

# COMP 691: Online Algorithms and Competitive Analysis

## Lecture 8

Denis Pankratov

Office Hour: Th 1:00PM – 2:00PM @ EV 3.127

Or: appointments by email

# The time before the midterm

- Online graph problems
  - Input models: EM, VAM-PH, EAM, VAM-FI, BVAM
  - Hard online problems: Maximum Independent Set, Maximum Clique, Longest Path, Minimum Spanning Tree, Travelling Salesperson Problem
  - Maximum Bipartite Matching
- **Announcement:** A3 and projects will be posted this weekend on Moodle

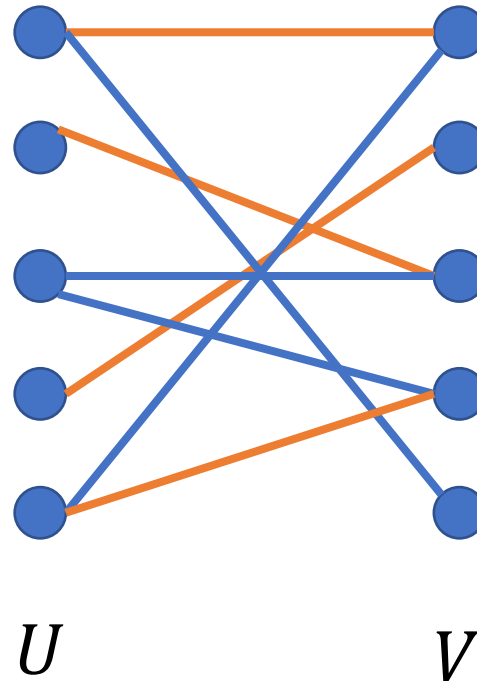
# Maximum Bipartite Matching

# Maximum Bipartite Matching

Bipartite graph  $G = (U, V, E)$

Matching  $M \subseteq E$  is a subset of vertex-disjoint edges

$\forall e_1, e_2 \in E$  we have  $e_1 \cap e_2 = \emptyset$

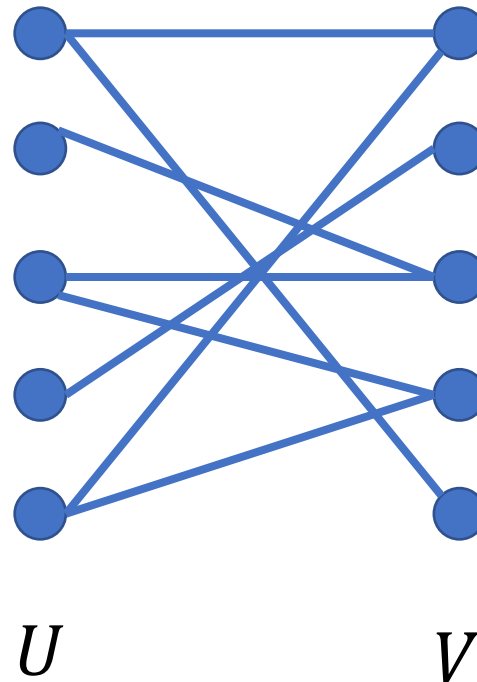


# Vertex Arrival Model for bipartite graphs

One side  $V$  known in advance, called the **offline side**

Another side  $U$  arrives one node at a time, called the **online side**

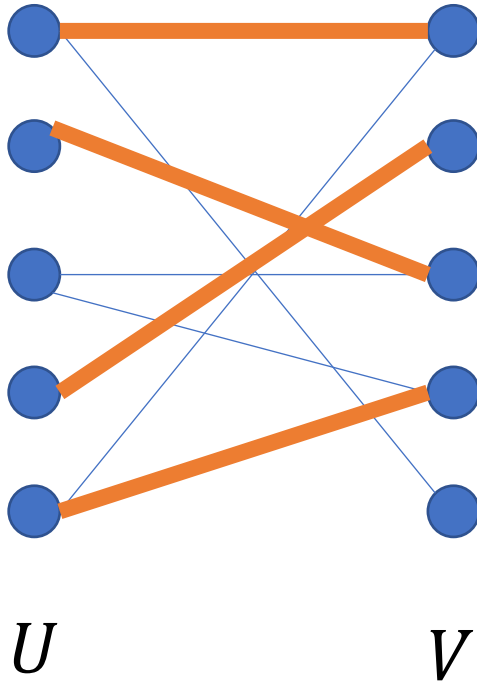
When node  $u \in U$  arrives also learn  $N(u) \subseteq V$



