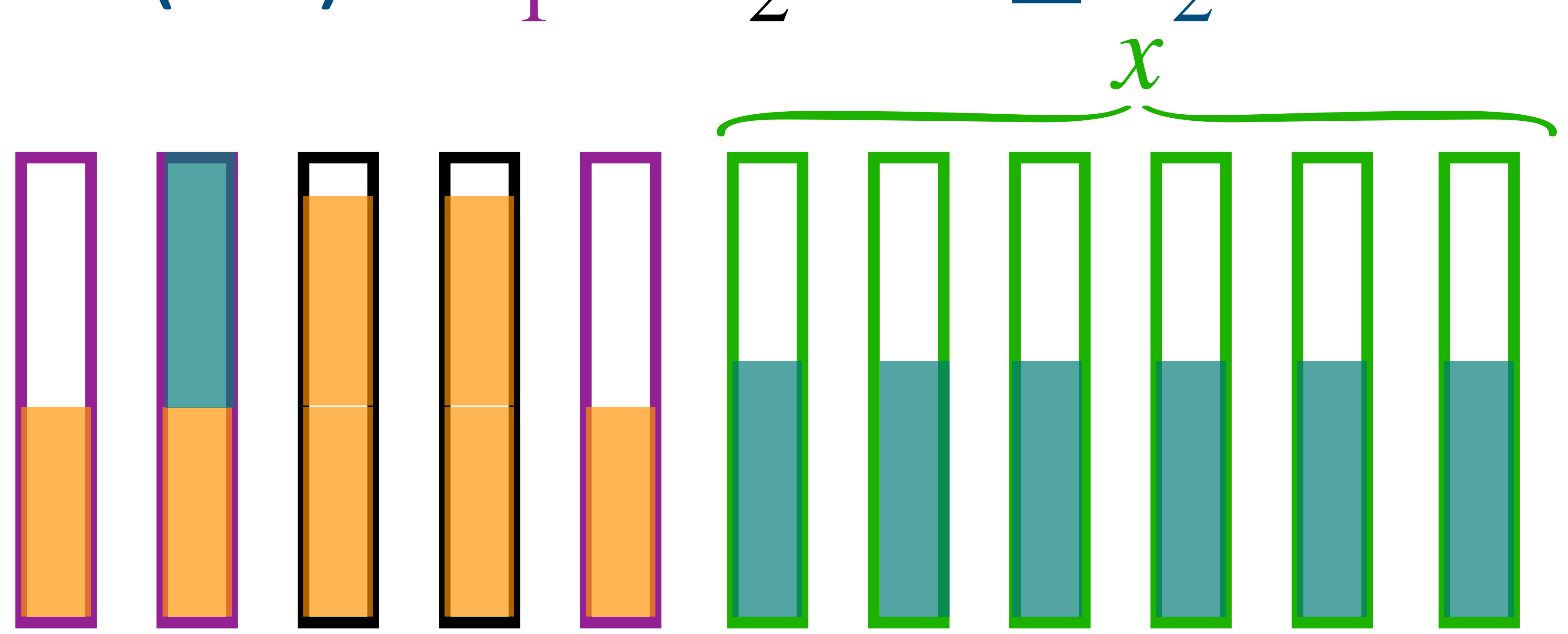
OPT(I) = $\dfrac{m}{2}$

OPT(I+I) = m

$$\text{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2} = \frac{2}{3} \cdot m$$

$$= a_1 + a_2 = m - a_2$$

$$\text{ALG}(I+I) = a_1 + a_2 + x \geq a_2 + m$$



a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

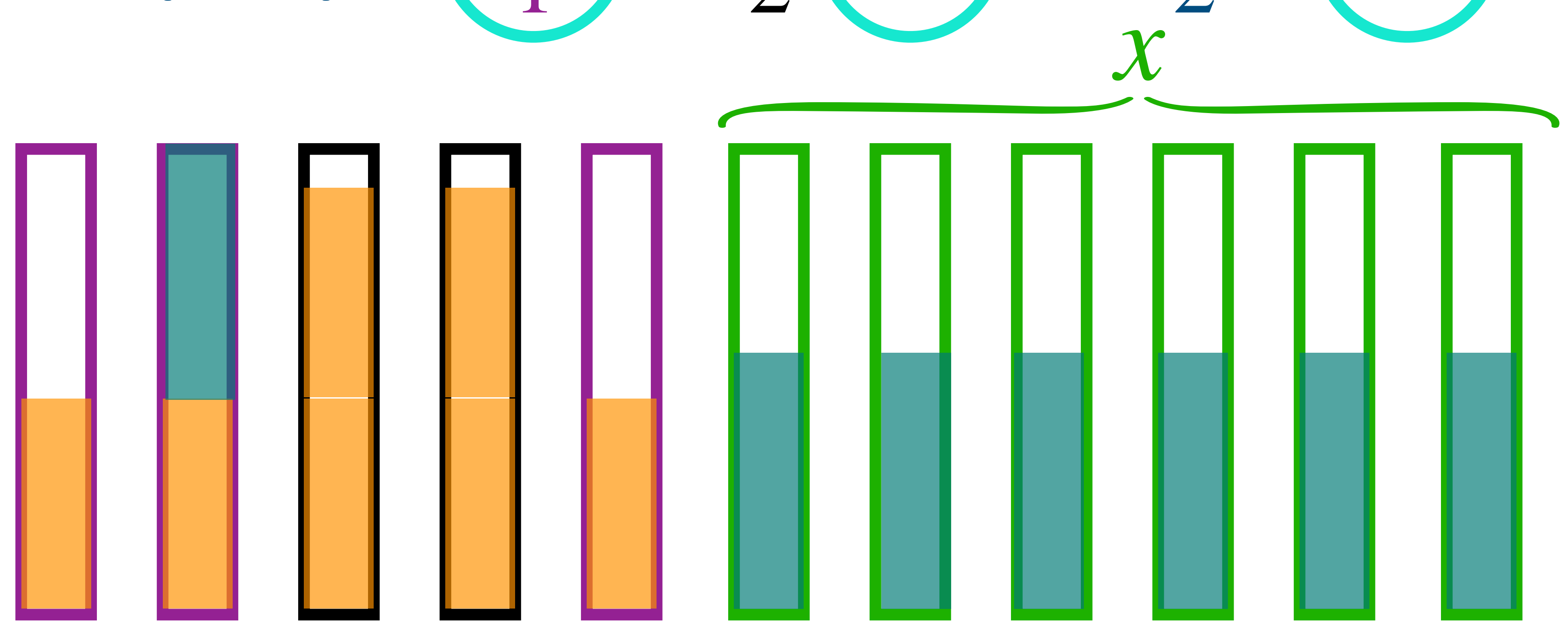
OPT(I) = $\dfrac{m}{2}$

OPT(I+I) = m

$$\boxed{\text{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2}} = \frac{2}{3} \cdot m$$

$$= a_1 + a_2 = m - a_2$$

ALG(I+I) = $a_1 + a_2 + x \geq a_2 + m$



a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

OPT(I) = $\frac{m}{2}$

OPT(I+I) = m

ALG(I) $< \frac{4}{3} \cdot \frac{m}{2} = \frac{2}{3} \cdot m$

$= a_1 + a_2 = m - a_2$

ALG(I+I) = $a_1 + a_2 + x \geq a_2 + m$

ALG(I+I) $< \frac{4}{3} \cdot$ OPT(I+I)

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

$$\text{OPT}(I) = \frac{m}{2}$$

$$\text{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2} = \frac{2}{3} \cdot m$$

$$= a_1 + a_2 = m - a_2$$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

$$\text{OPT}(I+I) = m$$

$$\text{ALG}(I+I) = a_1 + a_2 + x \geq a_2 + m$$

$$\text{ALG}(I+I) < \frac{4}{3} \cdot \text{OPT}(I+I) = \frac{4}{3} \cdot m$$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

$$\mathrm{OPT}(I) = \frac{m}{2}$$

$$\mathrm{OPT}(I+I) = m$$

$$\mathrm{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2} = \frac{2}{3} \cdot m$$

$$= a_1 + a_2 = m - a_2$$

$$\mathrm{ALG}(I+I) = a_1 + a_2 + x \geq a_2 + m$$

$$\mathrm{ALG}(I+I) < \frac{4}{3} \cdot \mathrm{OPT}(I+I) = \frac{4}{3} \cdot m$$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is $(4/3-\epsilon)$-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is $(4/3-\epsilon)$-competitive for the first half of the instance, it cannot be $(4/3-\epsilon)$-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

$\mathrm{OPT}(I) = \dfrac{m}{2}$

$\mathrm{OPT}(I+I) = m$

$$\mathrm{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2} = \frac{2}{3} \cdot m$$

$$= a_1 + a_2 = m - a_2$$

a_1: #bins with 1 item in $\mathrm{ALG}(I)$

a_2: #bins with 2 items in $\mathrm{ALG}(I)$

$m = a_1 + 2a_2$

$$\mathrm{ALG}(I+I) = a_1 + a_2 + x \geq a_2 + m$$

$$\mathrm{ALG}(I+I) < \frac{4}{3} \cdot \mathrm{OPT}(I+I) = \frac{4}{3} \cdot m$$

$$a_2 < \frac{m}{3}$$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

OPT(I) = $\frac{m}{2}$

ALG(I) $< \frac{4}{3} \cdot \frac{m}{2} = \frac{2}{3} \cdot m$

$= a_1 + a_2 = m - a_2$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

OPT(I+I) = m

ALG(I+I) = $a_1 + a_2 + x \geq a_2 + m$

ALG(I+I) $< \frac{4}{3} \cdot$ OPT(I+I) $= \frac{4}{3} \cdot m$

$a_2 < \frac{m}{3} \Longleftrightarrow$ ALG(I) = $m - a_2 > \frac{2}{3} \cdot m$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

OPT(I) = $\frac{m}{2}$

OPT(I+I) = m

ALG(I) $< \frac{4}{3} \cdot \frac{m}{2} = \frac{2}{3} \cdot m$

$= a_1 + a_2 = m - a_2$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

ALG(I+I) = $a_1 + a_2 + x \geq a_2 + m$

ALG(I+I) $< \frac{4}{3} \cdot$ OPT(I+I) $= \frac{4}{3} \cdot m$

$a_2 < \frac{m}{3} \Longleftrightarrow$ ALG(I) = $m - a_2 > \frac{2}{3} \cdot m$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is ($4/3-\epsilon$)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is ($4/3-\epsilon$)-competitive for the first half of the instance, it cannot be ($4/3-\epsilon$)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

OPT(I) = $\frac{m}{2}$

OPT(I+I) = m

$$\boxed{\text{ALG}(I) < \frac{4}{3} \cdot \frac{m}{2} = \frac{2}{3} \cdot m}$$

$$= a_1 + a_2 = m - a_2$$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

$$\text{ALG}(I+I) = a_1 + a_2 + x \geq a_2 + m$$

$$\text{ALG}(I+I) < \frac{4}{3} \cdot \text{OPT}(I+I) = \frac{4}{3} \cdot m$$

Contradiction!

$$a_2 < \frac{m}{3} \Longleftrightarrow \boxed{\text{ALG}(I) = m - a_2 > \frac{2}{3} \cdot m}$$

Any deterministic online algorithm is at least 1.333-competitive

<Proof idea> Assume ALG is (4/3-ϵ)-competitive

Prove by contradiction: design an instance such that any algorithm ALG that is (4/3-ϵ)-competitive for the first half of the instance, it cannot be (4/3-ϵ)-competitive for the whole instance. Consider the adversarial input:

$$\underbrace{\frac{1}{2}-\epsilon, \frac{1}{2}-\epsilon, \cdots, \frac{1}{2}-\epsilon}_{m}, \underbrace{\frac{1}{2}+\epsilon, \frac{1}{2}+\epsilon, \cdots, \frac{1}{2}+\epsilon}_{m}$$

OPT(I) = $\dfrac{m}{2}$

OPT(I+I) = m

ALG(I) $< \dfrac{4}{3} \cdot \dfrac{m}{2} = \dfrac{2}{3} \cdot m$

$= a_1 + a_2 = m - a_2$

a_1: #bins with 1 item in ALG(I)

a_2: #bins with 2 items in ALG(I)

$m = a_1 + 2a_2$

ALG(I+I) = $a_1 + a_2 + x \geq a_2 + m$

ALG(I+I) $< \dfrac{4}{3} \cdot$ OPT(I+I) $= \dfrac{4}{3} \cdot m$

$a_2 < \dfrac{m}{3} \Longleftrightarrow$ ALG(I) $= m - a_2 > \dfrac{2}{3} \cdot m$ $\square$

Outline

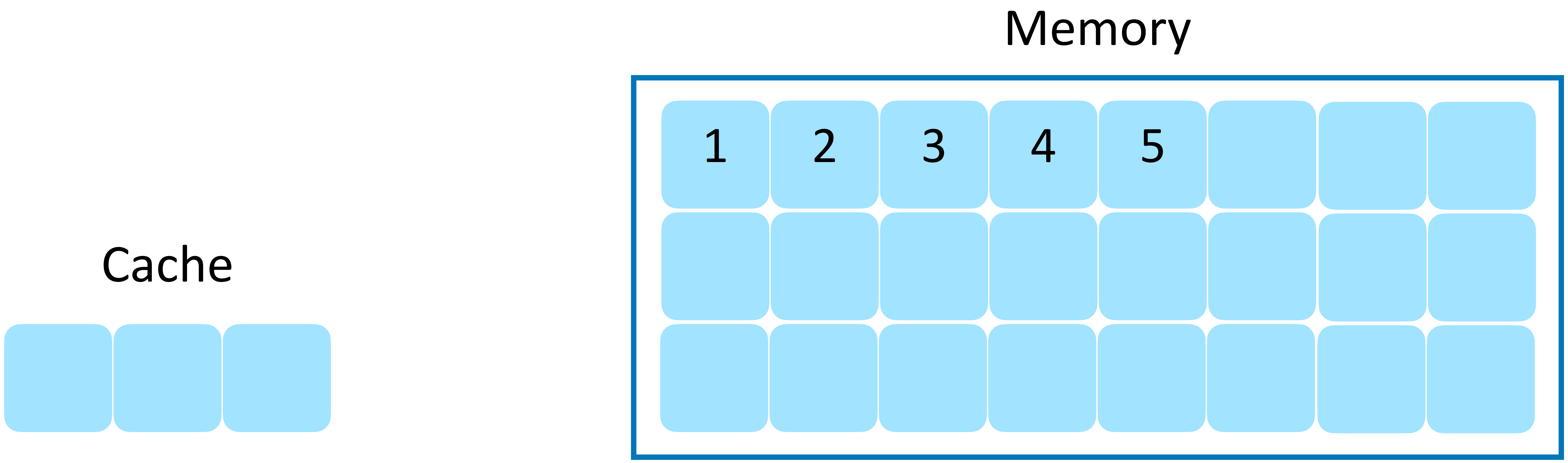
- **Bin Packing** problem
  - Assume that we know the **ALG** cost

- **Paging** problem
  - We know very little about the **ALG** or the **OPT**

Paging

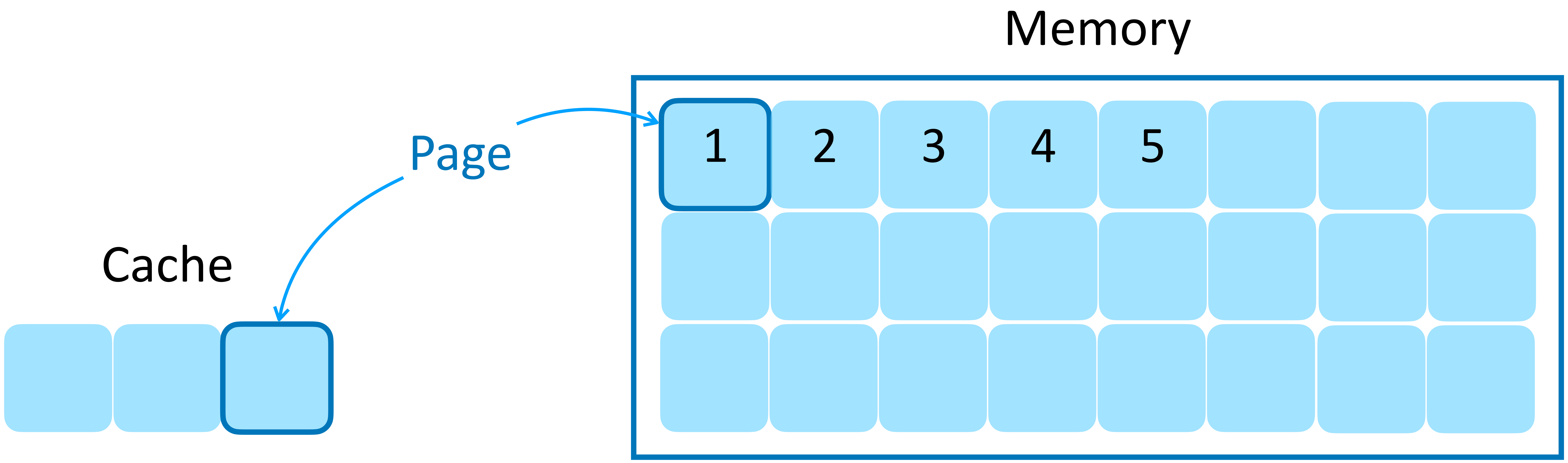
Paging

- In computer systems, the memory system is two-level
  - Data (in blocks) are stored in the **memory**, which cannot accessed directly
    - A block of data is called a page
  - The **cache** can be accessed directly and is fast, but very small



Paging

- In computer systems, the memory system is two-level
  - Data (in blocks) are stored in the **memory**, which cannot accessed directly
    - A block of data is called a **page**
  - The **cache** can be accessed directly and is fast, but very small

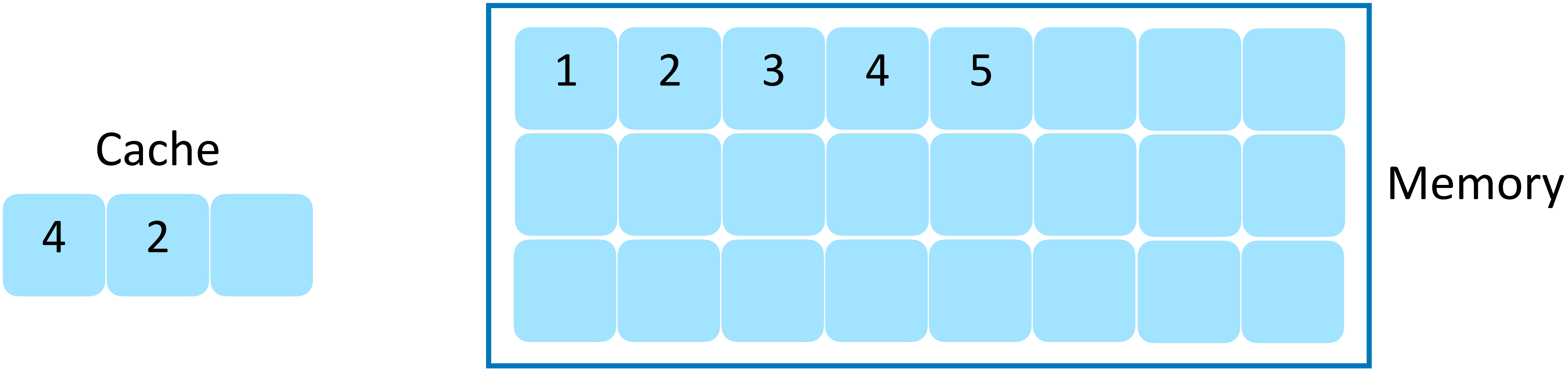


Paging

- When the processor needs a page p_i …
  - If p_i is in the cache (called a **hit**), the system needs not do anything
  - If p_i is not in the cache (a **miss**), the system incurs one **page fault** and must copy the page p_i from the slow memory to the fast cache

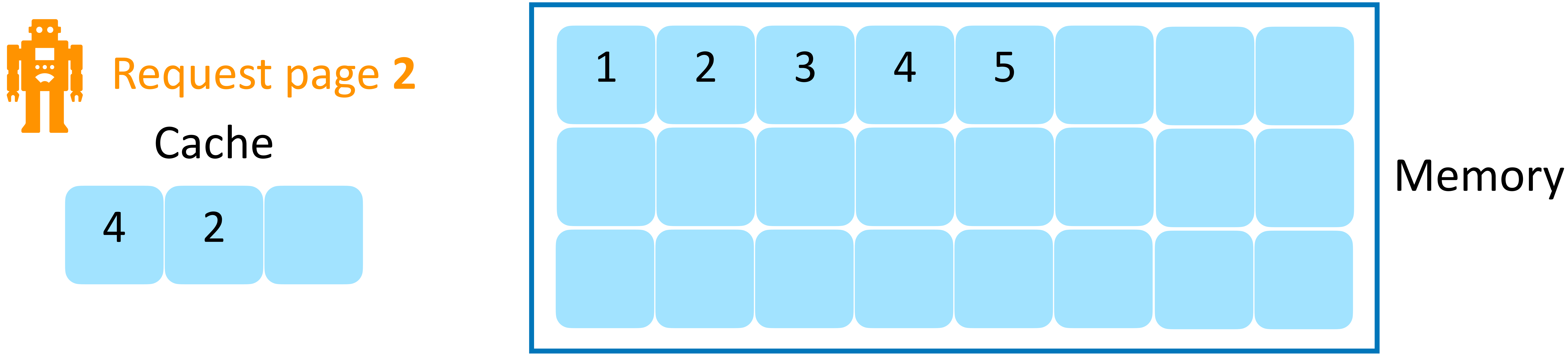
Paging

- When the processor needs a page p_i …
  - If p_i is in the cache (called a **hit**), the system needs not do anything
  - If p_i is not in the cache (a **miss**), the system incurs one **page fault** and must copy the page p_i from the slow memory to the fast cache