# Maximum Bipartite Matching

- Input:**  $G = (U, V, E, <)$  unweighted undirected bipartite graph  
 $<$  is a total order on  $U$   
 $(u_1, N(u_1)), (u_2, N(u_2)), \dots, (u_n, N(u_n))$  input sequence,  
where  $u_1 < u_2 < \dots < u_n$
- Output:**  $d_1, \dots, d_n$  where  $d_i \in N(u_i) \cup \{\perp\}$  indicates how to match  $u_i$  to its neighbor ( $\perp$  indicates  $u_i$  remains unmatched)
- Objective:** maximize the size of the constructed matching  
 $M = \{\{u_i, d_i\} : d_i \neq \perp\}$  subject to  $M$  being a well-defined matching

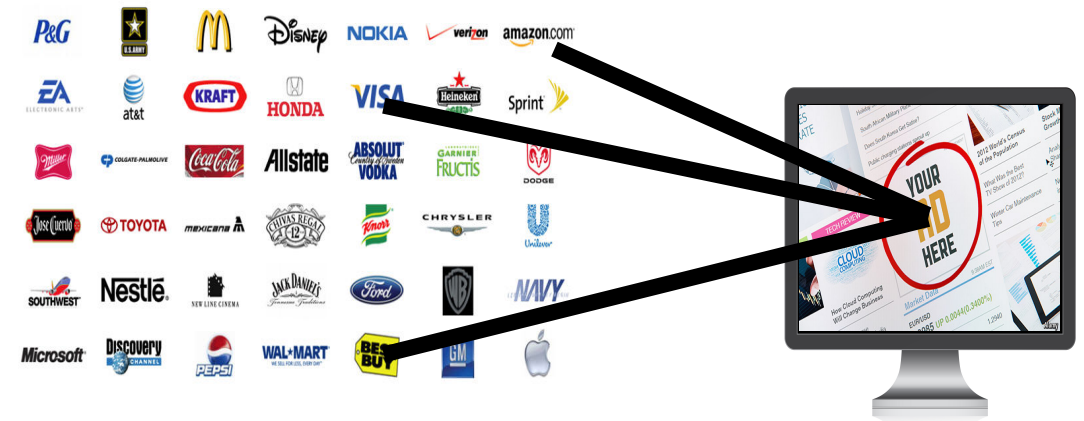
# Maximum Bipartite Matching



# Example Application: Online Advertising

Consider an online ad platform called **G** (for no particular reason)

- **G** has a database of advertisers
- Each advertiser has a limit on the number of times to be displayed
- User **U** clicks on a website
- **G** knows **U** better than **U** knows **U**
- **G** knows advertisers compatible with **U**



**G** wants to maximize the number of ads shown.

The underlying combinatorial problem: online bipartite matching.



# Short History of Nearly Everything re Bipartite Matching

- Offline
  - rich history dating as far back as 1931 (König and Egerváry), maybe earlier
  - solvable optimally in polytime: Hopcroft-Karp [1974], matrix-mult. approach of Mucha-Sankowski [2004], electrical flow approach in sparse regime of Madry [2013], even more efficient approx. approaches.
- Online
  - introduced by Karp, Vazirani, Vazirani [1990]: adversarial input model
  - not solvable optimally: tight approx. ratio of  $1/2$  by det. algos, tight approx. ratio of  $1 - 1/e = 0.632\dots$  by rand. algos
  - resurgence of interest in 2009 – Feldman et al. beat  $1-1/e$  in known i.i.d., applications to online advertising
  - Bahmani, Kapralov [2010], Manshadi et al. [2011], Jaillet, Lu [2014], ...