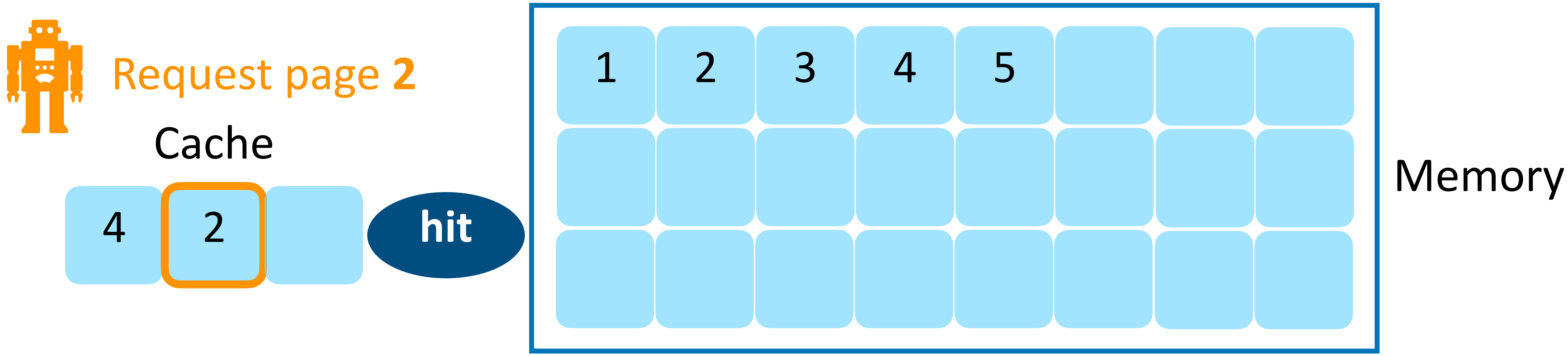
Paging

- When the processor needs a page p_i …
  - If p_i is in the cache (called a **hit**), the system needs not do anything
  - If p_i is not in the cache (a **miss**), the system incurs one **page fault** and must copy the page p_i from the slow memory to the fast cache



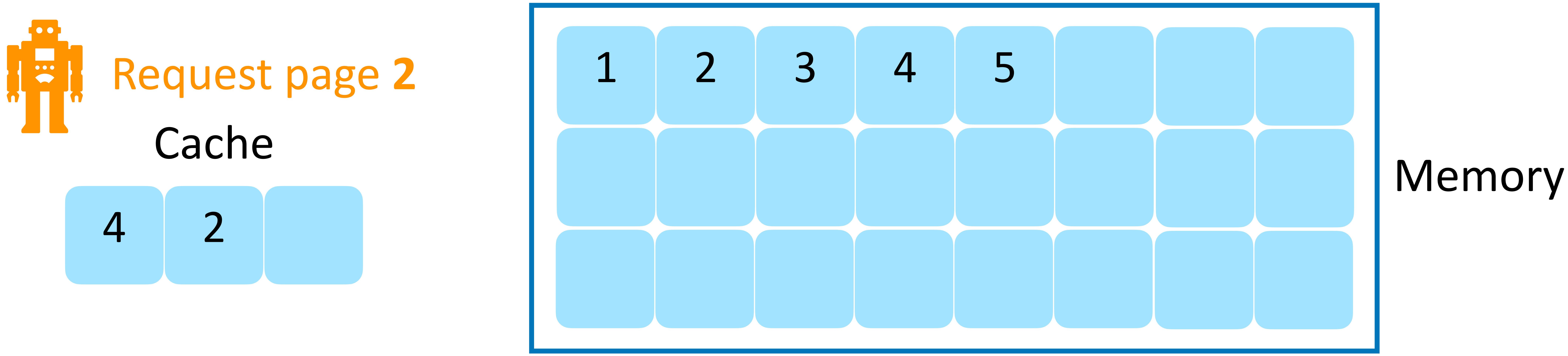
Paging

- When the processor needs a page p_i …
  - If p_i is in the cache (called a **hit**), the system needs not do anything
  - If p_i is not in the cache (a **miss**), the system incurs one **page fault** and must copy the page p_i from the slow memory to the fast cache



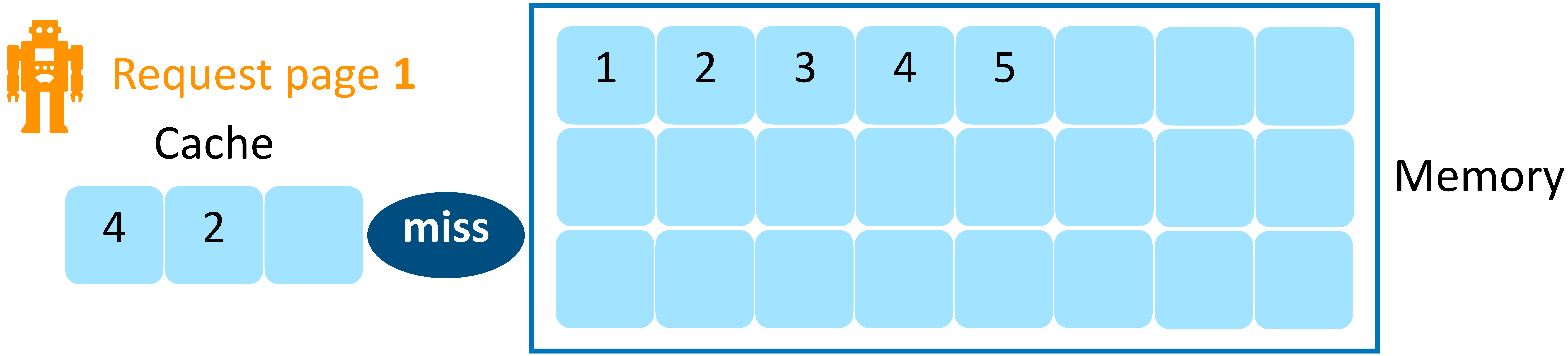
Paging

- When the processor needs a page p_i …
  - If p_i is in the cache (called a **hit**), the system needs not do anything
  - If p_i is not in the cache (a **miss**), the system incurs one **page fault** and must copy the page p_i from the slow memory to the fast cache



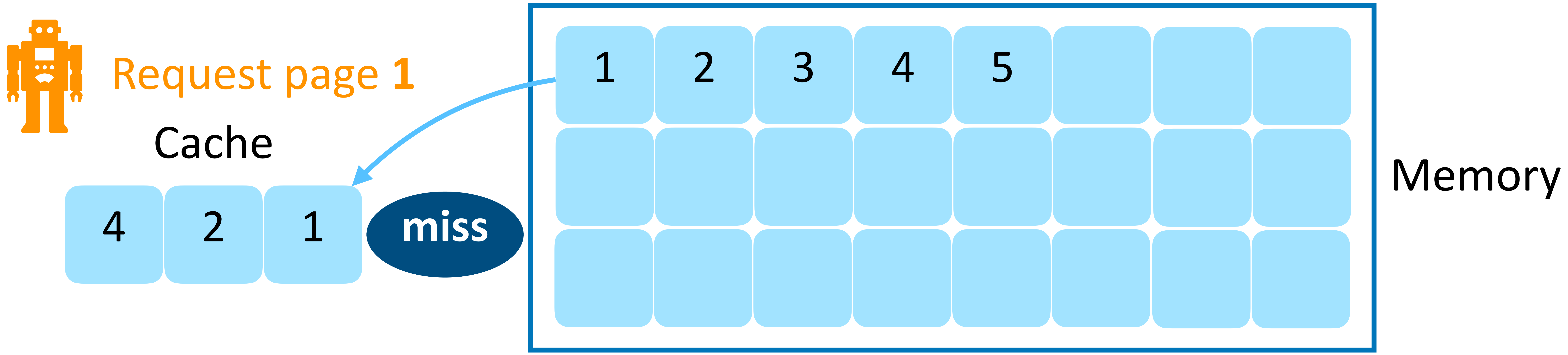
Paging

- When the processor needs a page p_i …
  - If p_i is in the cache (called a **hit**), the system needs not do anything
  - If p_i is not in the cache (a **miss**), the system incurs one **page fault** and must copy the page p_i from the slow memory to the fast cache



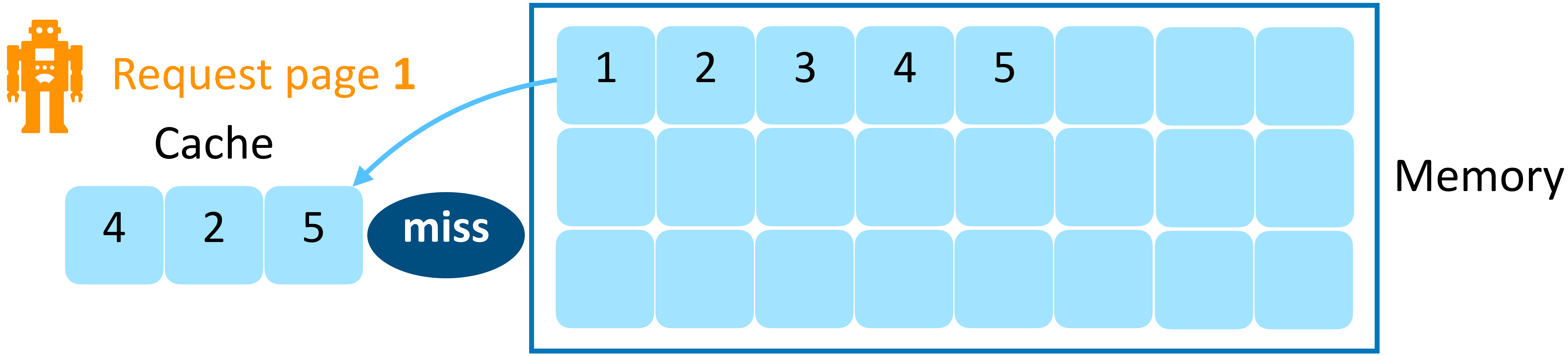
Paging

- When the processor needs a page p_i …
  - If p_i is in the cache (called a **hit**), the system needs not do anything
  - If p_i is not in the cache (a **miss**), the system incurs one **page fault** and must copy the page p_i from the slow memory to the fast cache



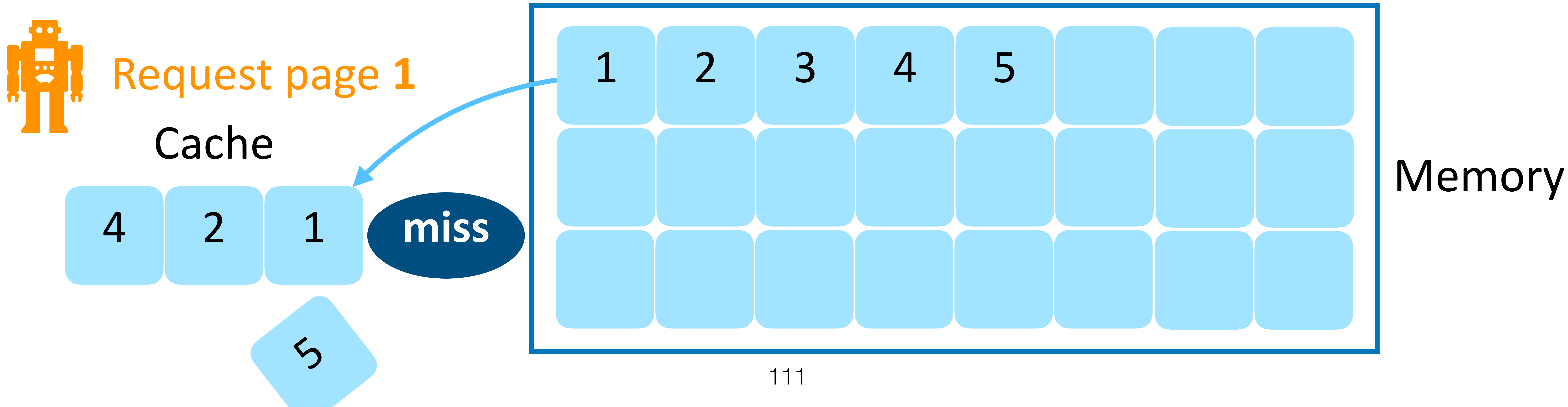
Paging

- When the processor needs a page p_i …
  - If p_i is in the cache (called a **hit**), the system needs not do anything
  - If p_i is not in the cache (a **miss**), the system incurs one **page fault** and must copy the page p_i from the slow memory to the fast cache
    - If the cache is full, one of the pages in the cache has to be evicted



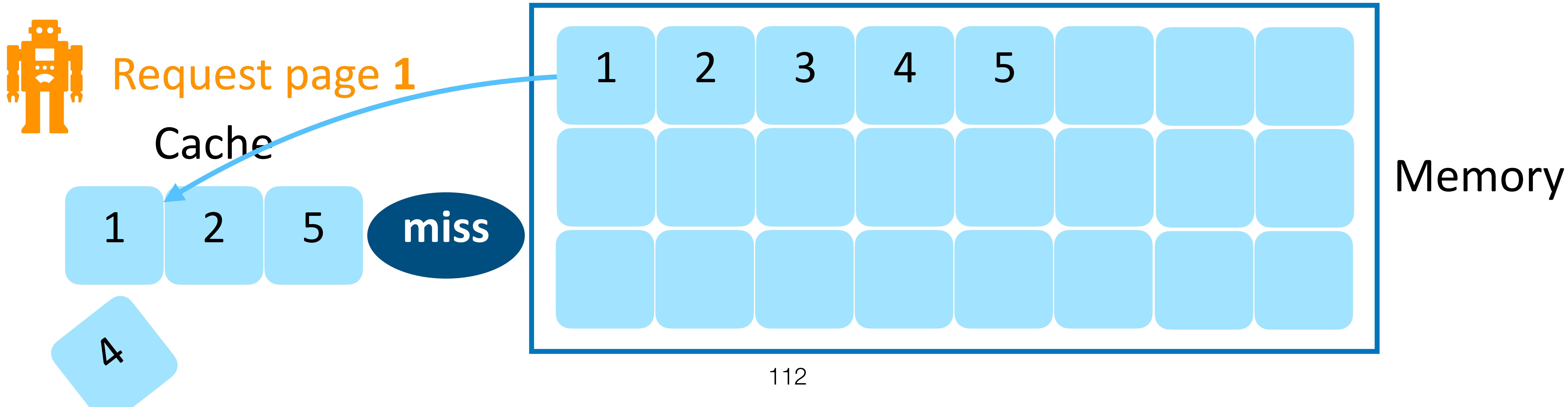
Paging

- When the processor needs a page p_i …
  - If p_i is in the cache (called a **hit**), the system needs not do anything
  - If p_i is not in the cache (a **miss**), the system incurs one **page fault** and must copy the page p_i from the slow memory to the fast cache
    - If the cache is full, one of the pages in the cache has to be evicted



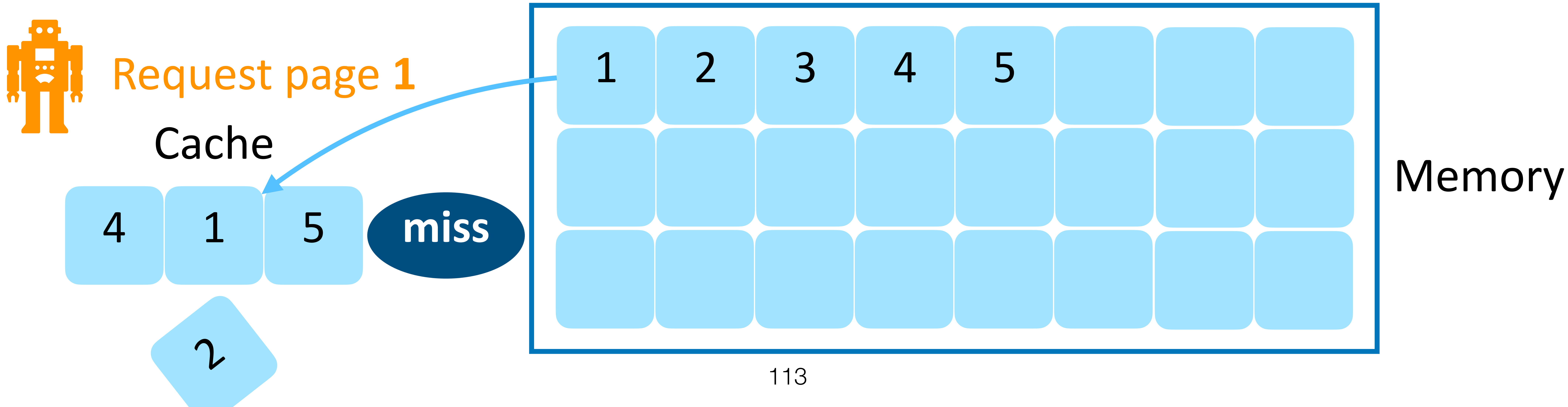
Paging

- When the processor needs a page p_i …
  - If p_i is in the cache (called a **hit**), the system needs not do anything
  - If p_i is not in the cache (a **miss**), the system incurs one **page fault** and must copy the page p_i from the slow memory to the fast cache
    - If the cache is full, one of the pages in the cache has to be evicted



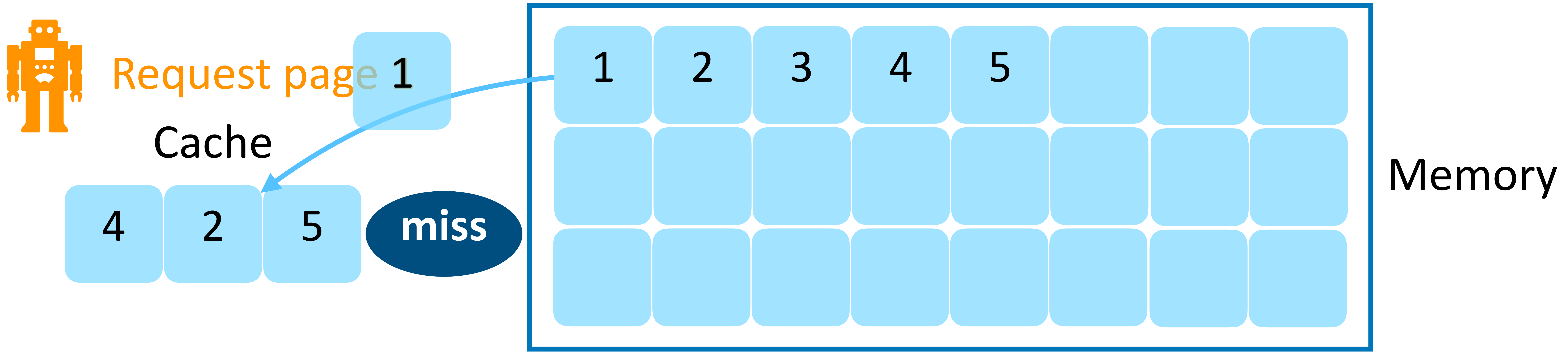
Paging

- When the processor needs a page p_i …
  - If p_i is in the cache (called a **hit**), the system needs not do anything
  - If p_i is not in the cache (a **miss**), the system incurs one **page fault** and must copy the page p_i from the slow memory to the fast cache
    - If the cache is full, one of the pages in the cache has to be evicted



Paging

- Given a sequence of n requests of pages $r_1, r_2, \cdots, r_n$, revealed one by one
  - Set of pages $= \{1, 2, 3, \cdots, n\}$
- With a size-k cache, the algorithm has to serve all the requests with a minimum number of page faults
  - Choose which page to evict wisely



Paging Algorithms

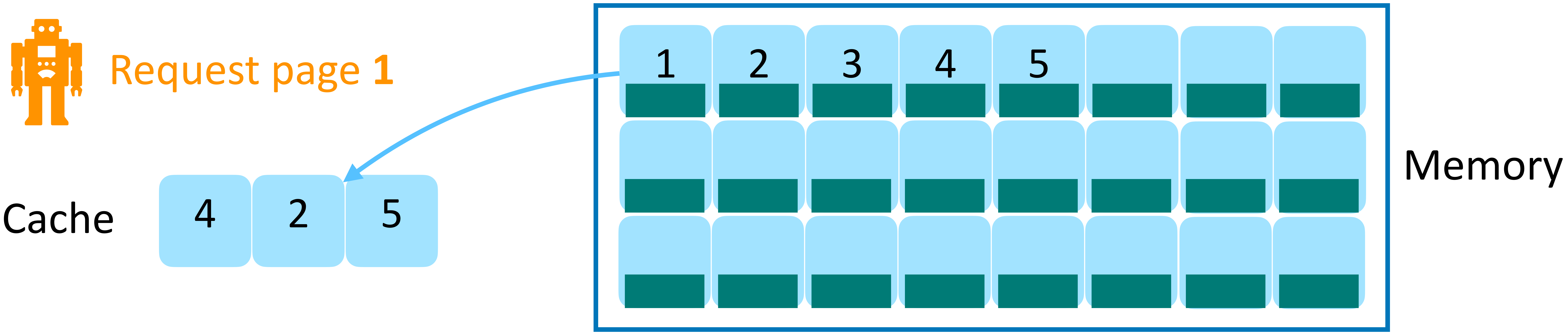
- LIFO (Last-In-First-Out)
- FIFO (First-In-First-Out)
- LFU (Least-Frequently-Used)
- LRU (Least-Recently-Used)
- CLOCK (CLOCK-replacement)
- LFD (Longest-Forward-Distance)

LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

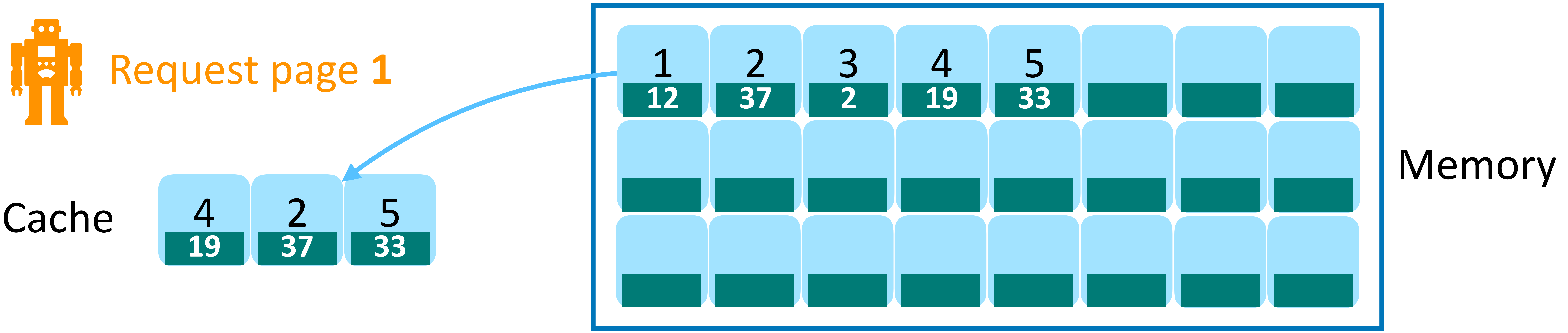


LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

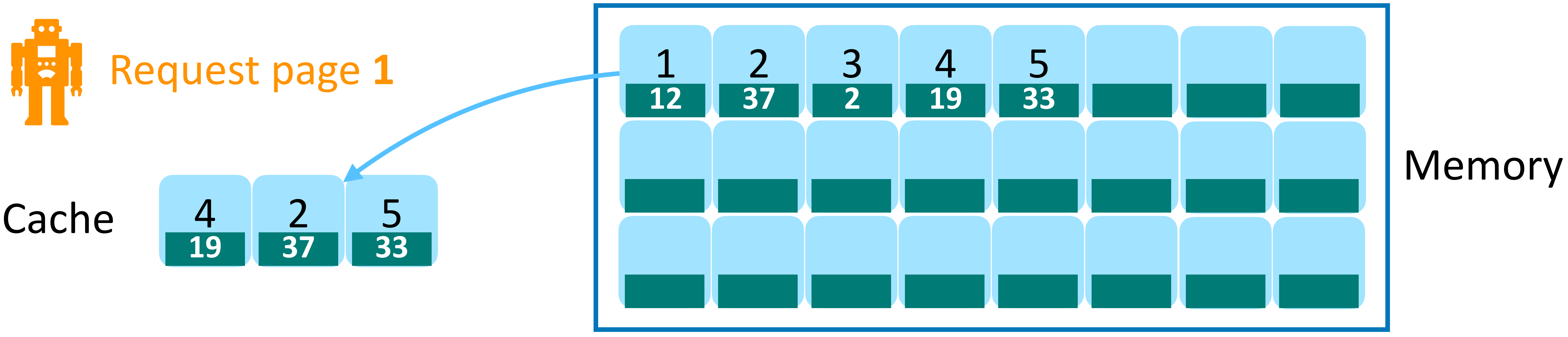


LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

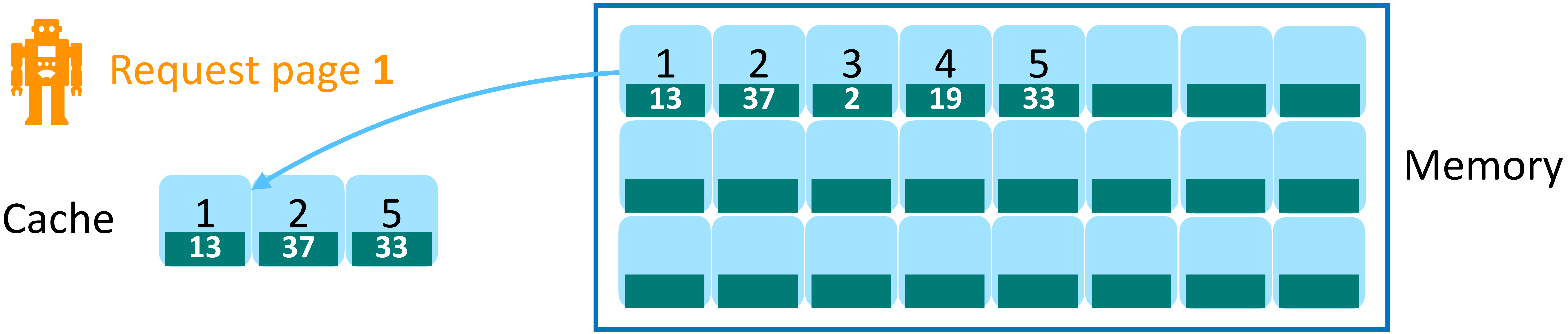


LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

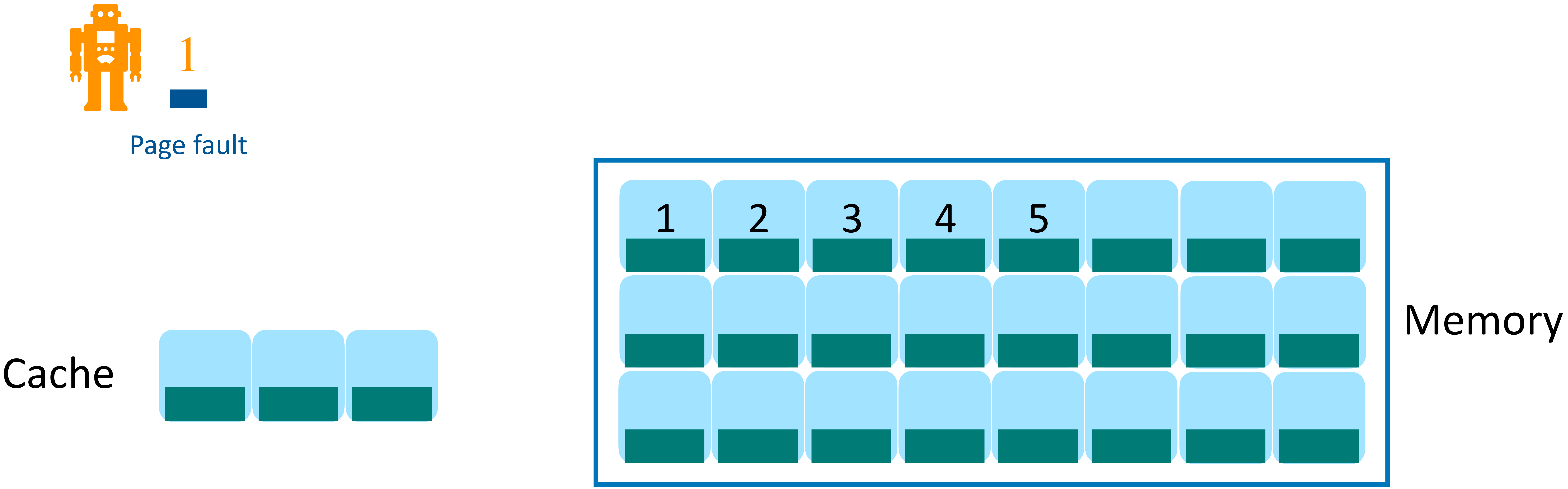


LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

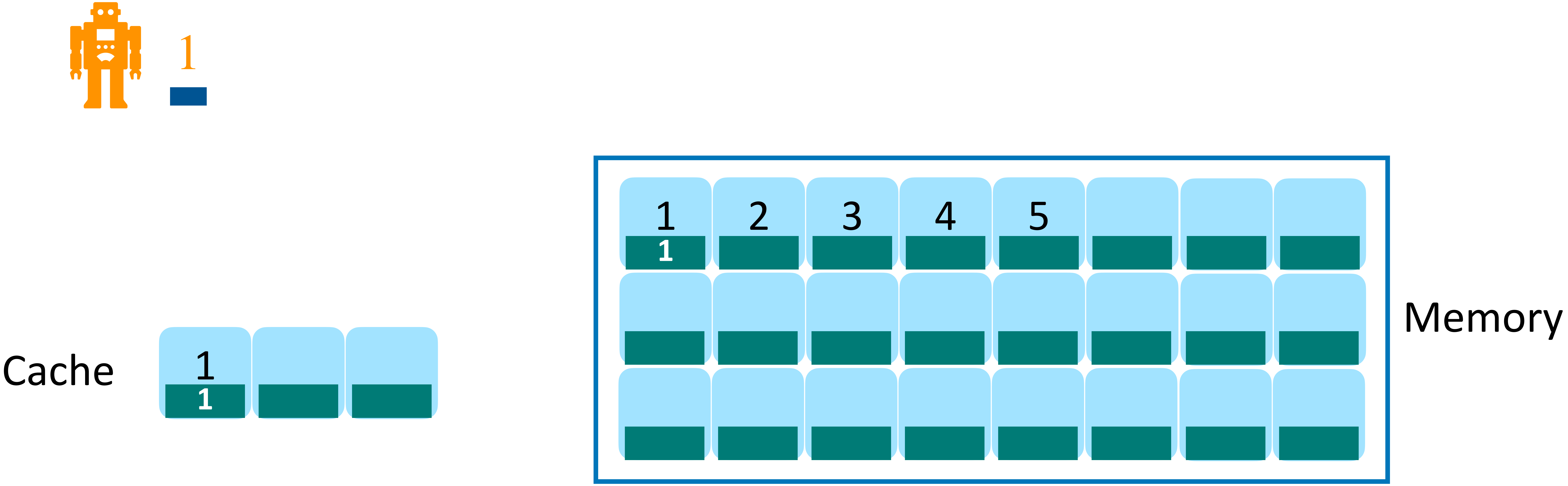


LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

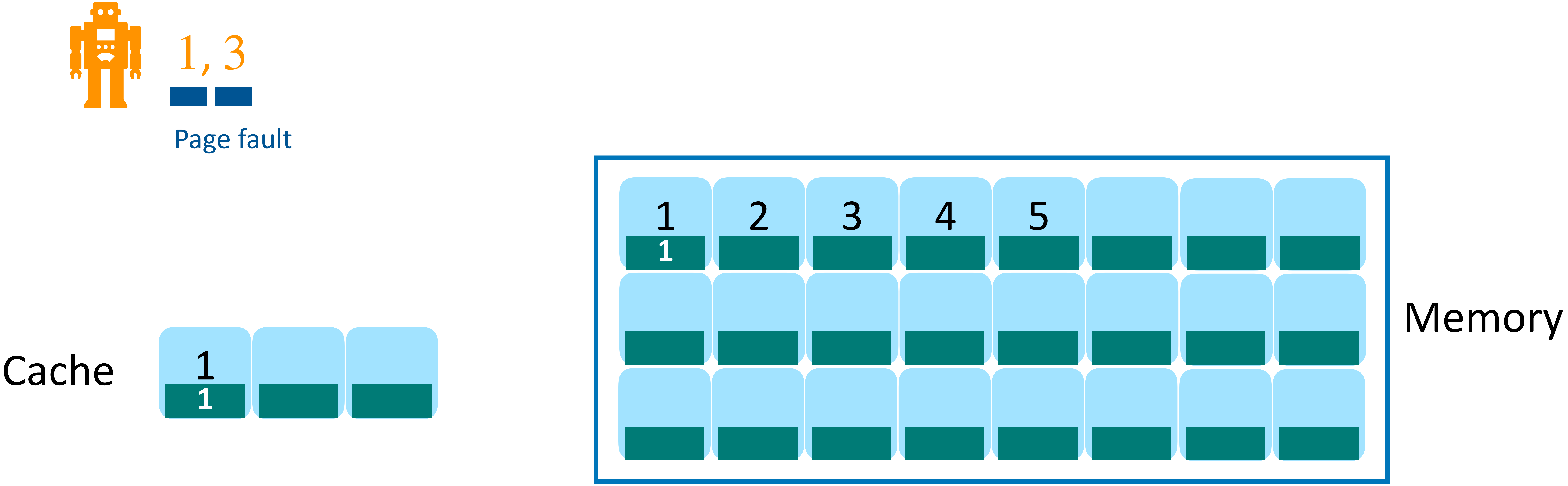


LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

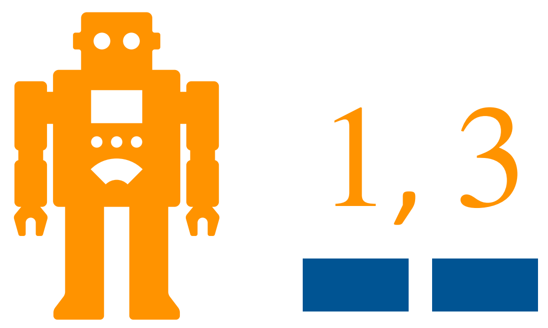


LFU (Least-Frequently-Used)

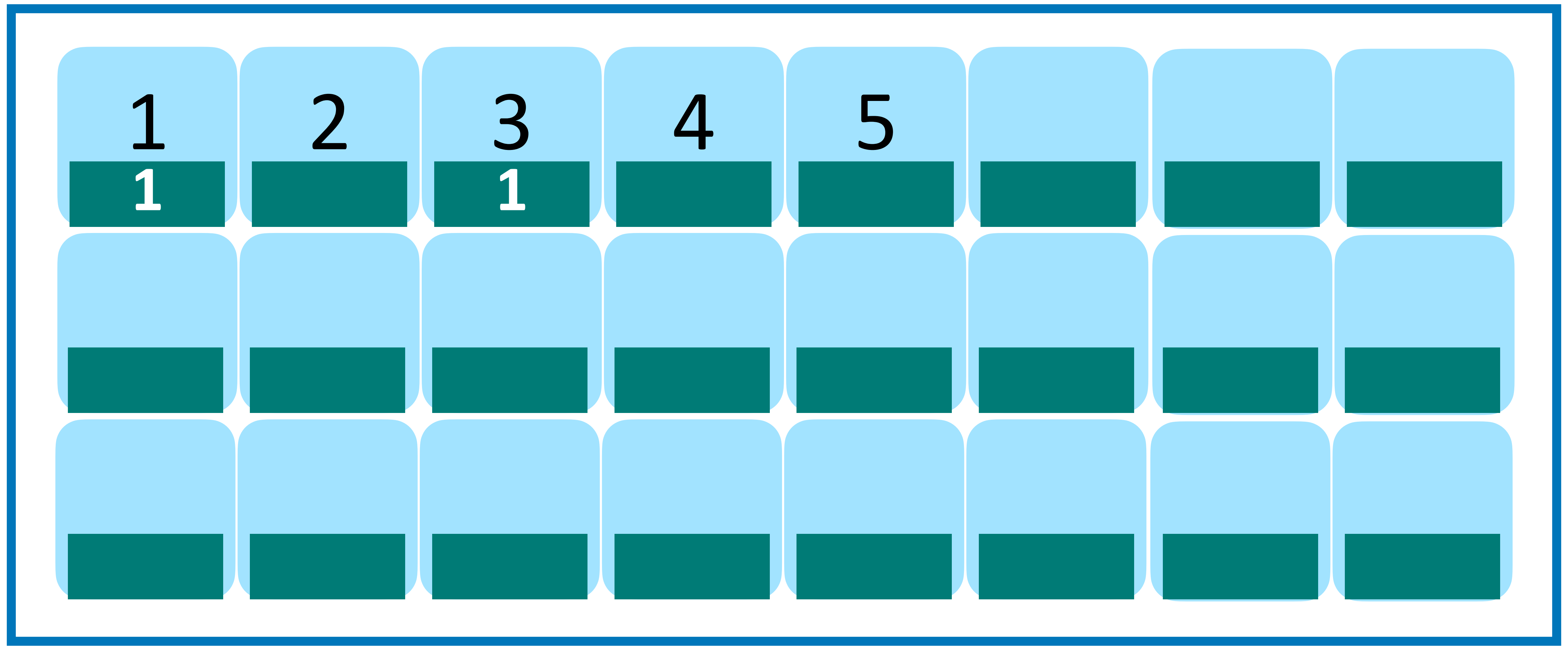
LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)



Cache

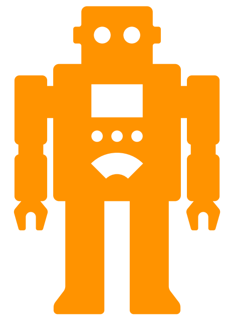


Memory

LFU (Least-Frequently-Used)

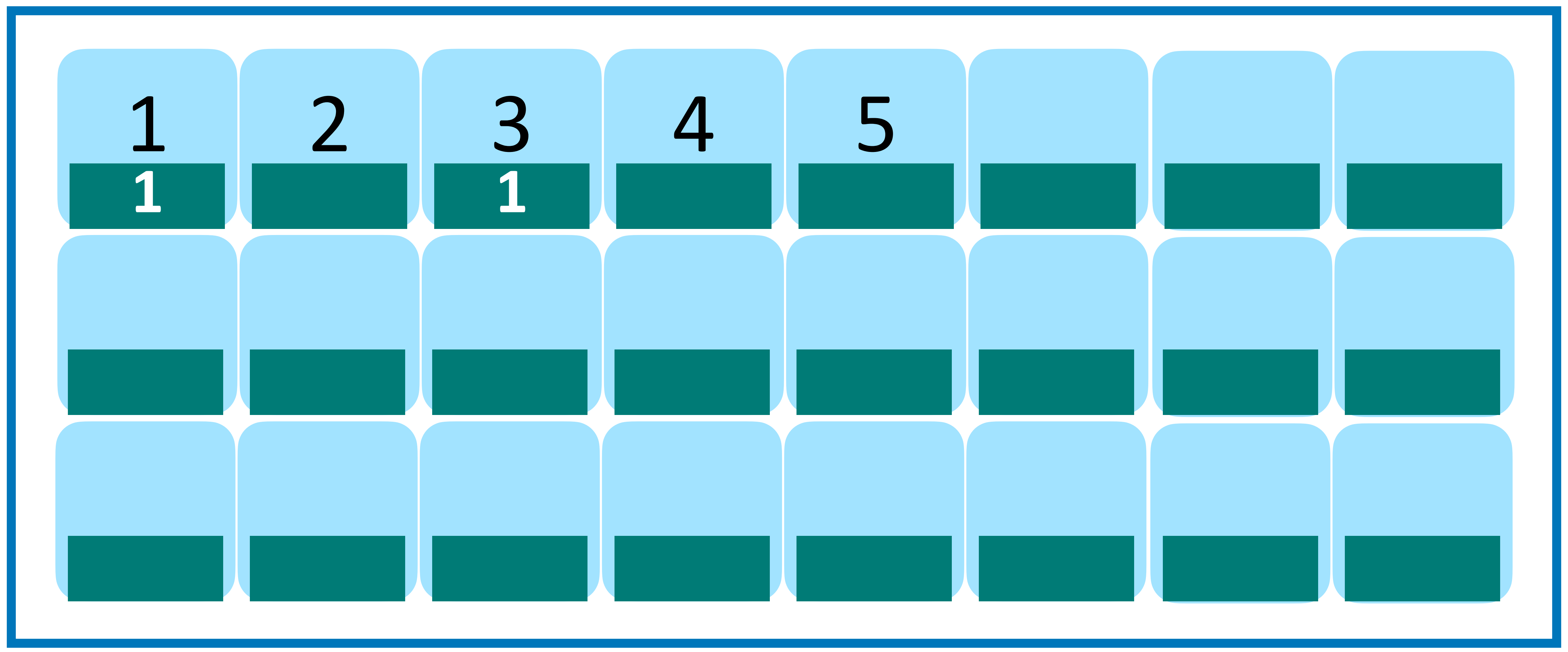
LFU (Least-Frequently-Used) algorithm:

- Every page has a **counter** that keep the number of times it has been accessed
- Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)



1, 3, 3

Cache



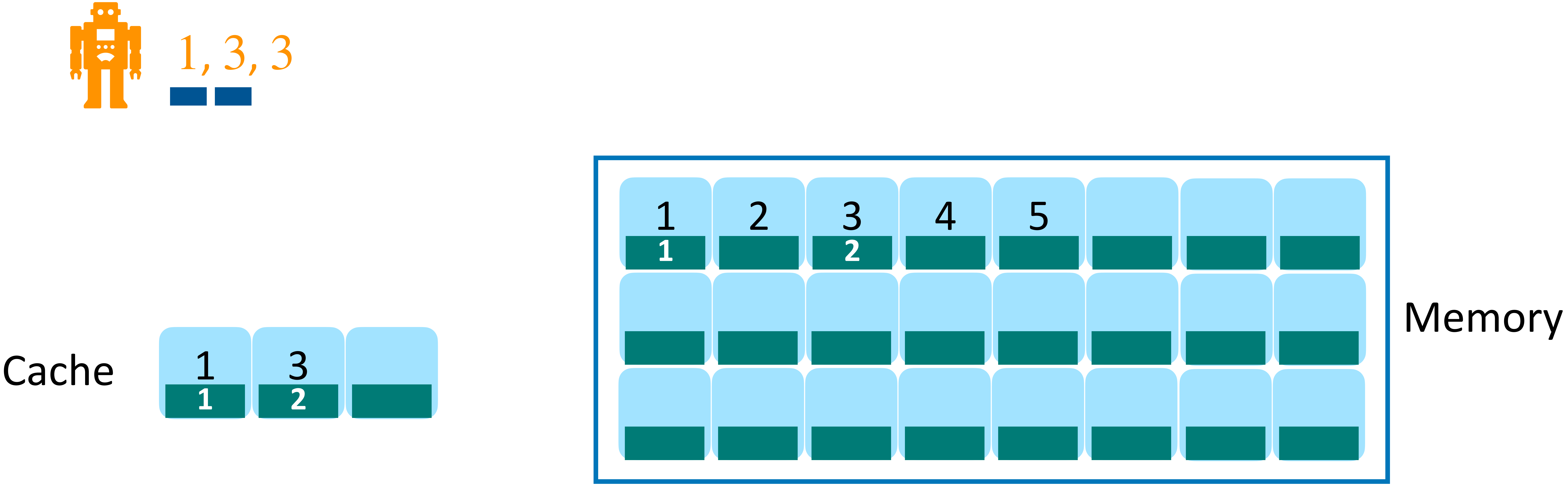
Memory

LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

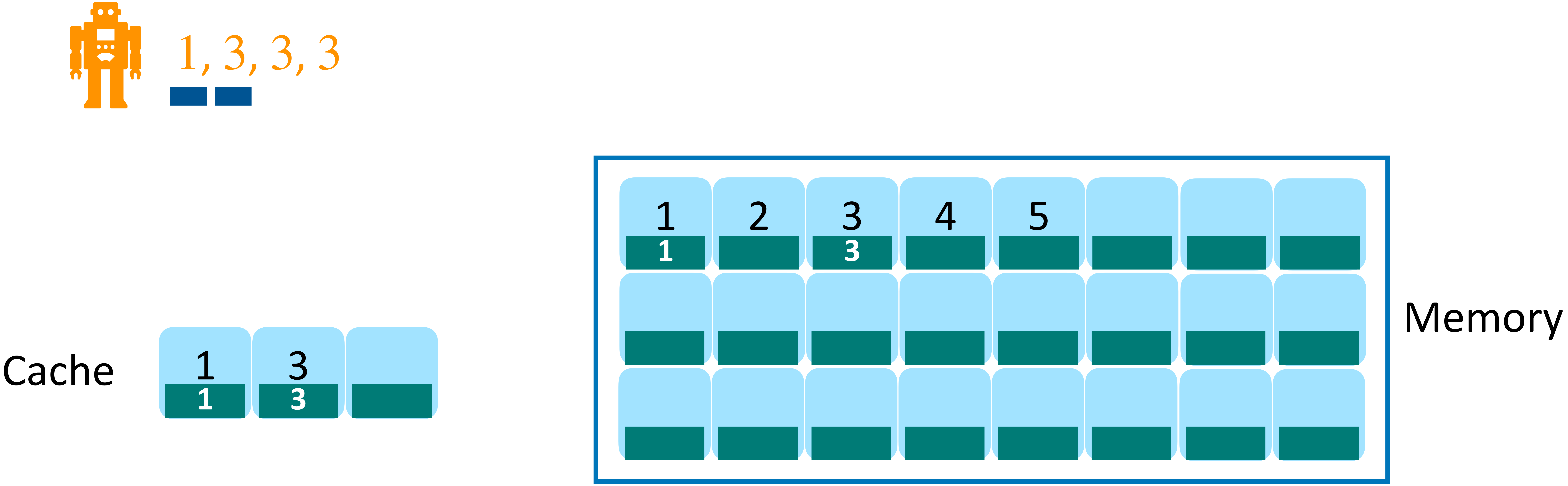


LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

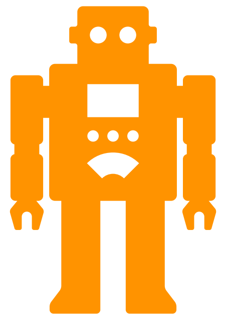


LFU (Least-Frequently-Used)

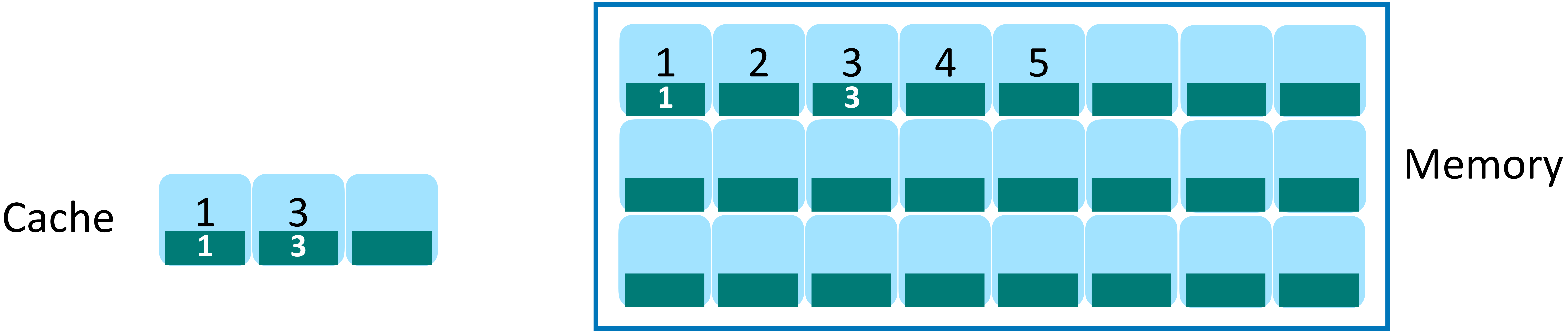
LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)



1, 3, 3, 3, 5

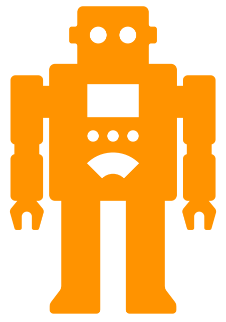


LFU (Least-Frequently-Used)

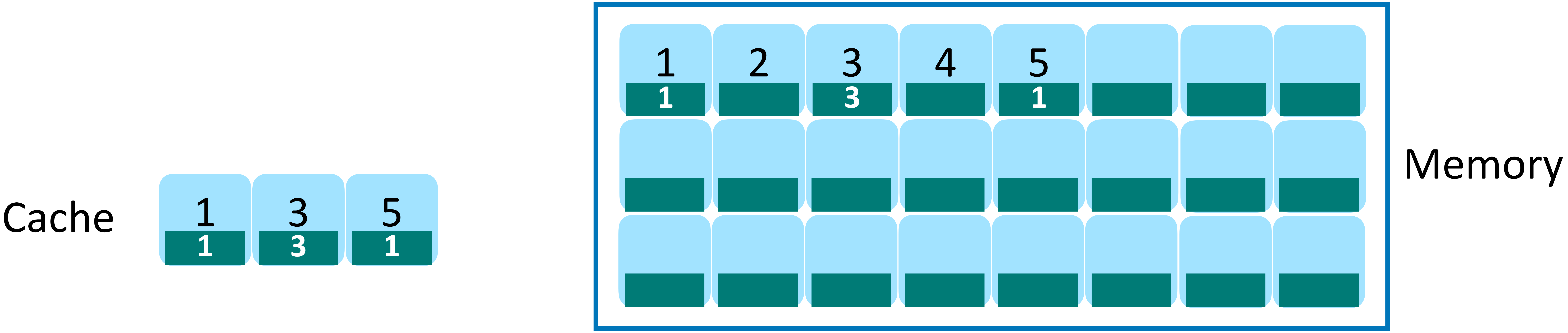
LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)



1, 3, 3, 3, 5



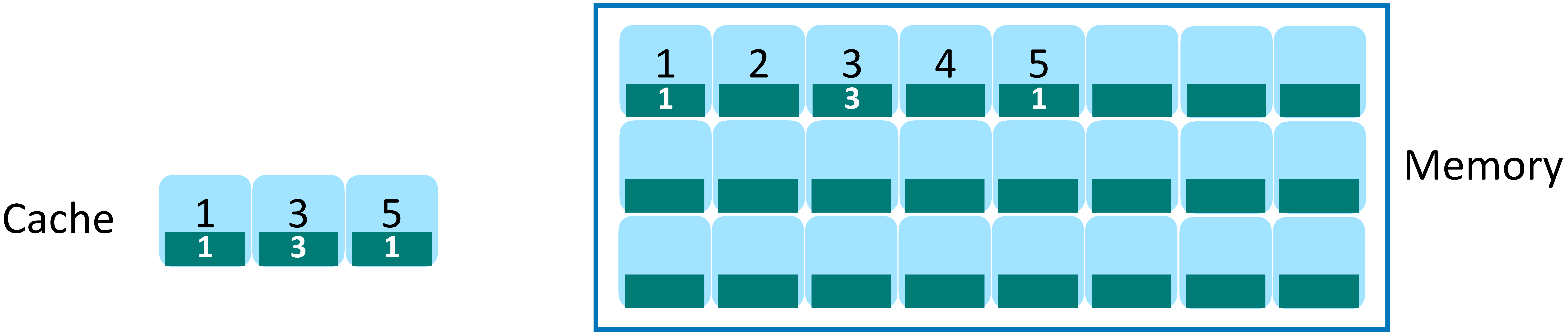
LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

1, 3, 3, 3, 5, 4

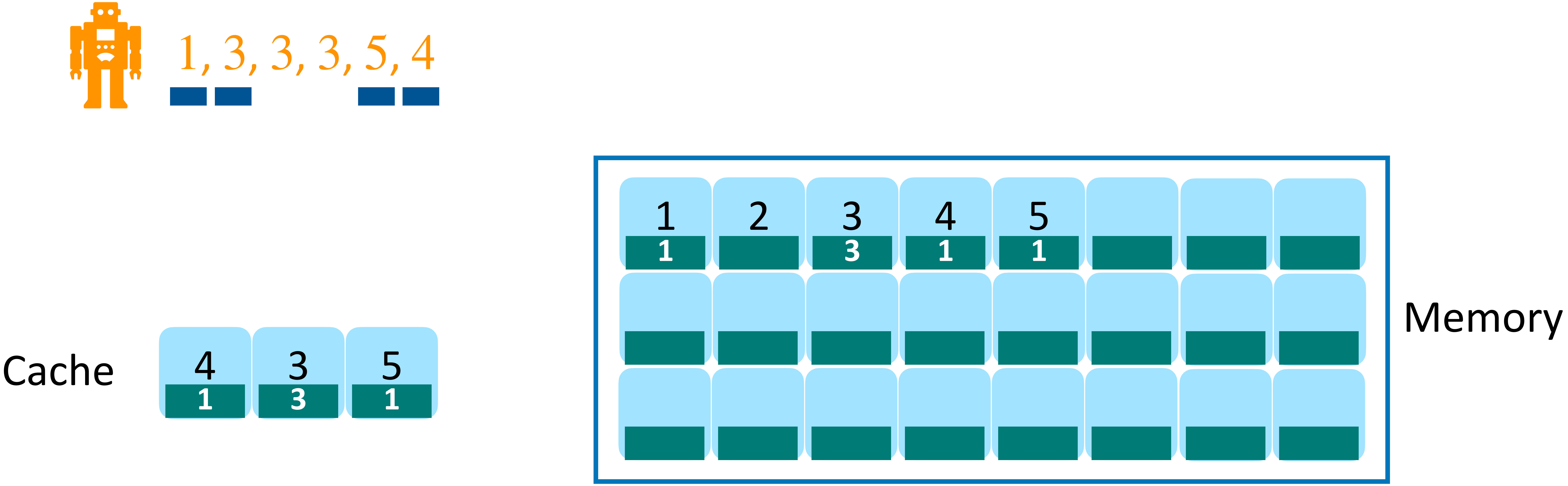


LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

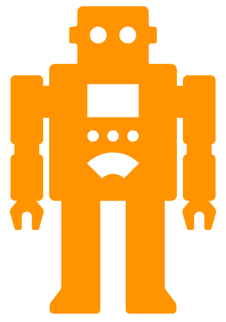


LFU (Least-Frequently-Used)

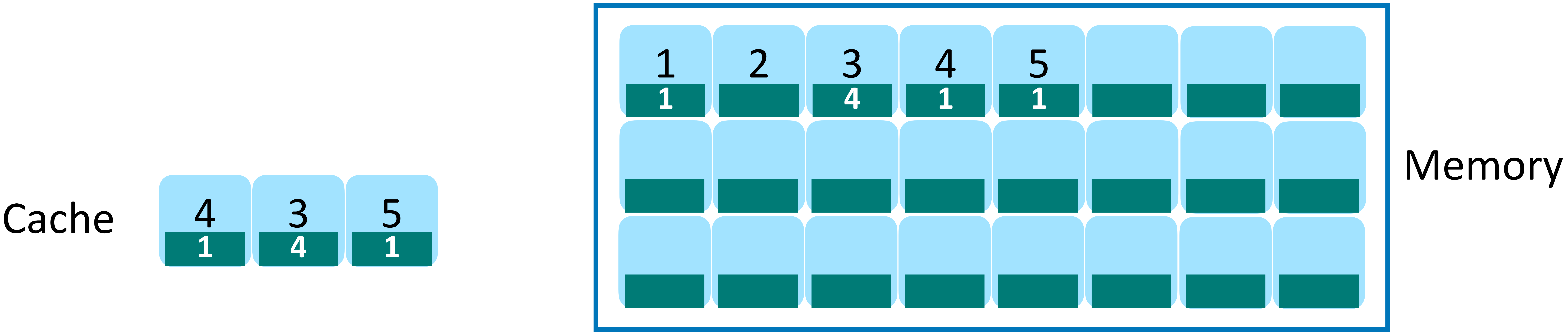
LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)



1, 3, 3, 3, 5, 4, 3



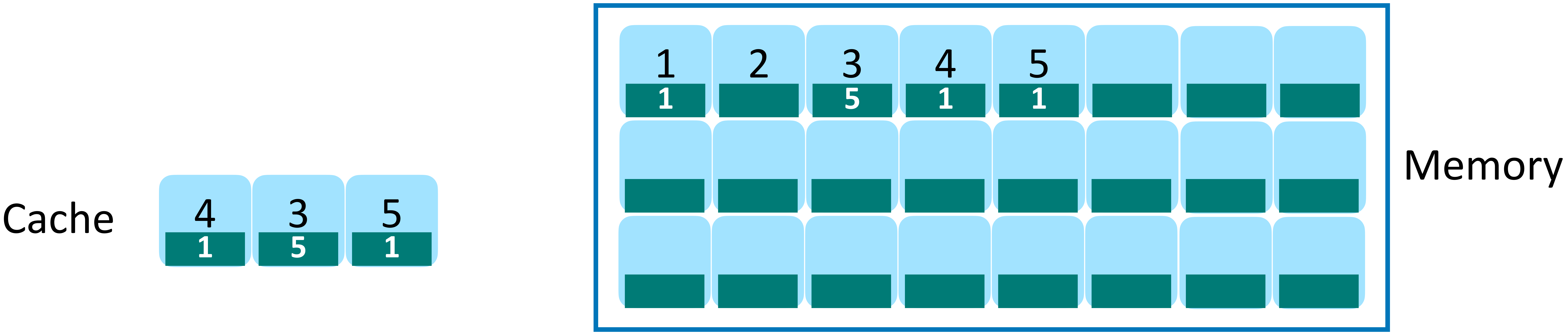
LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

1, 3, 3, 3, 5, 4, 3, 3



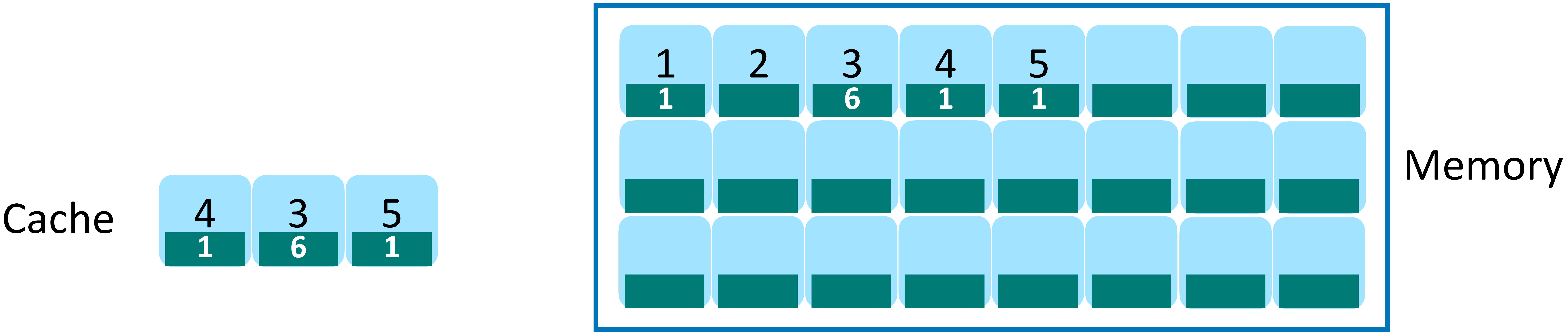
LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

1, 3, 3, 3, 5, 4, 3, 3, 3

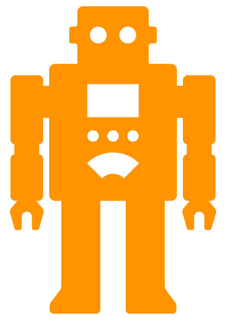


LFU (Least-Frequently-Used)

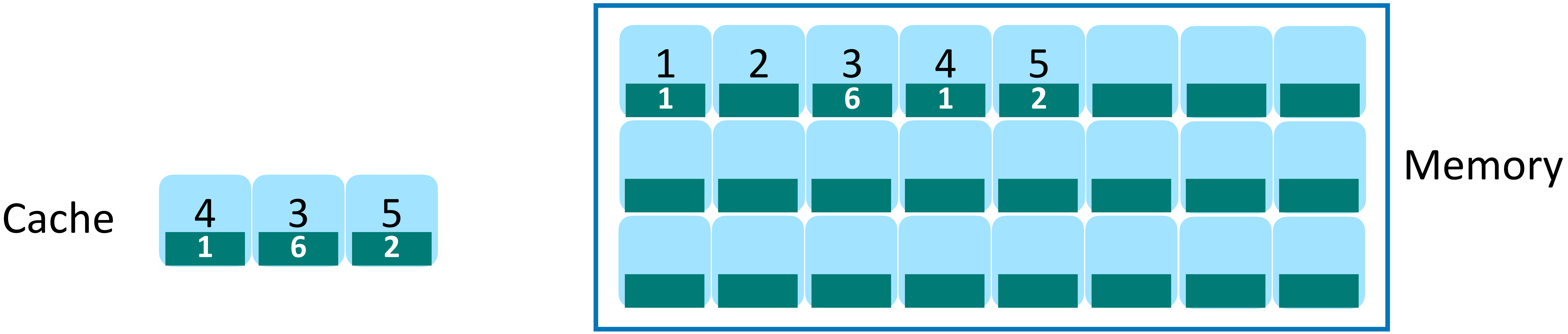
LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)



1, 3, 3, 3, 5, 4, 3, 3, 3, 5



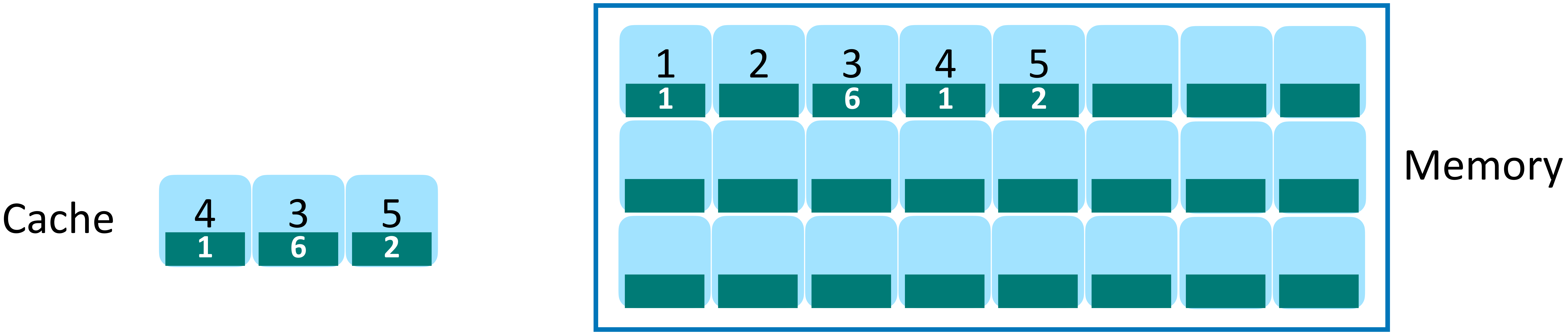
LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

1, 3, 3, 3, 5, 4, 3, 3, 3, 5, 2



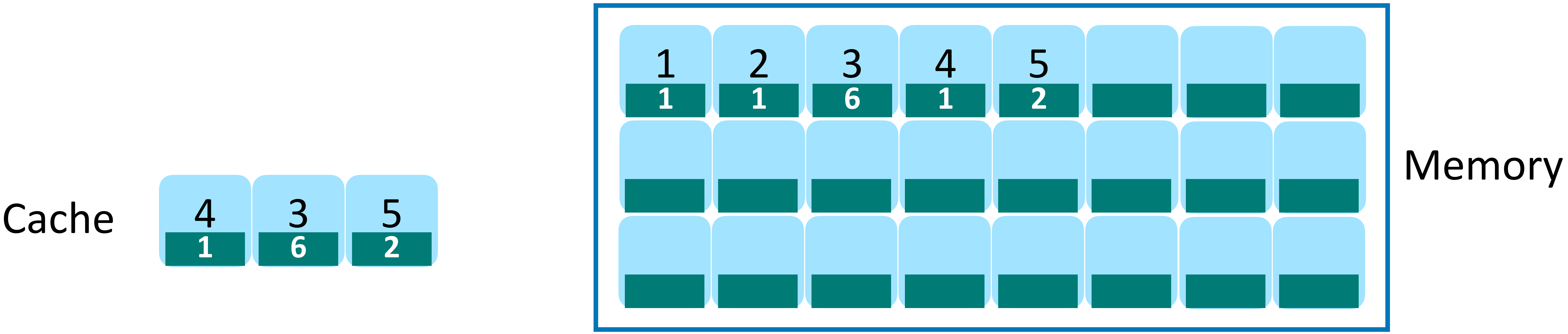
LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

1, 3, 3, 3, 5, 4, 3, 3, 3, 5, 2



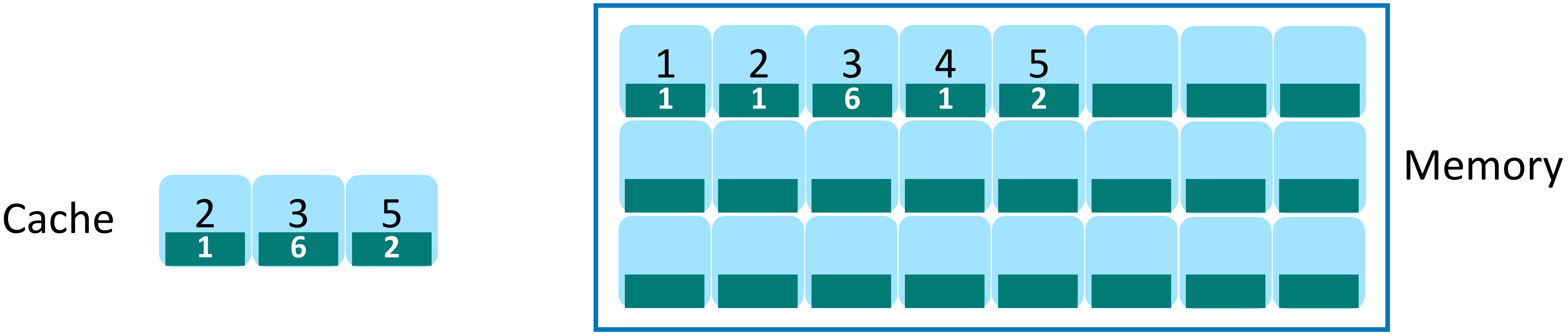
LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