# Simple Greedy Algorithm

Pick an ordering  $\sigma$  of offline vertices  $V$

When a new item  $(u, N(u))$  arrives

if there are no unmatched neighbors then  $u$  is unmatched

otherwise, pick the first neighbor  $v$  according to  $\sigma$ , match  $u$  to  $v$

Some notation:  $\sigma : V \rightarrow [n]$  – ordering or ranking

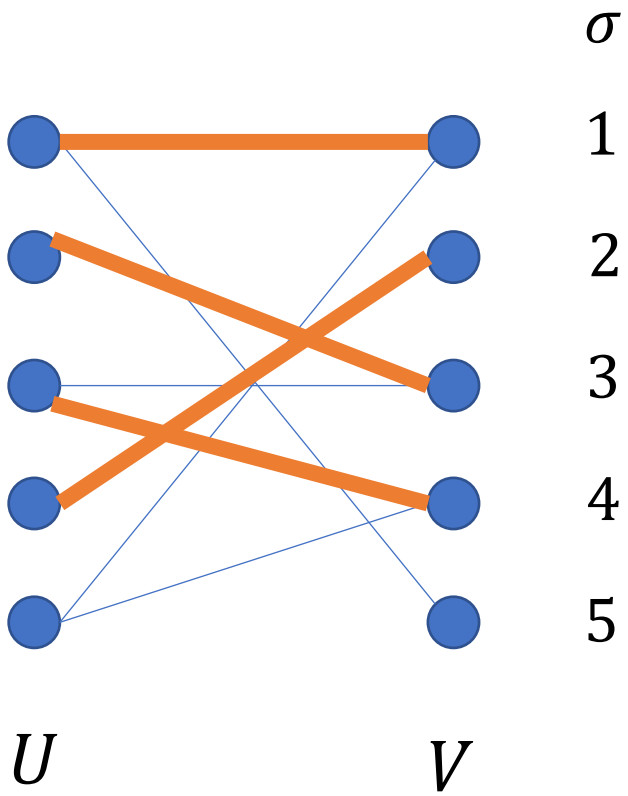
$\sigma(v)$  – rank of vertex  $v$

$\sigma^{-1}(t)$  – vertex of rank  $t$

$v_1$  has better rank than  $v_2$  if  $\sigma(v_1) < \sigma(v_2)$

$N_c(v)$  – currently available (unmatched) neighbors

# Greedy example



---

**Algorithm 13** Simple greedy algorithm for BMM.

---

**procedure** SIMPLEGREEDY $V$  – set of offline verticesFix a ranking  $\sigma$  on vertices  $V$  $M \leftarrow \emptyset$  $i \leftarrow 1$ **while**  $i \leq n$  **do**New online vertex  $u_i$  arrives according to  $\prec$  together with  $N(u_i)$ **if**  $N_c(u) \neq \emptyset$  **then** $\triangleright$  if there is an unmatched vertex in  $N(u_i)$  $\triangleright$  select the vertex of best rank in  $N_c(u_i)$  $v \leftarrow \arg \min \{ \sigma(v) : v \in N_c(u) \}$  $M \leftarrow M \cup \{ (u_i, v) \}$  $\triangleright$  match  $u_i$  with  $v$  $i \leftarrow i + 1$ 

---

## Theorem

$$\rho(\text{SimpleGreedy}) = 2$$

### Proof:

$$\text{Part 1: } \rho(\text{SimpleGreedy}) \geq 2$$

We present a **gadget-based** proof

We use the following gadget:

Adversary declares two offline vertices  $v_1, v_2$

Without loss of generality assume that  $\sigma(v_1) < \sigma(v_2)$

Adversary presents  $(u_1, \{v_1, v_2\}), (u_2, \{v_1\})$

## Theorem

$$\rho(\text{SimpleGreedy}) = 2$$

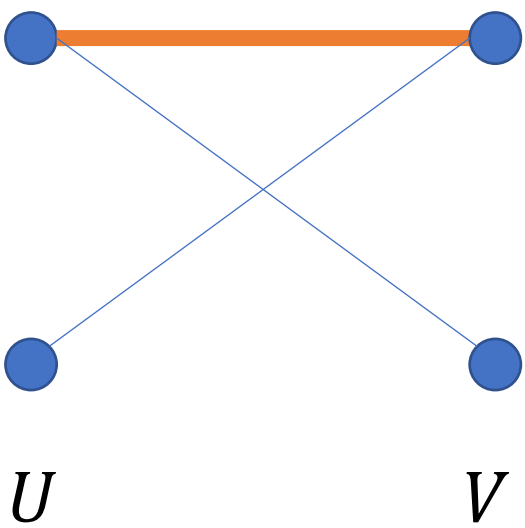
**Proof:**

Part 1:  $\rho(\text{SimpleGreedy}) \geq 2$

$ALG = 1$

$u_1$    $v_1, \sigma(v_1) = 1$

$OPT = 2$

$u_2$    $v_2, \sigma(v_2) = 2$

$U$

$V$

Competitive ratio = 2

## Theorem

$$\rho(\text{SimpleGreedy}) = 2$$

### Proof:

$$\text{Part 1: } \rho(\text{SimpleGreedy}) \geq 2$$

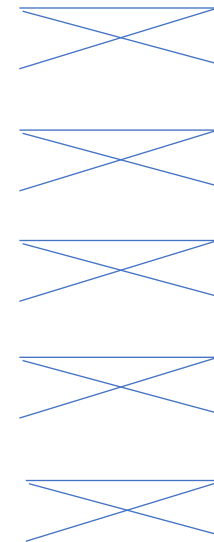
This proves that strict competitive ratio is  $\geq 2$

What about asymptotic?

Repeat the gadget!

Gives asymptotic competitive ratio

Can be used to show  $\rho(\text{ALG}) \geq 2$  for any deterministic algorithm  $\text{ALG}$



## Theorem

$$\rho(\text{SimpleGreedy}) = 2$$

### Proof:

Part 2:  $\rho(\text{SimpleGreedy}) \leq 2$

Let  $M$  be the matching constructed by *SimpleGreedy*

**Claim:**  $M$  is maximal, i.e., there is no edge  $e = \{u, v\}$  that can be added to  $M$

**Pf by contradiction:** suppose that  $e = \{u, v\}$  is such that  $M \cup \{e\}$  is also a valid matching

Therefore,  $u$  and  $v$  both were not matched by *SimpleGreedy*

Why wasn't  $u$  matched? It had at least one available neighbor  $v$ !

Contradiction! QED.



## Theorem

$$\rho(\text{SimpleGreedy}) = 2$$

### Proof:

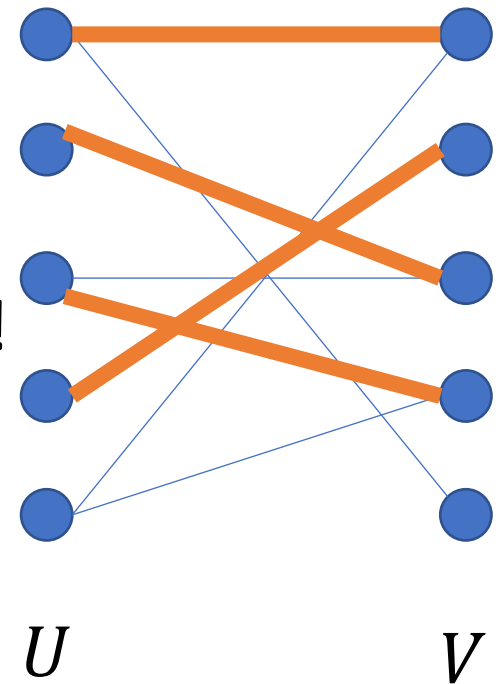
Part 2:  $\rho(\text{SimpleGreedy}) \leq 2$

Let  $M$  be the matching constructed by *SimpleGreedy*

**Claim:**  $M$  is maximal

In other words, every edge is incident on some vertex in  $M$

Therefore, if we take all vertices in  $M$ , we get a vertex cover!