1, 3, 3, 3, 5, 4, 3, 3, 3, 5, 2



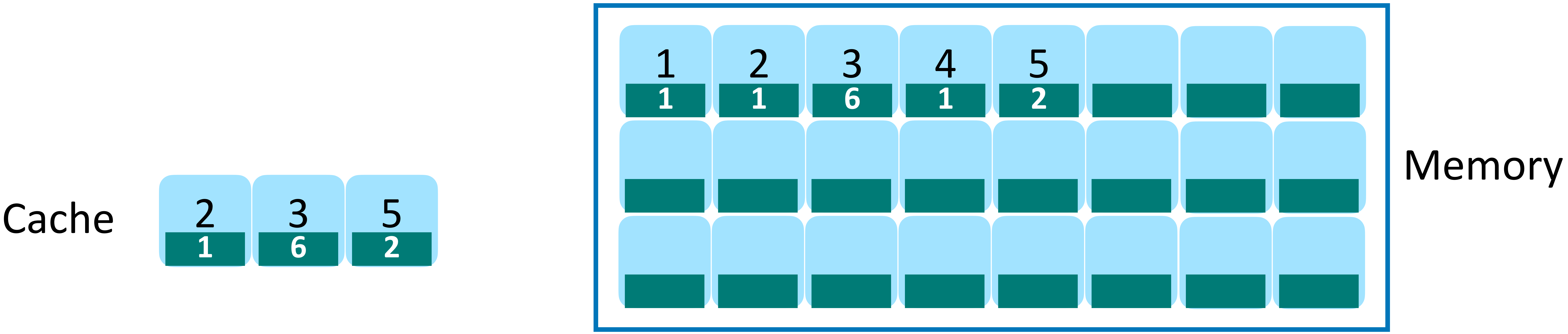
LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

1, 3, 3, 3, 5, 4, 3, 3, 3, 5, 2, 1



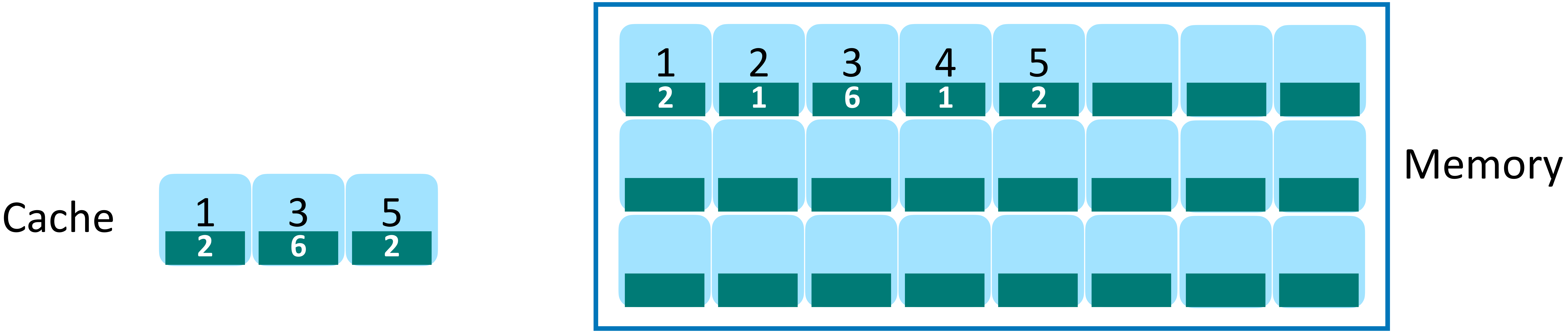
LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

1, 3, 3, 3, 5, 4, 3, 3, 3, 5, 2, 1

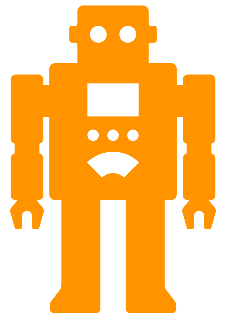


LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

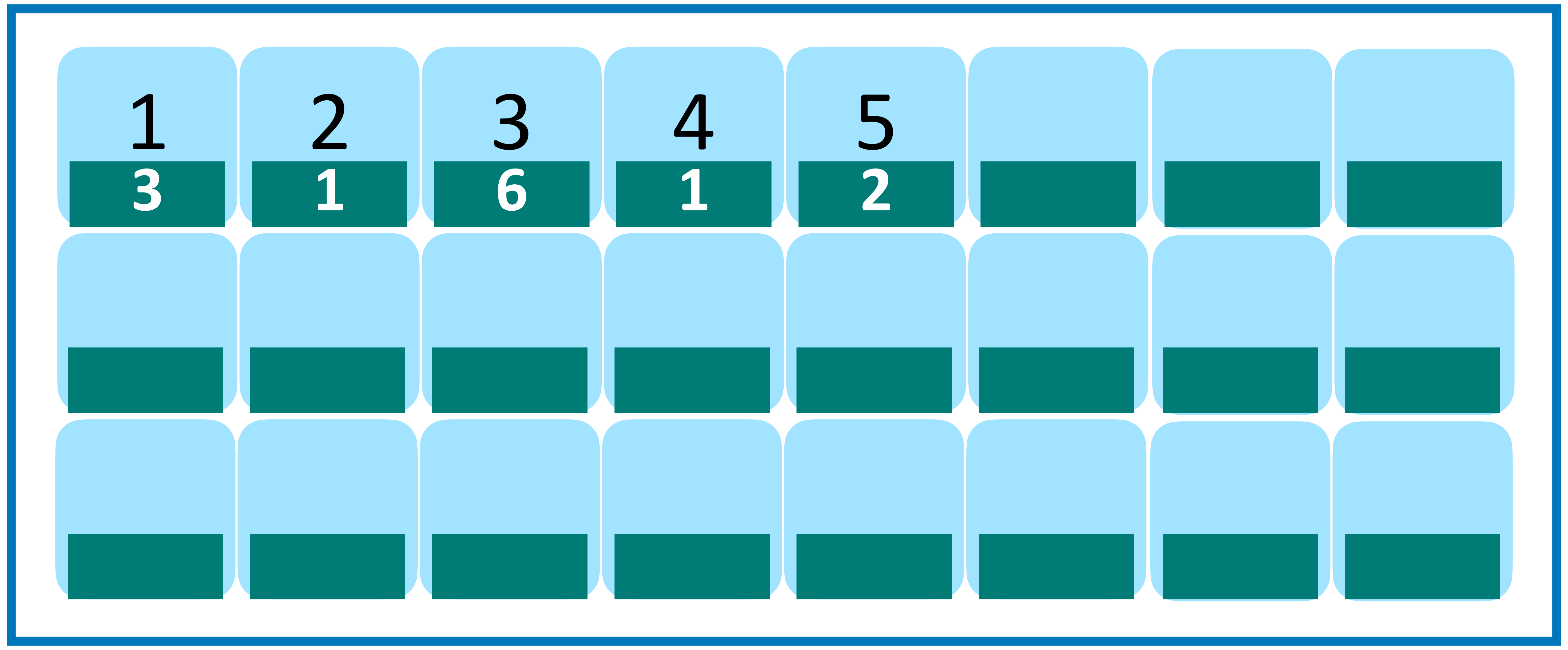
Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)



1, 3, 3, 3, 5, 4, 3, 3, 3, 5, 2, 1, 1

Cache



Memory

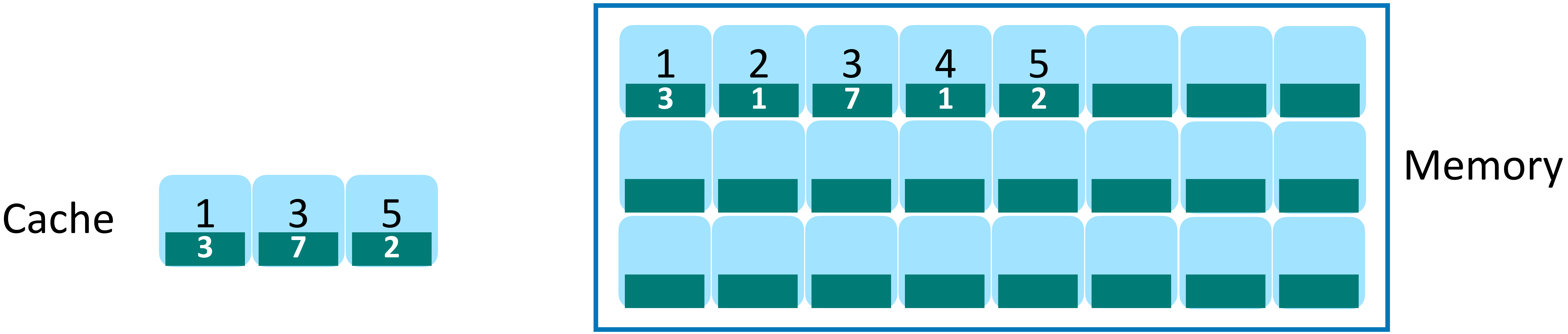
LFU (Least-Frequently-Used)

LFU (Least-Frequently-Used) algorithm:

Every page has a **counter** that keep the number of times it has been accessed

Once a page fault is incurred, evict the one with the lowest counter value (break tie arbitrarily)

1, 3, 3, 3, 5, 4, 3, 3, 3, 5, 2, 1, 1, 3



LFU competitive ratio is unbounded

LFU competitive ratio is unbounded

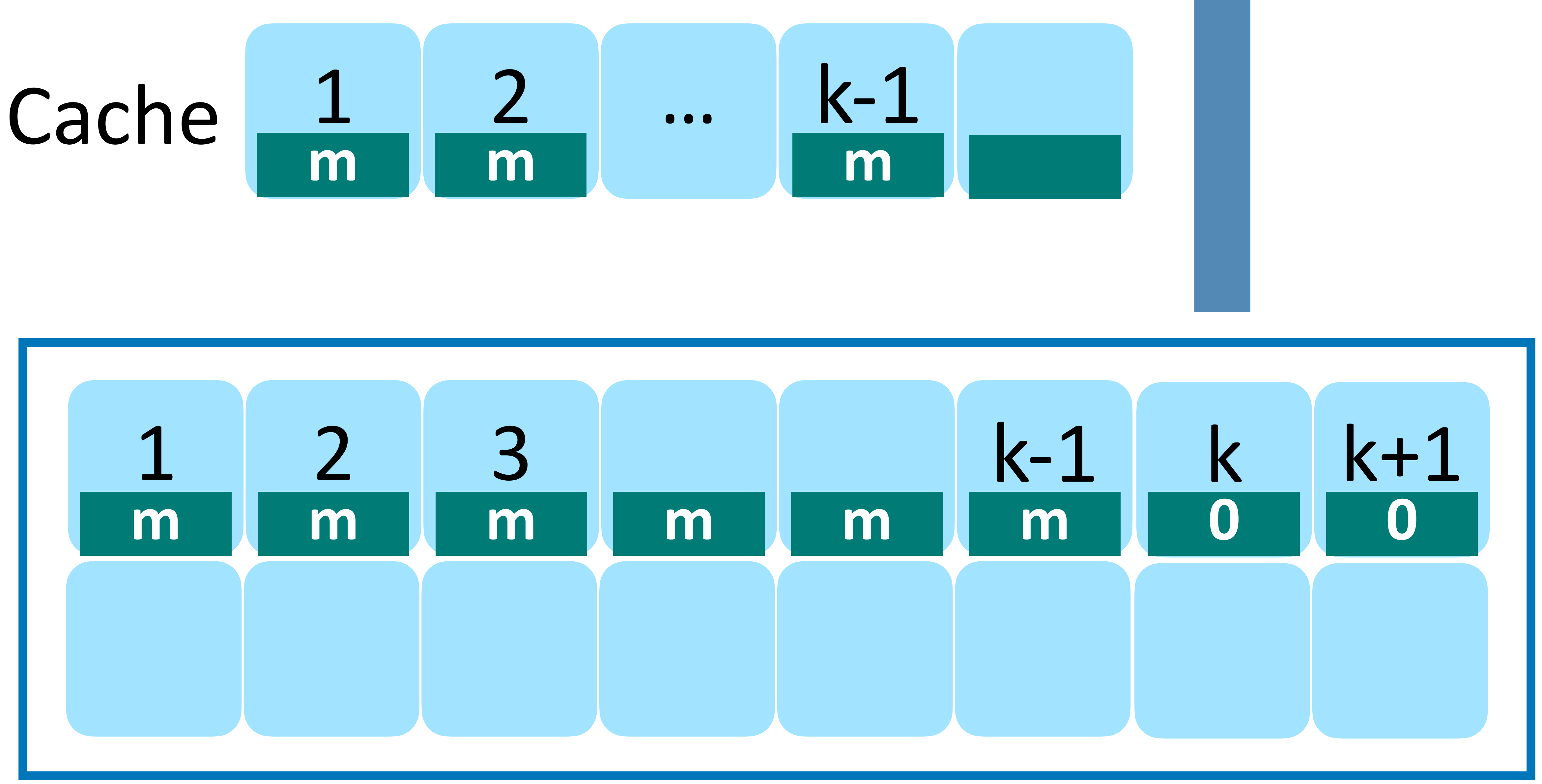
- Consider the sequence of requests:

$$\underbrace{1, 1, \cdots, 1}_{m \text{ x } 1\text{'s}}, \underbrace{2, 2, \cdots, 2}_{m \text{ x } 2\text{'s}}, \underbrace{3, 3, \cdots, 3}_{m \text{ x } 3\text{'s}}, \cdots, \underbrace{k-1, k-1, \cdots, k-1}_{m \text{ x } k-1\text{'s}}, \underbrace{k, k+1, \text{k}, k+1, \cdots, k, k+1}_{m \text{ x } (k, k+1)\text{'s}}$$

LFU competitive ratio is unbounded

- Consider the sequence of requests:

$$\underbrace{1, 1, \cdots, 1}_{m \text{ x } 1\text{'s}}, \underbrace{2, 2, \cdots, 2}_{m \text{ x } 2\text{'s}}, \underbrace{3, 3, \cdots, 3}_{m \text{ x } 3\text{'s}}, \cdots, \underbrace{k-1, k-1, \cdots, k-1}_{m \text{ x } k-1\text{'s}}, \underbrace{k, k+1, \mathrm{k}, k+1, \cdots, k, k+1}_{m \text{ x } (k, k+1)\text{'s}}$$

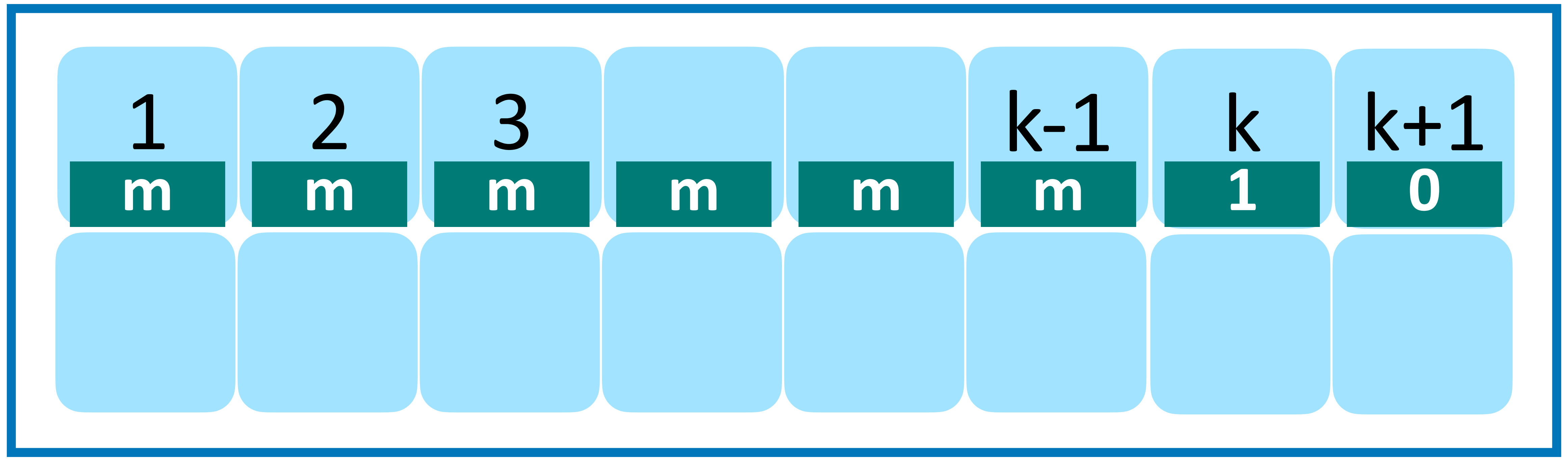


LFU competitive ratio is unbounded

- Consider the sequence of requests:

$$\underbrace{1, 1, \cdots, 1}_{m \text{ x } 1\text{'s}}, \underbrace{2, 2, \cdots, 2}_{m \text{ x } 2\text{'s}}, \underbrace{3, 3, \cdots, 3}_{m \text{ x } 3\text{'s}}, \cdots, \underbrace{k-1, k-1, \cdots, k-1}_{m \text{ x } k-1\text{'s}}, \underbrace{k, k+1, \text{k}, k+1, \cdots, k, k+1}_{m \text{ x } (k,\, k+1)\text{'s}}$$

Cache

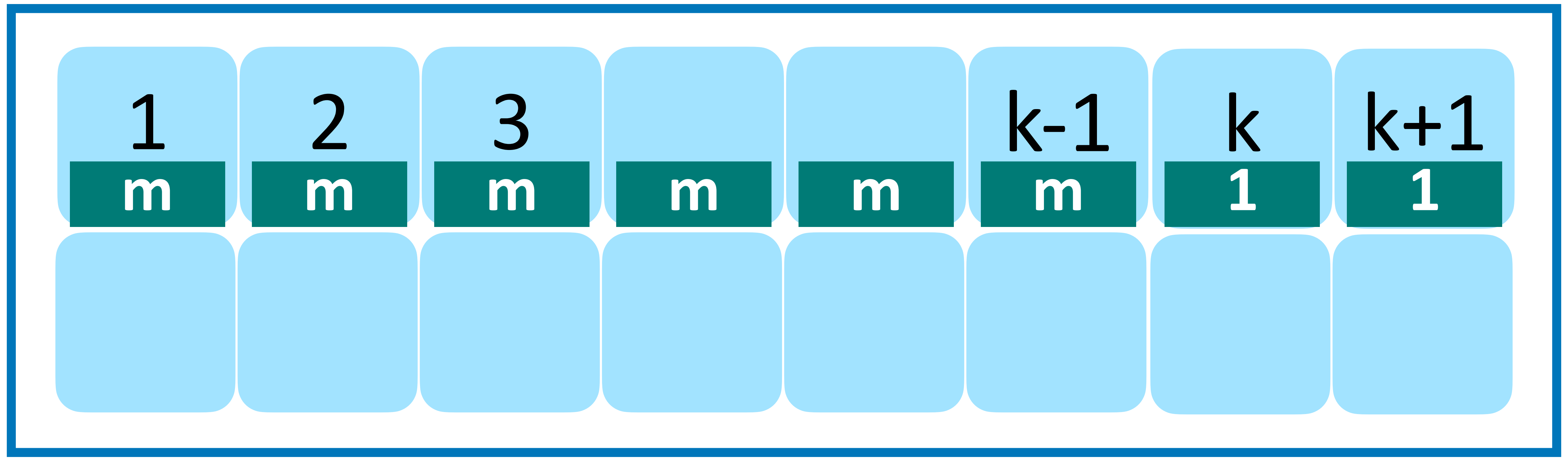


LFU competitive ratio is unbounded

- Consider the sequence of requests:

$$\underbrace{1, 1, \cdots, 1}_{m \text{ x } 1\text{'s}}, \underbrace{2, 2, \cdots, 2}_{m \text{ x } 2\text{'s}}, \underbrace{3, 3, \cdots, 3}_{m \text{ x } 3\text{'s}}, \cdots, \underbrace{k-1, k-1, \cdots, k-1}_{m \text{ x } k-1\text{'s}}, \underbrace{k, k+1, \mathrm{k}, k+1, \cdots, k, k+1}_{m \text{ x } (k,\, k+1)\text{'s}}$$

Cache

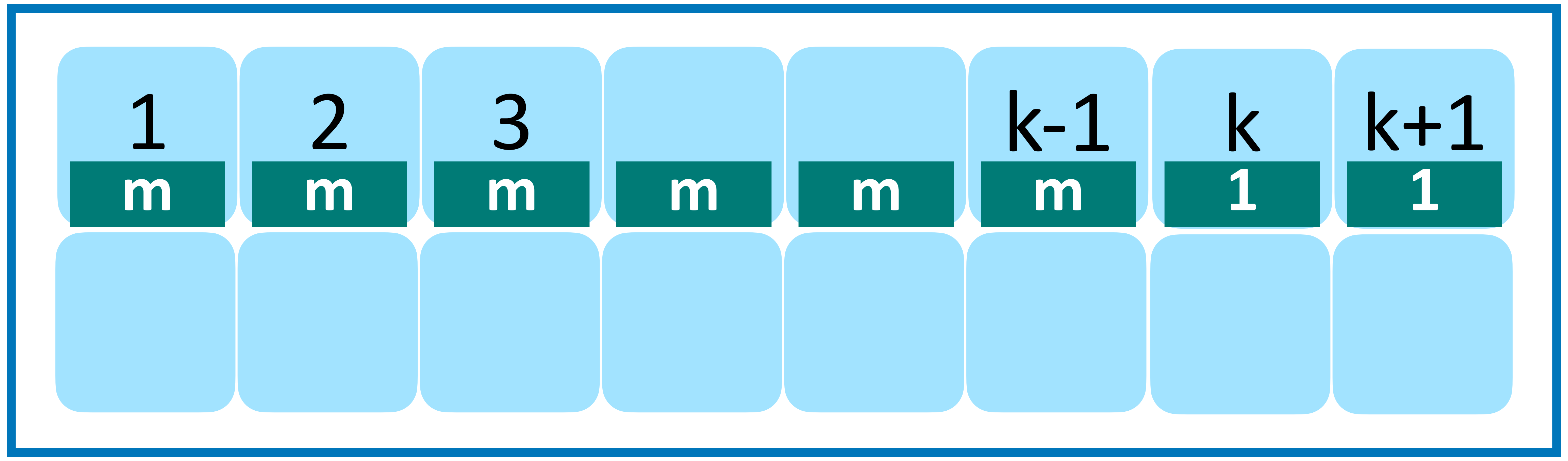


LFU competitive ratio is unbounded

- Consider the sequence of requests:

$$\underbrace{1, 1, \cdots, 1}_{m \text{ x } 1\text{'s}}, \underbrace{2, 2, \cdots, 2}_{m \text{ x } 2\text{'s}}, \underbrace{3, 3, \cdots, 3}_{m \text{ x } 3\text{'s}}, \cdots, \underbrace{k-1, k-1, \cdots, k-1}_{m \text{ x } k-1\text{'s}}, \underbrace{k, k+1, \text{k}, k+1, \cdots, k, k+1}_{m \text{ x } (k, k+1)\text{'s}}$$