## Theorem

$$\rho(\text{SimpleGreedy}) = 2$$

### Proof:

Part 2:  $\rho(\text{SimpleGreedy}) \leq 2$

Let  $M$  be the matching constructed by *SimpleGreedy*

Let  $OPT$  denote the size of maximum matching

Let  $VC(G)$  denote the size of minimum vertex cover

Since  $M$  is maximal, we get  $VC(G) \leq 2|M| = 2ALG$

Trivially, we have  $VC(G) \geq OPT$  (why?)

Therefore  $OPT \leq 2 ALG$

QED

# Deterministic Bipartite Matching

To sum up:

deterministic algorithms have  $\rho(ALG) \geq 2$

*SimpleGreedy* achieves this bound

What about randomized algorithms?

# Simple Randomized Algorithm

Attempt 1: *SimpleRandom*

When an item  $(u, N(u))$  arrives

form  $N_c(u)$  – unmatched neighbors of  $u$

match  $u$  to a uniformly random element of  $N_c(u)$

Exercise:

$$\rho_{OBL}(\textit{SimpleRandom}) = 2$$

Thus, *SimpleRandom* does not improve upon *SimpleGreedy*

# Ranking Algorithm

Attempt 2: *Ranking* algorithm, or KVV (Karp, Vazirani, Vazirani)

Pick an ordering/ranking  $\sigma$  on  $V$  uniformly at random

Run *SimpleGreedy* with respect to  $\sigma$

We will prove

Theorem

$$\rho_{OBL}(\textit{Ranking}) = \frac{e}{e-1} \approx 1.582$$

---

**Algorithm 14** The Ranking algorithm for BMM.

---

**procedure** RANKING

$V$  – set of offline vertices

Pick a ranking  $\sigma$  on vertices  $V$  *uniformly at random*

$M \leftarrow \emptyset$

$i \leftarrow 1$

**while**  $i \leq n$  **do**

    New online vertex  $u_i$  arrives according to  $\prec$  together with  $N(u_i)$

**if**  $N_c(u) \neq \emptyset$  **then**

$\triangleright$  if there is an unmatched vertex in  $N(u_i)$

$\triangleright$  select the vertex of best rank in  $N_c(u_i)$

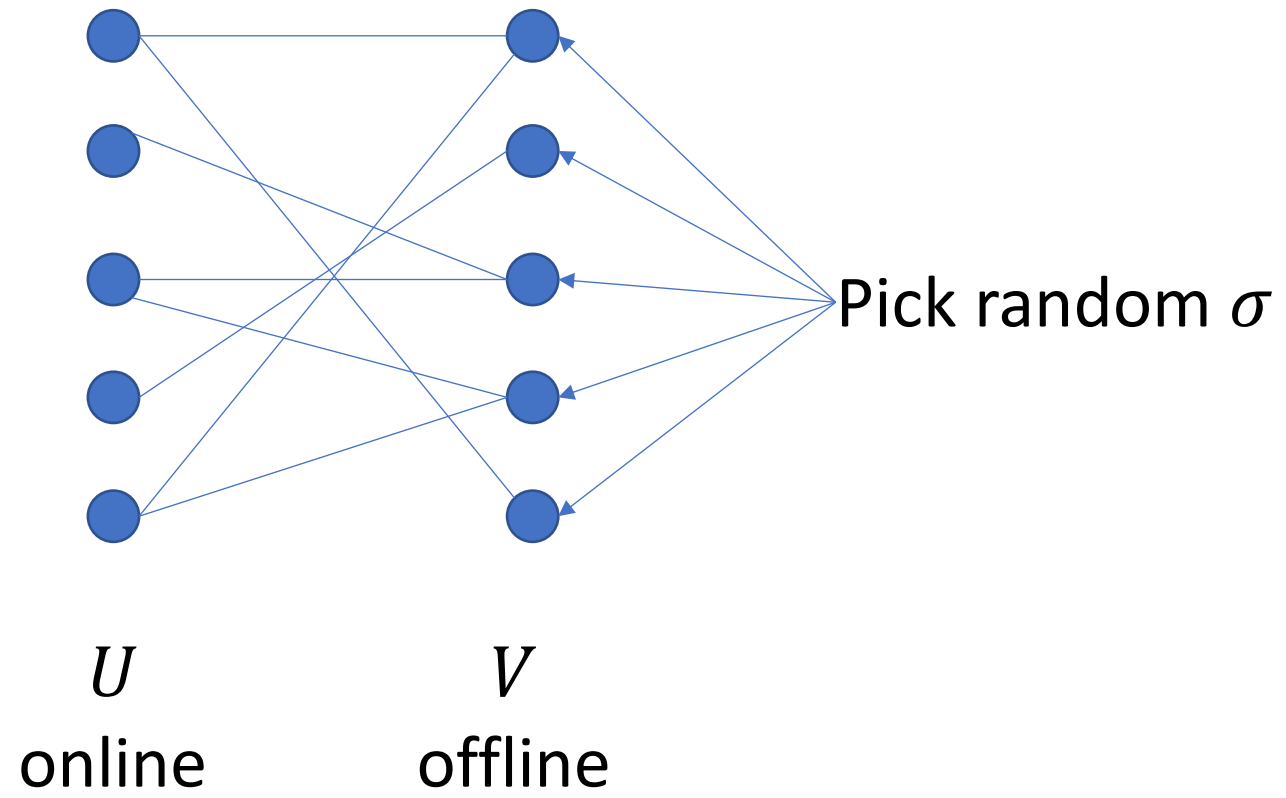
$v \leftarrow \arg \min \{ \sigma(v) : v \in N_c(u) \}$