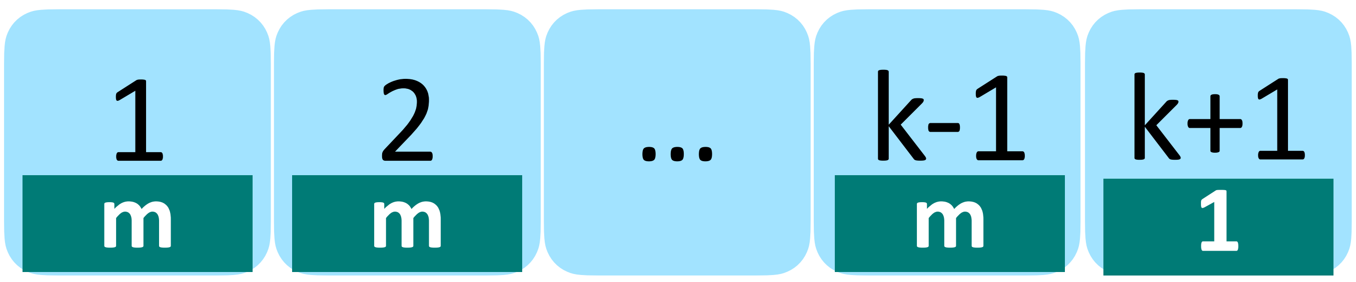
Cache

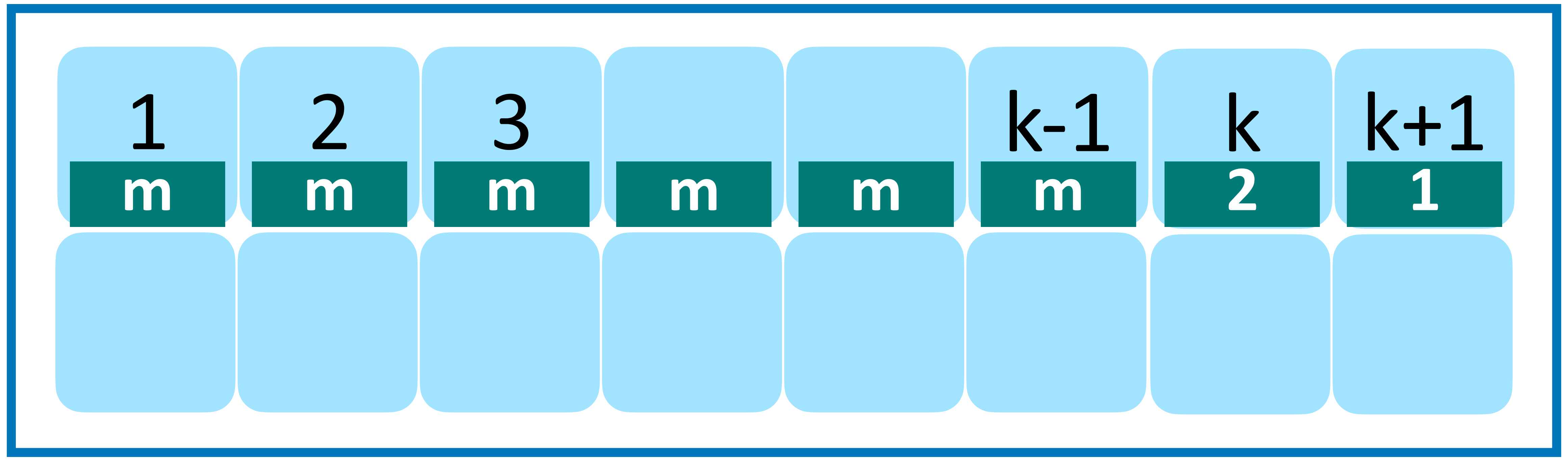


LFU competitive ratio is unbounded

- Consider the sequence of requests:

$$\underbrace{1, 1, \cdots, 1}_{m \text{ x } 1\text{'s}}, \underbrace{2, 2, \cdots, 2}_{m \text{ x } 2\text{'s}}, \underbrace{3, 3, \cdots, 3}_{m \text{ x } 3\text{'s}}, \cdots, \underbrace{k-1, k-1, \cdots, k-1}_{m \text{ x } k-1\text{'s}}, \underbrace{k, k+1, k, k+1, \cdots, k, k+1}_{m \text{ x } (k, k+1)\text{'s}}$$

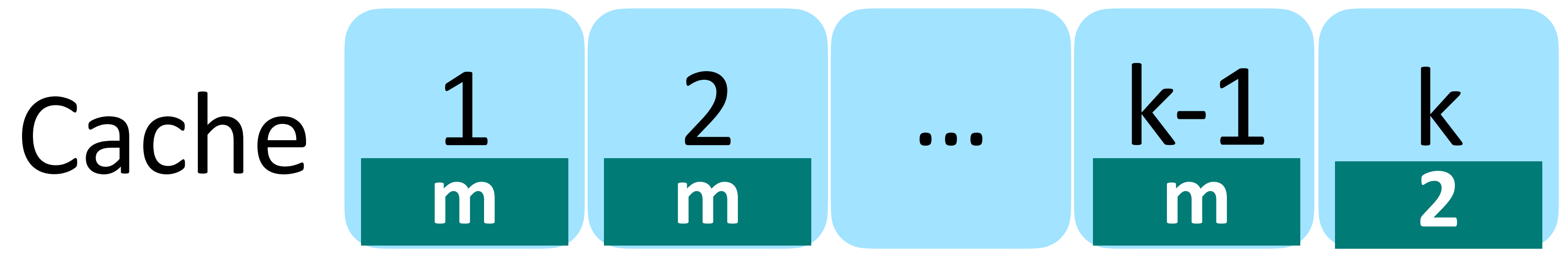
Cache 

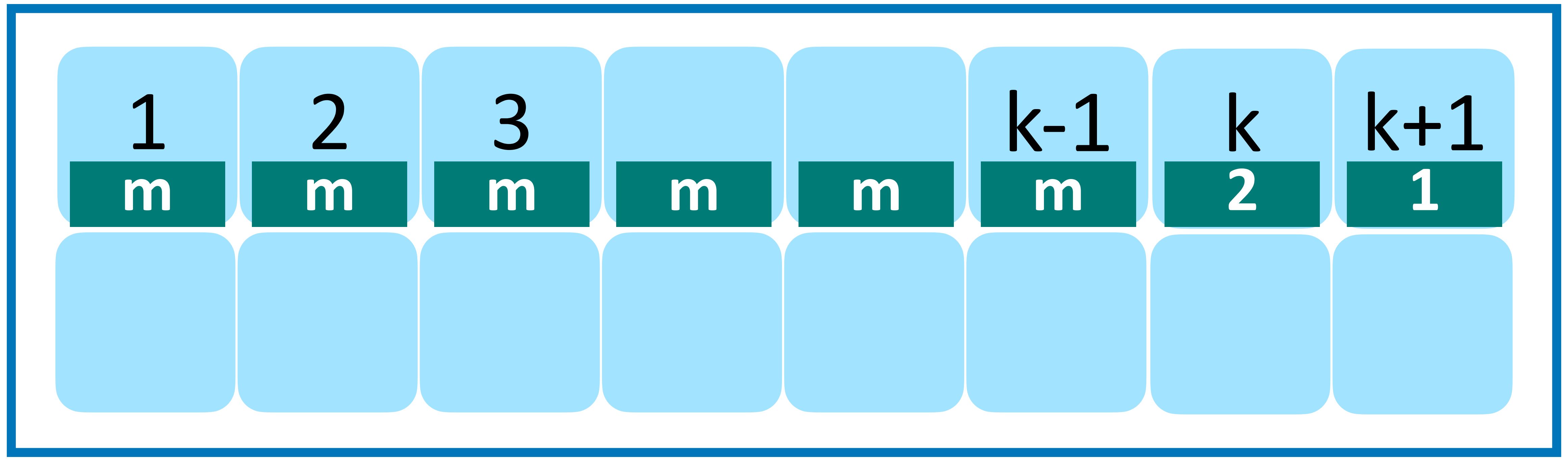


LFU competitive ratio is unbounded

- Consider the sequence of requests:

$$\underbrace{1, 1, \cdots, 1}_{m \text{ x } 1\text{'s}}, \underbrace{2, 2, \cdots, 2}_{m \text{ x } 2\text{'s}}, \underbrace{3, 3, \cdots, 3}_{m \text{ x } 3\text{'s}}, \cdots, \underbrace{k-1, k-1, \cdots, k-1}_{m \text{ x } k-1\text{'s}}, \underbrace{k, k+1, \mathrm{k}, k+1, \cdots, k, k+1}_{m \text{ x } (k, k+1)\text{'s}}$$

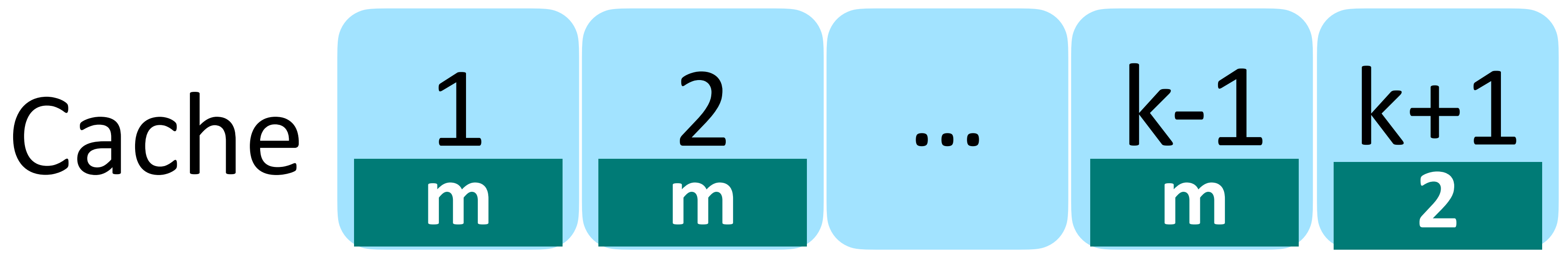


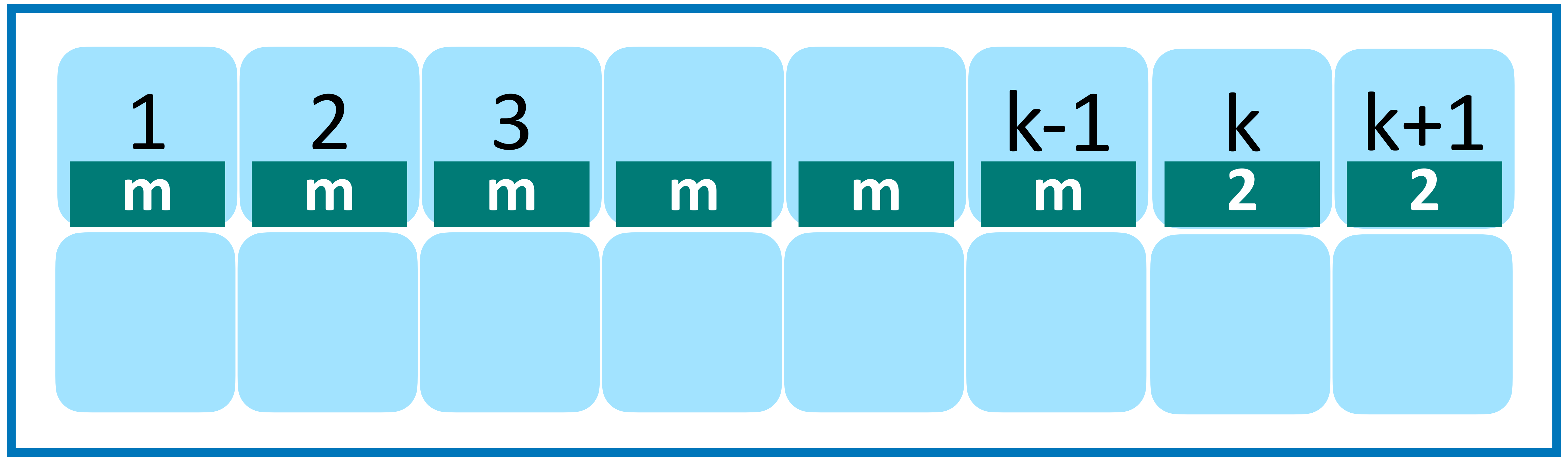


LFU competitive ratio is unbounded

- Consider the sequence of requests:

$$\underbrace{1,1,\cdots,1}_{m \text{ x } 1\text{'s}},\underbrace{2,2,\cdots,2}_{m \text{ x } 2\text{'s}},\underbrace{3,3,\cdots,3}_{m \text{ x } 3\text{'s}},\cdots,\underbrace{k-1,k-1,\cdots,k-1}_{m \text{ x } k-1\text{'s}},\underbrace{k,k+1,\text{k},k+1,\cdots,k,k+1}_{m \text{ x } (k,k+1)\text{'s}}$$





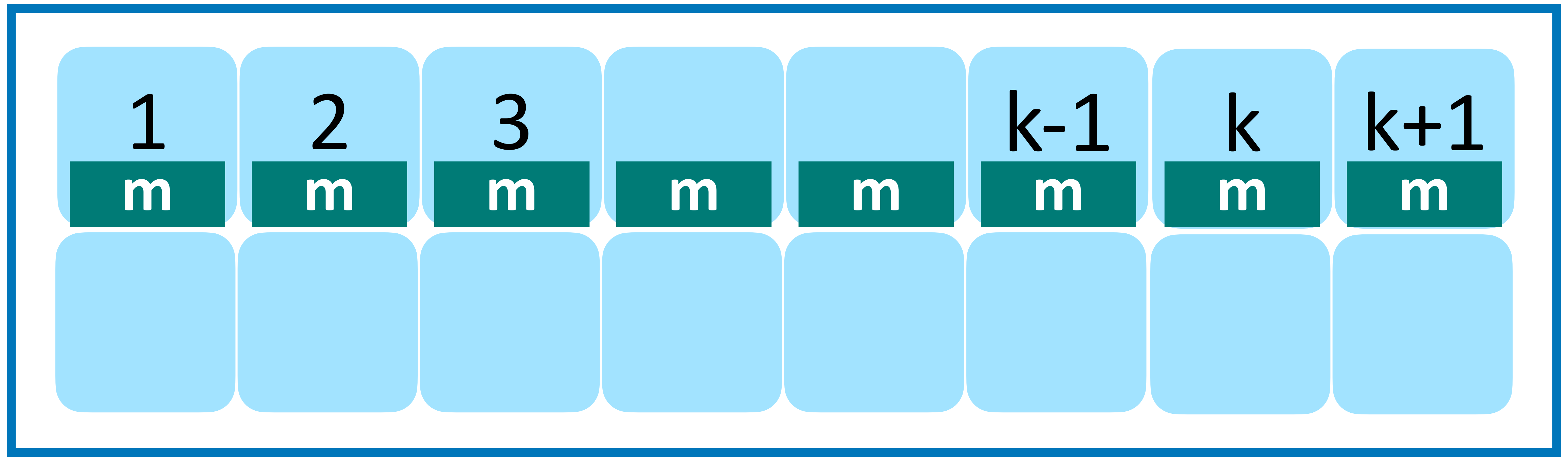
LFU competitive ratio is unbounded

- Consider the sequence of requests:

$$\underbrace{1, 1, \cdots, 1}_{m \text{ x } 1\text{'s}}, \underbrace{2, 2, \cdots, 2}_{m \text{ x } 2\text{'s}}, \underbrace{3, 3, \cdots, 3}_{m \text{ x } 3\text{'s}}, \cdots, \underbrace{k-1, k-1, \cdots, k-1}_{m \text{ x } k-1\text{'s}}, \underbrace{k, k+1, \text{k}, k+1, \cdots, k, k+1}_{m \text{ x } (k, k+1)\text{'s}}$$

Cache

LFU $= (k-1) + 2m$



LFU competitive ratio is unbounded

- Consider the sequence of requests:

$$\underbrace{1, 1, \cdots, 1}_{m \text{ x } 1\text{'s}}, \underbrace{2, 2, \cdots, 2}_{m \text{ x } 2\text{'s}}, \underbrace{3, 3, \cdots, 3}_{m \text{ x } 3\text{'s}}, \cdots, \underbrace{k-1, k-1, \cdots, k-1}_{m \text{ x } k-1\text{'s}}, \underbrace{k, k+1, \text{k}, k+1, \cdots, k, k+1}_{m \text{ x } (k,\, k+1)\text{'s}}$$

Cache	1 m	2 m	…	k-1 m	k+1 m
OPT cache	1	2	…	k-1	k

$\textbf{LFU} = (k-1) + 2m$

LFU competitive ratio is unbounded

- Consider the sequence of requests:

$$\underbrace{1, 1, \cdots, 1}_{m \text{ x } 1\text{'s}}, \underbrace{2, 2, \cdots, 2}_{m \text{ x } 2\text{'s}}, \underbrace{3, 3, \cdots, 3}_{m \text{ x } 3\text{'s}}, \cdots, \underbrace{k-1, k-1, \cdots, k-1}_{m \text{ x } k-1\text{'s}}, \underbrace{k, k+1, \text{k}, k+1, \cdots, k, k+1}_{m \text{ x } (k, k+1)\text{'s}}$$

Cache: 1 (m), 2 (m), …, k-1 (m), k+1 (m)

$\textbf{LFU} = (k-1) + 2m$

OPT cache: 1, 2, …, k+1, k

$\textbf{OPT} = k + 1$

LFU competitive ratio is unbounded

- Consider the sequence of requests:

$$\underbrace{1, 1, \cdots, 1}_{m \text{ x } 1\text{'s}}, \underbrace{2, 2, \cdots, 2}_{m \text{ x } 2\text{'s}}, \underbrace{3, 3, \cdots, 3}_{m \text{ x } 3\text{'s}}, \cdots, \underbrace{k-1, k-1, \cdots, k-1}_{m \text{ x } k-1\text{'s}}, \underbrace{k, k+1, \text{k}, k+1, \cdots, k, k+1}_{m \text{ x } (k, k+1)\text{'s}}$$

Cache	1 (m)	2 (m)	...	k-1 (m)	k+1 (m)
OPT cache	1	2	...	k+1	k

LFU $= (k-1) + 2m$

OPT $= k + 1$

When m is large enough, $\frac{\text{LFU}}{\text{OPT}} \approx O(\frac{m}{k})$

The ratio grows with the input $\Rightarrow$ **unbounded** $\square$

What Happened

- By a special requests sequence, we can force LFU to incur page faults frequently while the optimal assignment is still efficient

- The competitive ratio of LFU grows with the input size — **unbounded**

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

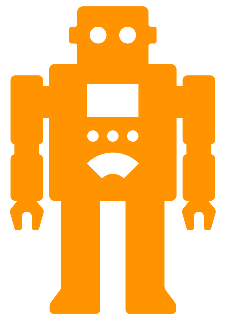
Once a page fault is incurred, evict the one that was used the least recently

Cache

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently

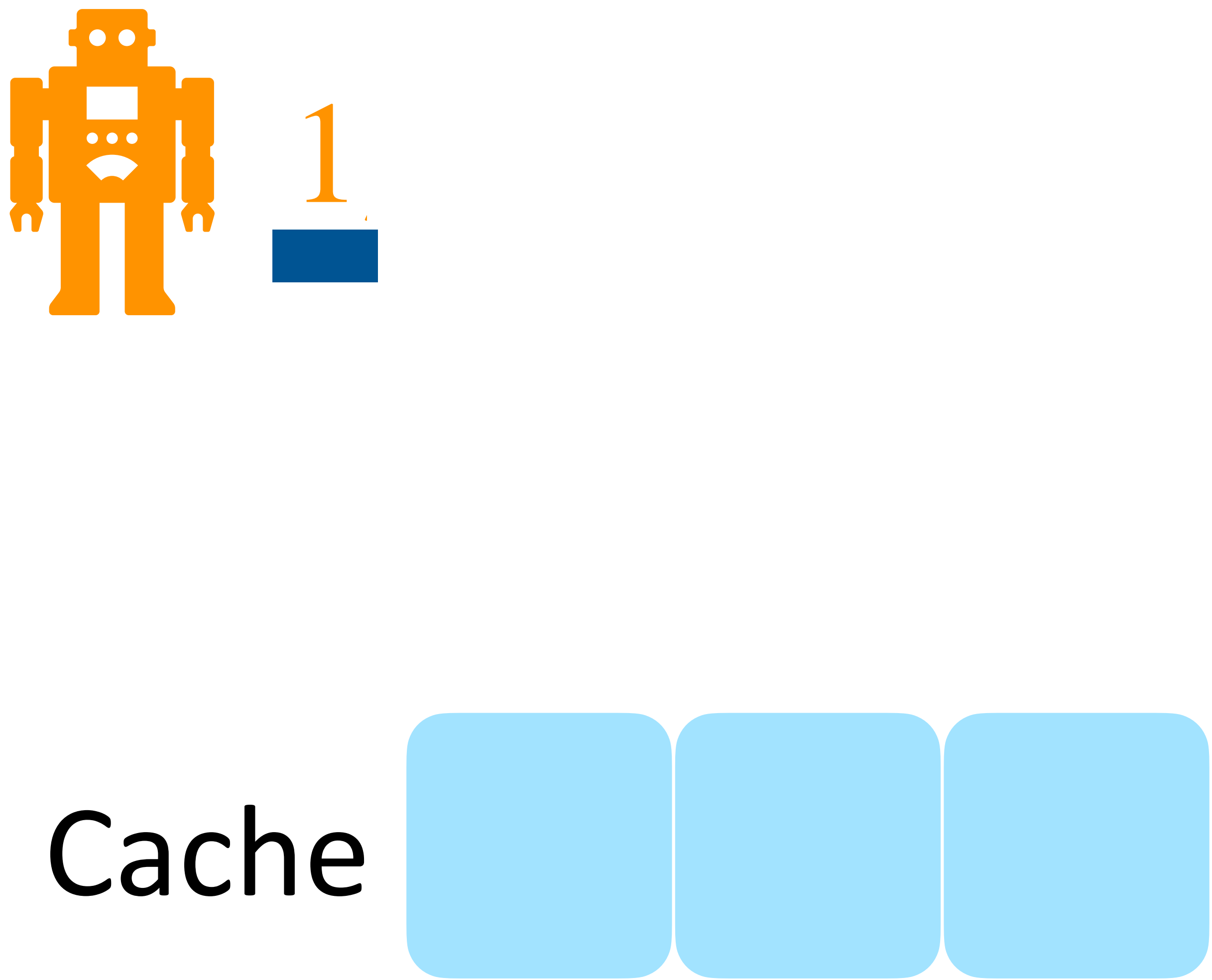
 1

Cache

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

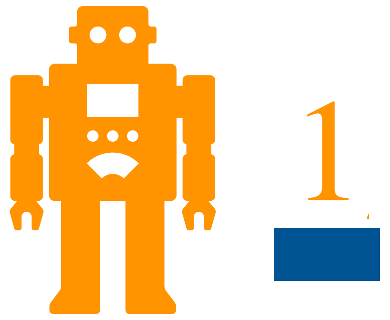
Once a page fault is incurred, evict the one that was used the least recently



LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently

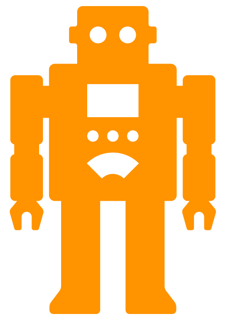


Cache	1		

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



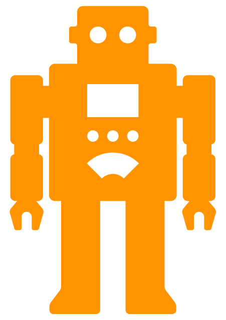
1, 3

Cache	1		

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



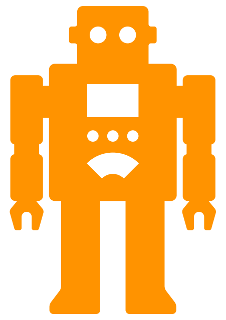
1, 3

Cache | 1 | 3 | |

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



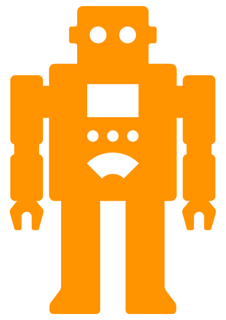
1, 3, 3

Cache	1	3	

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



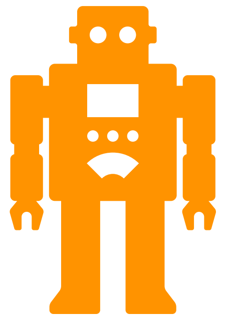
1, 3, 3, 5

Cache | 1 | 3 | 5 |

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



1, 3, 3, 5, 4

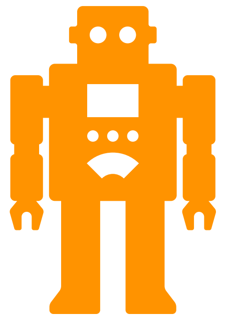
Cache

1	3	5

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



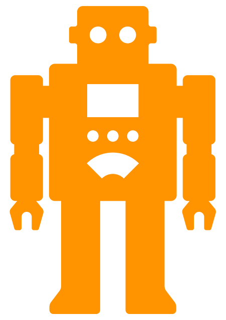
★ 1, 3, ★ 3, ★ 5, 4

Cache | 1 | 3 | 5 |

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



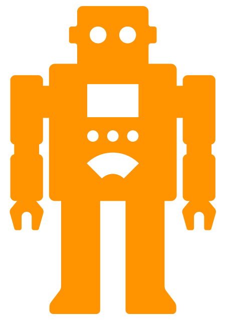
★ ★ ★
1, 3, 3, 5, 4

Cache | 4 | 3 | 5

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



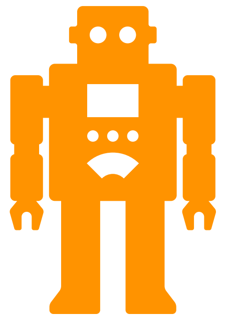
1, 3, 3, 5, 4, 3

Cache | 4 | 3 | 5

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



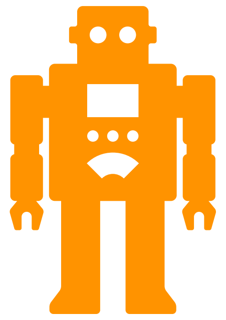
1, 3, 3, 5, 4, 3, 2

Cache | 4 | 3 | 5

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



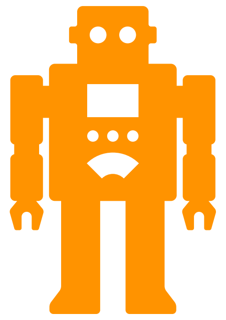
1, 3, 3, 5★, 4★, 3★, 2

Cache | 4 | 3 | 5

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



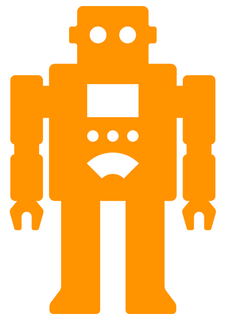
1, 3, 3, 5, 4, 3, 2

Cache | 4 | 3 | 2 |

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



1, 3, 3, 5, 4, 3, 2

Cache | 4 | 3 | 2

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently

1, 3, 3, 5, 4, 3, 2, 5

Cache 4 3 2

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently

1, 3, 3, 5, 4, 3, 2, 5

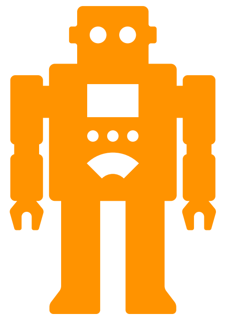
Cache

4	3	2

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



1, 3, 3, 5, 4★, 3★, 2★, 5

Cache | 5 | 3 | 2 |

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently

1, 3, 3, 5, 4, 3, 2, 5, 2

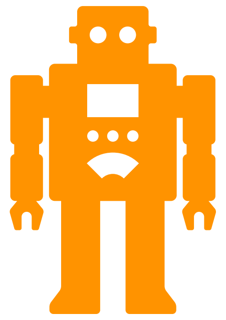
Cache

5	3	2

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently

 1, 3, 3, 5, 4, 3, 2, 5, 2, 1

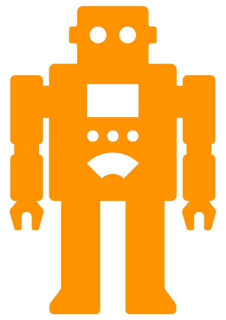
Cache

5	3	2

LRU (Least-Recently-Used)

LRU (Least-Recently-Used) algorithm:

Once a page fault is incurred, evict the one that was used the least recently



1, 3, 3, 5, 4, 3, 2, 5, 2, 1

Cache | 5 | 1 | 2 |

LRU is k-competitive

LRU is k-competitive

<Proof idea>

LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

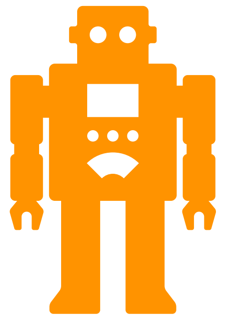
- Phase 0 is empty
- Phase i is the maximal sequence following phase $i-1$ that contains at most k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page)

LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Phase 0 is empty
- Phase i is the maximal sequence following phase $i-1$ that contains at most k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page)



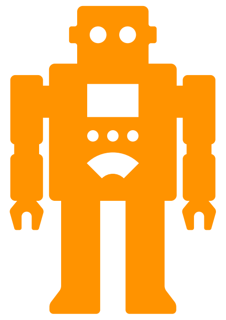
$1, 3, 3, 5, 4, 3, 2, 5, 2, 1, 1, 3, 2, 3, 1, 3, 3, 5, 3, 5, 2, 1$

LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Phase 0 is empty
- Phase i is the maximal sequence following phase $i - 1$ that contains at most k distinct page requests (that is, phase $i + 1$ begins on the request that is the $(k + 1)$-th distinct page)



$\boxed{1, 3, 3, 5}$ $4, 3, 2, 5, 2, 1, 1, 3, 2, 3, 1, 3, 3, 5, 3, 5, 2, 1$

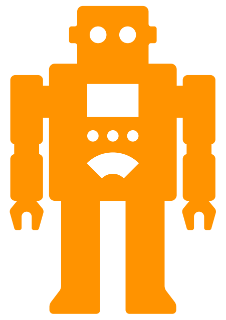
Phase 1

LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Phase 0 is empty
- Phase i is the maximal sequence following phase $i-1$ that contains at most k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page)



$\boxed{1, 3, 3, 5}\ \boxed{4, 3, 2}\ 5, 2, 1, 1, 3, 2, 3, 1, 3, 3, 5, 3, 5, 2, 1$

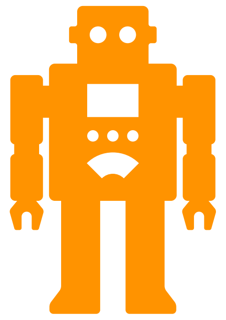
Phase 1 Phase 2

LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Phase 0 is empty
- Phase i is the maximal sequence following phase $i-1$ that contains at most k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page)



$\boxed{1, 3, 3, 5}, \boxed{4, 3, 2}, \boxed{5, 2, 1, 1}, 3, 2, 3, 1, 3, 3, 5, 3, 5, 2, 1$

Phase 1 Phase 2 Phase 3

LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Phase 0 is empty
- Phase i is the maximal sequence following phase $i-1$ that contains at most k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page)

$\boxed{1, 3, 3, 5}\,\boxed{4, 3, 2}\,\boxed{5, 2, 1, 1}\,\boxed{3, 2, 3, 1, 3, 3}\, 5, 3, 5, 2, 1$

Phase 1 Phase 2 Phase 3 Phase 4

LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

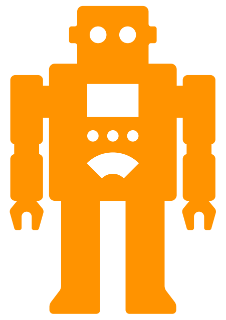
- Phase 0 is empty
- Phase i is the maximal sequence following phase $i-1$ that contains at most k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page)

Phase 1	Phase 2	Phase 3	Phase 4	Phase 5	
1, 3, 3, 5,	4, 3, 2,	5, 2, 1, 1,	3, 2, 3, 1, 3, 3,	5, 3, 5, 2,	1

LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

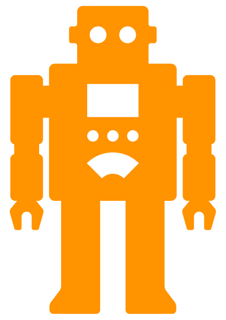
 $\boxed{1, 3, 3, 5,}\boxed{4, 3, 2,}\boxed{5, 2, 1, 1,}\boxed{3, 2, 3, 1, 3, 3,}\boxed{5, 3, 5, 2,}\boxed{1}$

LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Claim (a): In phase i, **LRU** only incurs at most k page faults



$\boxed{1, 3, 3, 5,}\ \boxed{4, 3, 2,}\ \boxed{5, 2, 1, 1,}\ \boxed{3, 2, 3, 1, 3, 3,}\ \boxed{5, 3, 5, 2,}\ \boxed{1}$

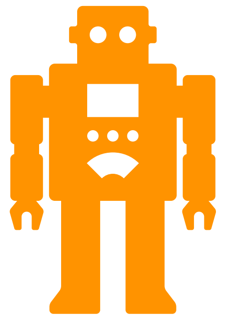
LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Claim (a): In phase i, **LRU** only incurs at most k page faults

  <Proof> Since **LRU** evicts the page that was used the longest time ago, it never evicts a page that was requested earlier in the same phase. Hence, **LRU** incurs at most k page faults in a phase as there are at most k distinct pages in each phase i.



$\boxed{1, 3, 3, 5,}\boxed{4, 3, 2,}\boxed{5, 2, 1, 1,}\boxed{3, 2, 3, 1, 3, 3,}\boxed{5, 3, 5, 2,}\boxed{1}$

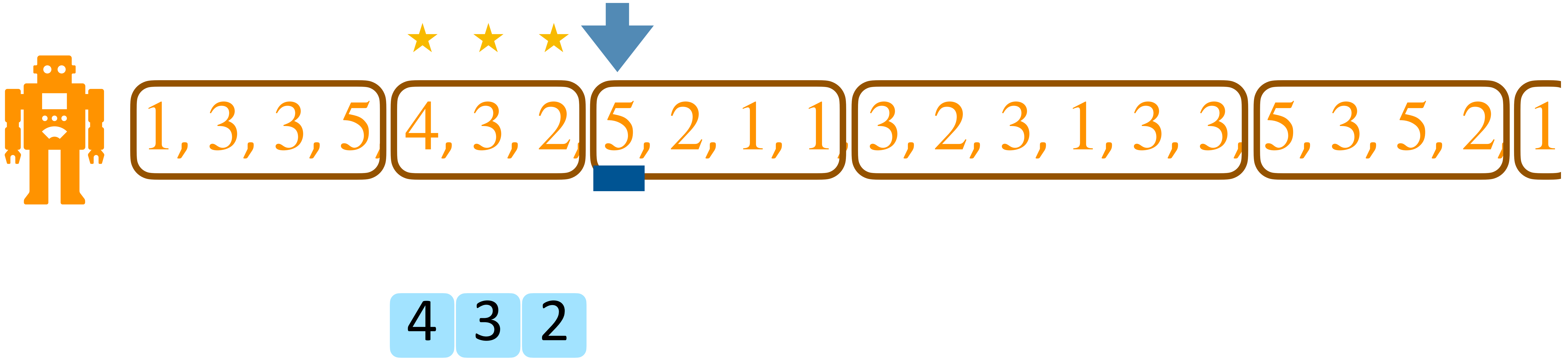
LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Claim (a): In phase i, **LRU** only incurs at most k page faults

  <Proof> Since **LRU** evicts the page that was used the longest time ago, it never evicts a page that was requested earlier in the same phase. Hence, **LRU** incurs at most k page faults in a phase as there are at most k distinct pages in each phase i.



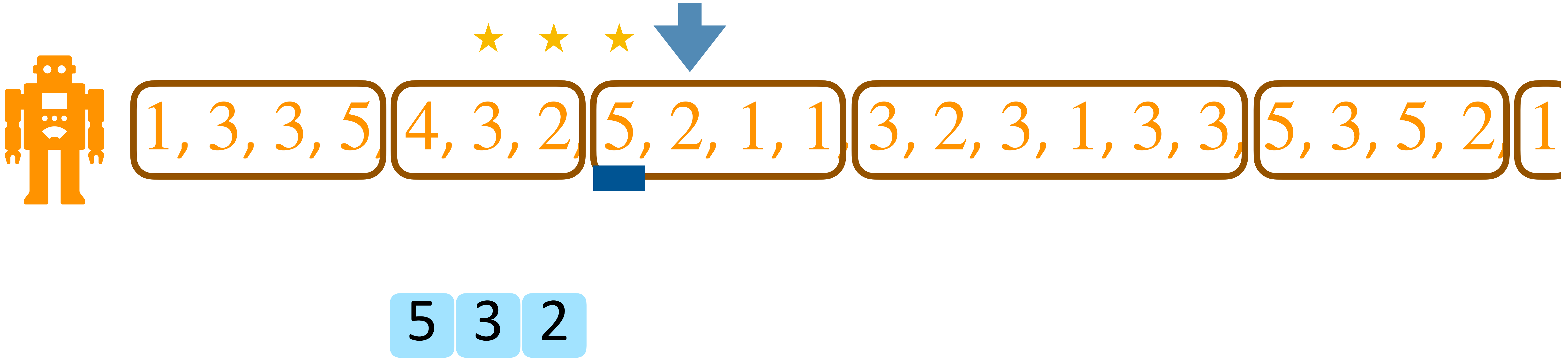
LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Claim (a): In phase i, **LRU** only incurs at most k page faults

<Proof> Since **LRU** evicts the page that was used the longest time ago, it never evicts a page that was requested earlier in the same phase. Hence, **LRU** incurs at most k page faults in a phase as there are at most k distinct pages in each phase i.



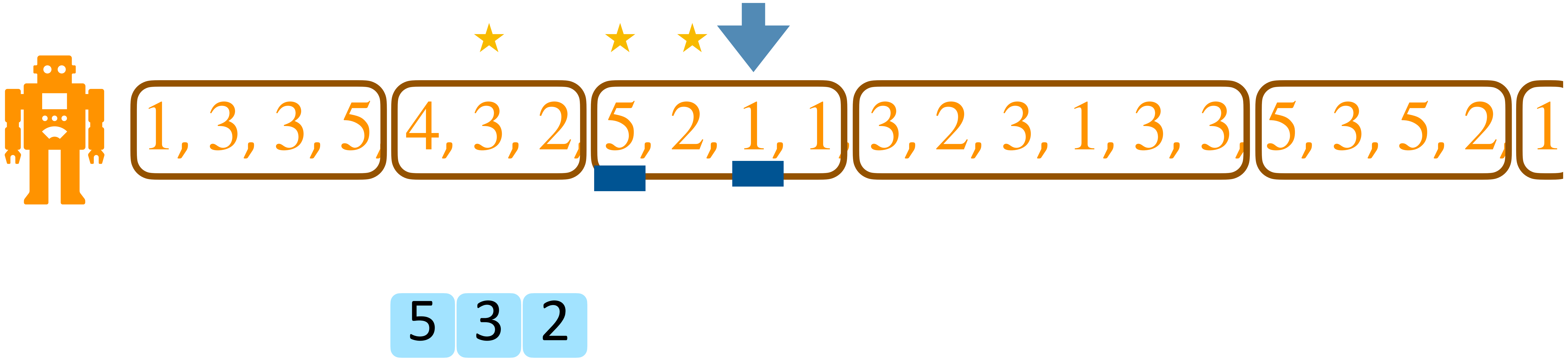
LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Claim (a): In phase i, **LRU** only incurs at most k page faults

<Proof> Since **LRU** evicts the page that was used the longest time ago, it never evicts a page that was requested earlier in the same phase. Hence, **LRU** incurs at most k page faults in a phase as there are at most k distinct pages in each phase i.



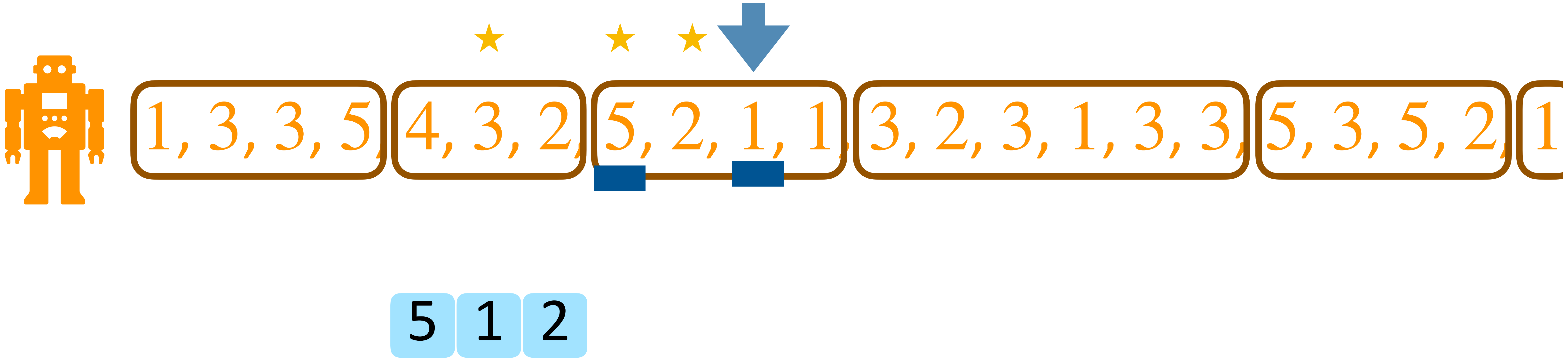
LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Claim (a): In phase i, **LRU** only incurs at most k page faults

<Proof> Since **LRU** evicts the page that was used the longest time ago, it never evicts a page that was requested earlier in the same phase. Hence, **LRU** incurs at most k page faults in a phase as there are at most k distinct pages in each phase i.



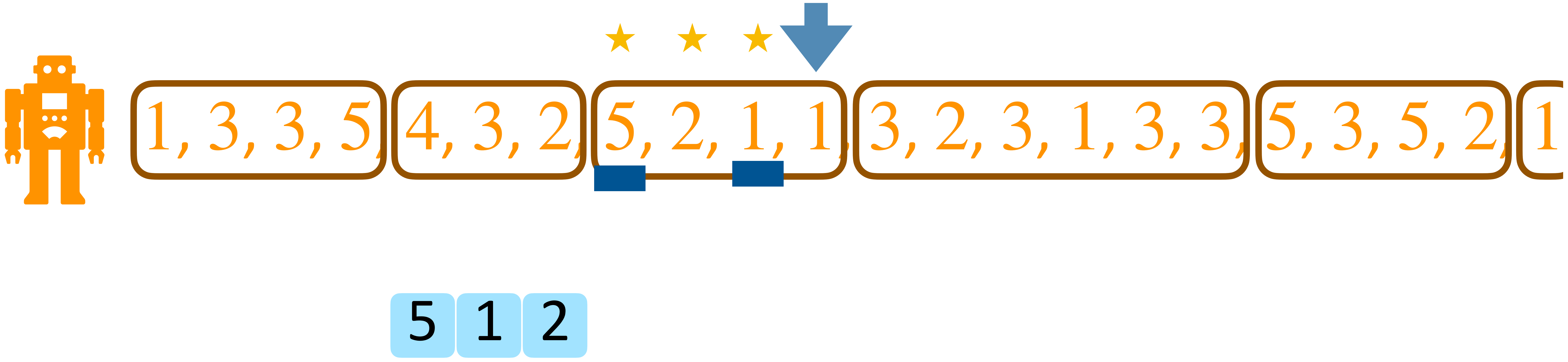
LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Claim (a): In phase i, **LRU** only incurs at most k page faults

  <Proof> Since **LRU** evicts the page that was used the longest time ago, it never evicts a page that was requested earlier in the same phase. Hence, **LRU** incurs at most k page faults in a phase as there are at most k distinct pages in each phase i.



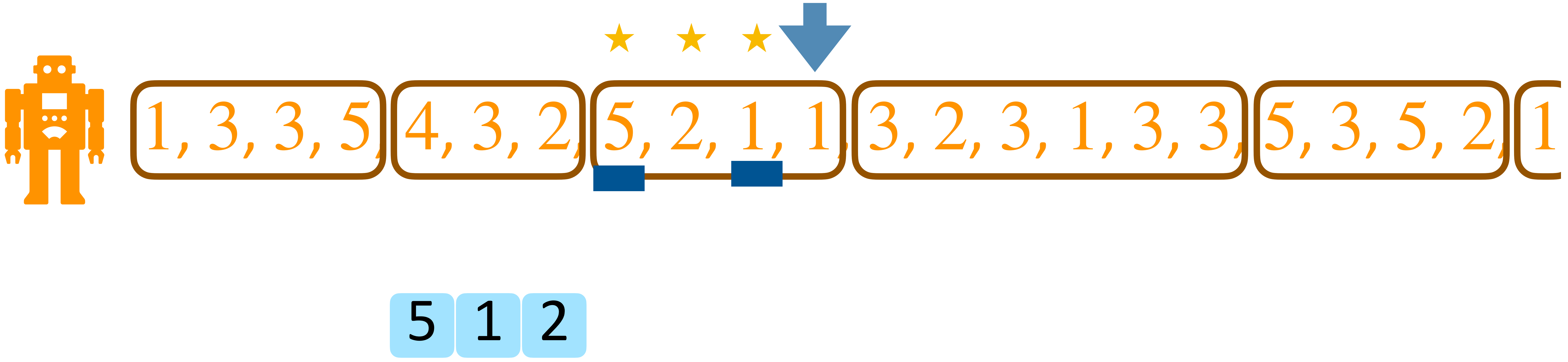
LRU is k-competitive

<Proof idea>

Phase partitioning: partition the request sequence into phases and bound the cost of LRU and OPT in each phase

- Claim (a): In phase i, **LRU** only incurs at most k page faults

<Proof> Since **LRU** evicts the page that was used the longest time ago, it never evicts a page that was requested earlier in the same phase. Hence, **LRU** incurs at most k page faults in a phase as there are at most k distinct pages in each phase i.



At the moment when the j-th distinct page in phase i is requested, there are $j-1$ pages accessed in phase i.
$\Rightarrow$ There are $k-(j-1)$ pages in the cache that haven't been accessed recently. Hence, LRU will evict one of them.

LRU is k-competitive

<Proof idea>

- Claim (b): In phase i, **OPT** incurs at least 1 page fault

LRU is k-competitive

<Proof idea>

- Claim (b): In phase i, **OPT** incurs at least 1 page fault

<Proof> Consider the cache just before phase i. At this moment, there are k pages in the cache. Since the first request in phase i is different from any of the pages in phase $i - 1$, any feasible algorithm has to evict one page to accommodate this request, and so does **OPT**.

1, 3, 3, 5 | 4, 3, 2 | 5, 2, 1, 1 | 3, 2, 3, 1, 3, 3 | 5, 3, 5, 2 | 1

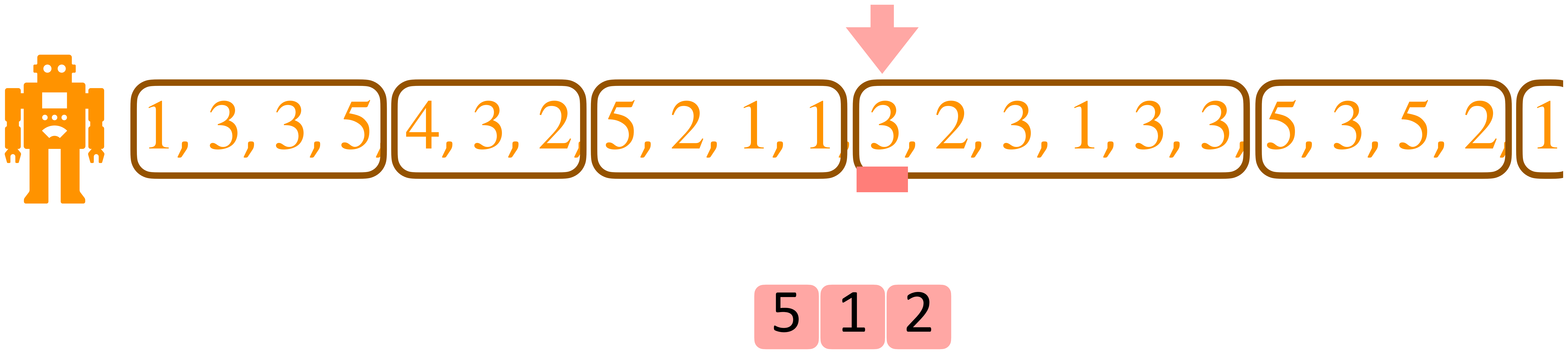
5	1	2

LRU is k-competitive

<Proof idea>

- Claim (b): In phase i, **OPT** incurs at least 1 page fault

<Proof> Consider the cache just before phase i. At this moment, there are k pages in the cache. Since the first request in phase i is different from any of the pages in phase $i-1$, any feasible algorithm has to evict one page to accommodate this request, and so does **OPT**.

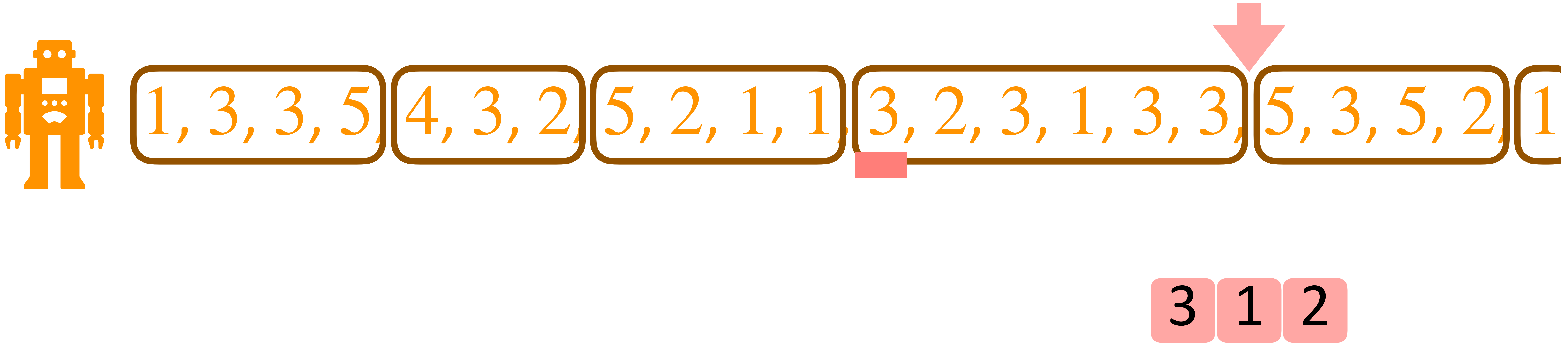


LRU is k-competitive

<Proof idea>

- Claim (b): In phase i, **OPT** incurs at least 1 page fault

  <Proof> Consider the cache just before phase i. At this moment, there are k pages in the cache. Since the first request in phase i is different from any of the pages in phase $i - 1$, any feasible algorithm has to evict one page to accommodate this request, and so does **OPT**.

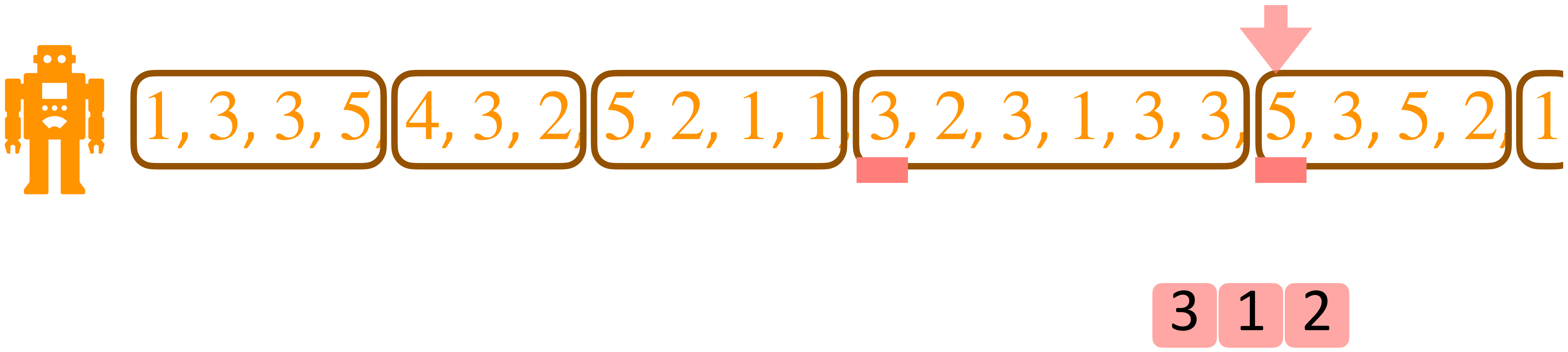


LRU is k-competitive

<Proof idea>

- Claim (b): In phase i, **OPT** incurs at least 1 page fault

  <Proof> Consider the cache just before phase i. At this moment, there are k pages in the cache. Since the first request in phase i is different from any of the pages in phase $i-1$, any feasible algorithm has to evict one page to accommodate this request, and so does **OPT**.

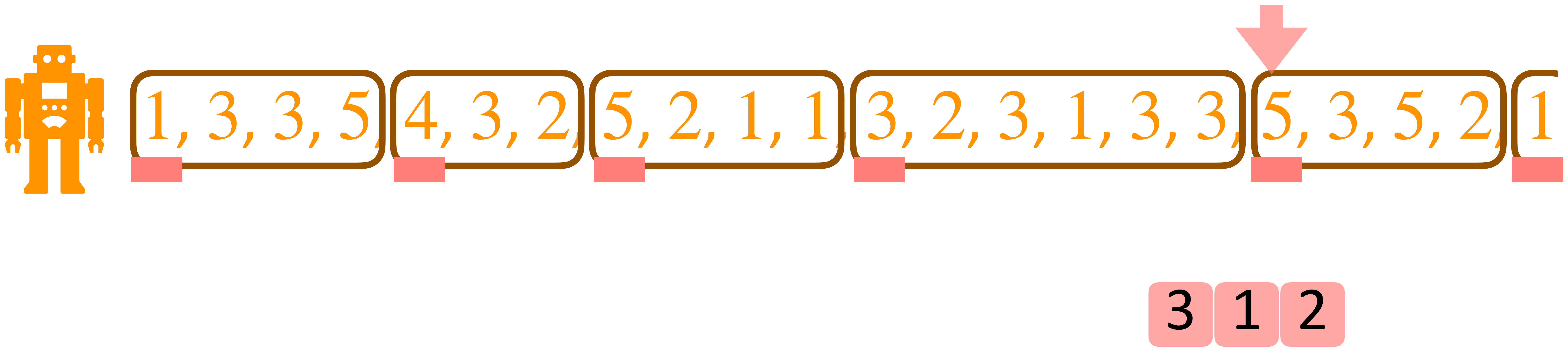


LRU is k-competitive

<Proof idea>

- Claim (b): In phase i, **OPT** incurs at least 1 page fault

<Proof> Consider the cache just before phase i. At this moment, there are k pages in the cache. Since the first request in phase i is different from any of the pages in phase $i-1$, any feasible algorithm has to evict one page to accommodate this request, and so does **OPT**.

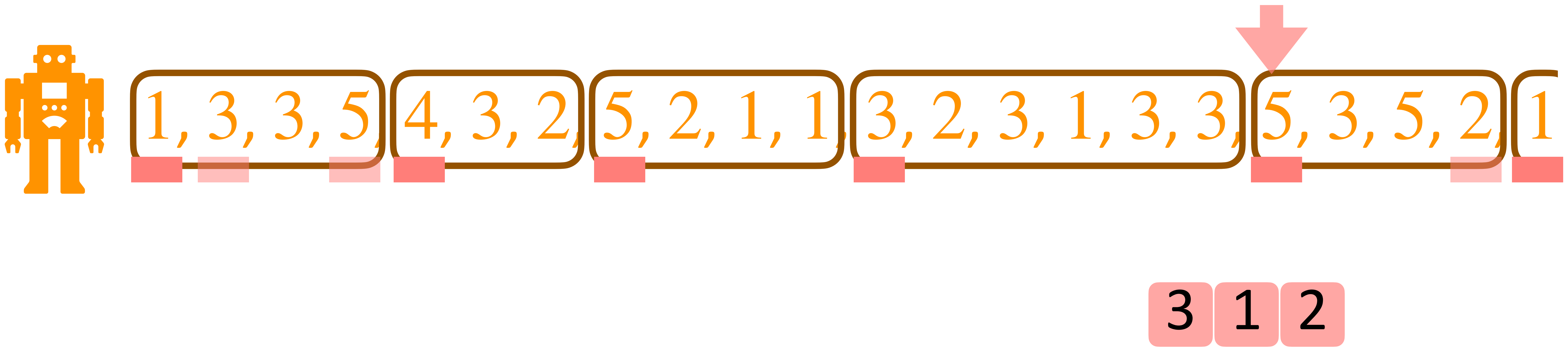


LRU is k-competitive

<Proof idea>

- Claim (b): In phase i, **OPT** incurs at least 1 page fault

  <Proof> Consider the cache just before phase i. At this moment, there are k pages in the cache. Since the first request in phase i is different from any of the pages in phase $i-1$, any feasible algorithm has to evict one page to accommodate this request, and so does **OPT**.



LRU is k-competitive

<Proof idea>

- Claim (b): In phase i, **OPT** incurs at least 1 page fault

  <Proof> Consider the cache just before phase i. At this moment, there are k pages in the cache. Since the first request in phase i is different from any of the pages in phase $i-1$, any feasible algorithm has to evict one page to accommodate this request, and so does **OPT**.

Let LRU_i and OPT_i denote the page fault incurred by **LRU** and **OPT** in phase i, respectively. By Claim (a) and Claim (b),

$$\frac{\text{LRU}(I)}{\text{OPT}(I)} = \frac{\sum_i \text{LRU}_i}{\sum_i \text{OPT}_i} \leq \frac{k}{1} = k$$

$\square$

What Happened

- **Phase partitioning**: partition the request sequence into phases such that each phase has k distinct pages

- By arguing that an algorithm incurs at most k page faults and OPT incurs at least 1 page fault in any phase, we can conclude that the algorithm is at most $O(\frac{n}{k})$-competitive

- Arguing that **an algorithm incurs at most k page faults** is the key!

Paging Problem Lower Bound

Paging Problem Lower Bound

<Proof idea>

Assume that the cache size is k. Consider any algorithm **ALG** and design the adversary as follows: First request pages $1, 2, 3, \cdots, k$

1	2	3	…	…	k

Paging Problem Lower Bound

<Proof idea>

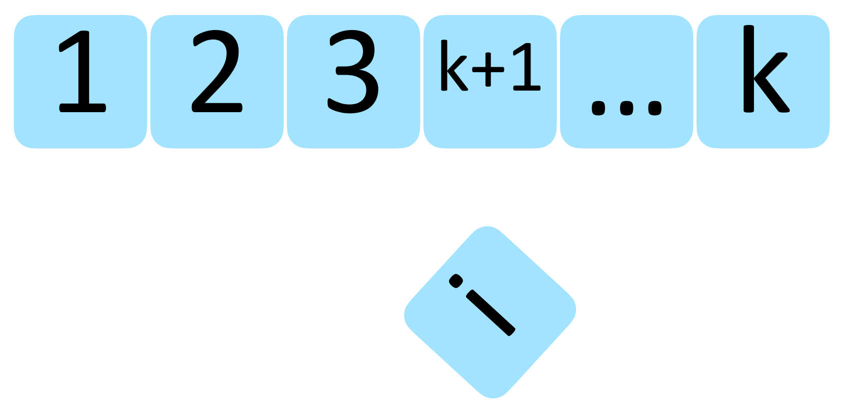
Assume that the cache size is k. Consider any algorithm **ALG** and design the adversary as follows: First request pages $1, 2, 3, \cdots, k, k+1$.

1	2	3	...	...	k

Paging Problem Lower Bound

<Proof idea>

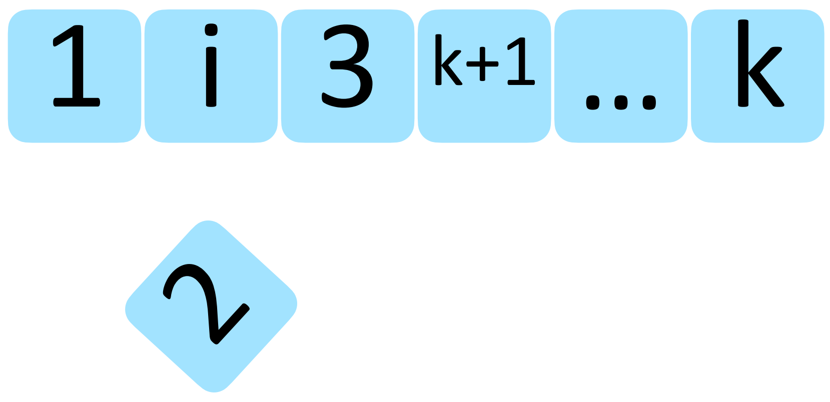
Assume that the cache size is k. Consider any algorithm **ALG** and design the adversary as follows: First request pages $1, 2, 3, \cdots, k, k+1$. At this moment, **ALG** evicts a page $i \in [1,k]$.



Paging Problem Lower Bound

<Proof idea>

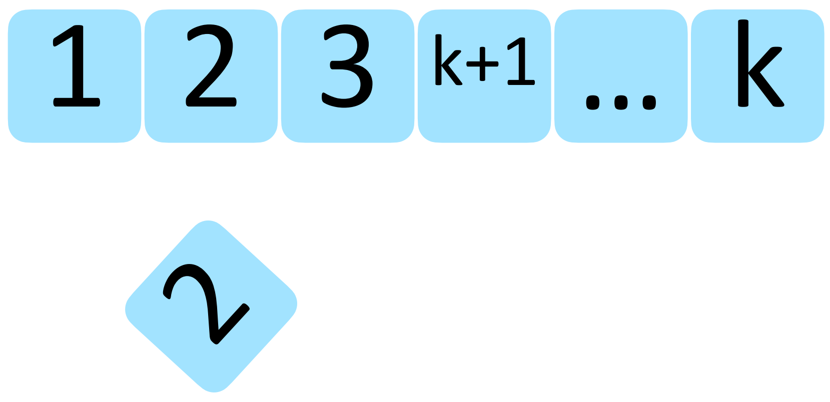
Assume that the cache size is k. Consider any algorithm **ALG** and design the adversary as follows: First request pages $1, 2, 3, \cdots, k, k+1$. At this moment, **ALG** evicts a page $i \in [1,k]$. Then, the adversary requests page i.



Paging Problem Lower Bound

<Proof idea>

Assume that the cache size is k. Consider any algorithm **ALG** and design the adversary as follows: First request pages $1, 2, 3, \cdots, k, k+1$. At this moment, **ALG** evicts a page $i \in [1,k]$. Then, the adversary requests page i. The adversary repeatedly requests the page evicted by **ALG** for $n-1$ rounds.



Paging Problem Lower Bound

<Proof idea>

Assume that the cache size is k. Consider any algorithm **ALG** and design the adversary as follows: First request pages $1, 2, 3, \cdots, k, k+1$. At this moment, **ALG** evicts a page $i \in [1,k]$. Then, the adversary requests page i. The adversary repeatedly requests the page evicted by **ALG** for $n-1$ rounds.

In this instance, each request incurs a page fault for ALG. Therefore, **ALG** costs $k+n$.

Paging Problem Lower Bound

<Proof idea>

Assume that the cache size is k. Consider any algorithm **ALG** and design the adversary as follows: First request pages $1, 2, 3, \cdots, k, k+1$. At this moment, **ALG** evicts a page $i \in [1,k]$. Then, the adversary requests page i. The adversary repeatedly requests the page evicted by **ALG** for $n-1$ rounds.

In this instance, each request incurs a page fault for ALG. Therefore, **ALG** costs $k+n$.

Because there are only $k+1$ pages involved, **OPT** incurs at most 1 page fault per k pages.

Paging Problem Lower Bound

<Proof idea>

Assume that the cache size is k. Consider any algorithm **ALG** and design the adversary as follows: First request pages $1, 2, 3, \cdots, k, k+1$. At this moment, **ALG** evicts a page $i \in [1,k]$. Then, the adversary requests page i. The adversary repeatedly requests the page evicted by **ALG** for $n-1$ rounds.

In this instance, each request incurs a page fault for ALG. Therefore, **ALG** costs $k+n$.

Because there are only $k+1$ pages involved, **OPT** incurs at most 1 page fault per k pages.

Therefore, $\dfrac{\text{ALG}(I)}{\text{OPT}(I)} \geq \dfrac{k+n}{k+n/k} \approx \Omega(k)$

Even when every page requests change dramatically, the optimal solution can keep the k pages that will be used in the most recent future and evict the one that will be used later.

Paging Problem Lower Bound

<Proof idea>

Assume that the cache size is k. Consider any algorithm **ALG** and design the adversary as follows: First request pages $1, 2, 3, \cdots, k, k+1$. At this moment, **ALG** evicts a page $i \in [1,k]$. Then, the adversary requests page i. The adversary repeatedly requests the page evicted by **ALG** for $n-1$ rounds.

In this instance, each request incurs a page fault for ALG. Therefore, **ALG** costs $k+n$.

Because there are only $k+1$ pages involved, **OPT** incurs at most 1 page fault per k pages.

Therefore, $\dfrac{\text{ALG}(I)}{\text{OPT}(I)} \geq \dfrac{k+n}{k+n/k} \approx \Omega(k)$

$\square$

Research cycle of online algorithms