$M \leftarrow M \cup \{(u_i, v)\}$

$\triangleright$  match  $u_i$  with  $v$

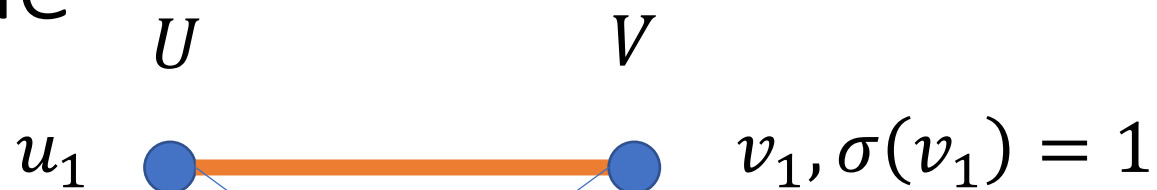
$i \leftarrow i + 1$

---



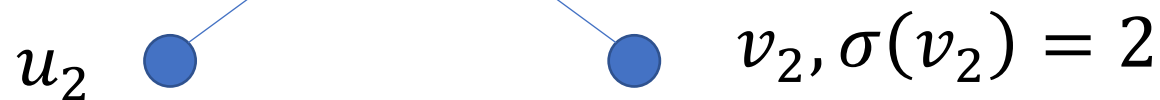
# Ranking example

$ALG = 1$



Probability  $1/2$

$OPT = 2$

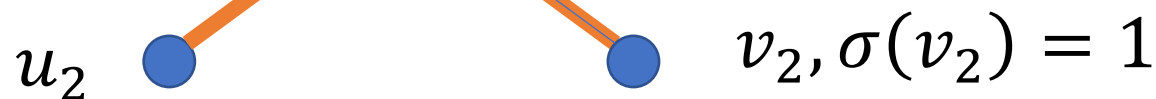


$ALG = 2$



Probability  $1/2$

$OPT = 2$





# Ranking example

$$ALG = 1 \qquad OPT = 2 \qquad \text{Probability } 1/2$$

$$ALG = 2 \qquad OPT = 2 \qquad \text{Probability } 1/2$$

$$\mathbb{E}(ALG) = \frac{3}{2}$$

$$\frac{OPT}{\mathbb{E}(ALG)} = \frac{4}{3} \approx 1.333$$

## Theorem

$$\rho_{OBL}(\textit{Ranking}) \leq \frac{e}{e-1} \approx 1.582$$

$G = (U, V, E)$  – input graph

Simplifying assumptions:

$|U| = |V| = n$  online and offline sides are of same size

let  $M^*$  denote some maximum matching, i.e.,  $|M^*| = OPT$

$|M^*| = |U|$ , i.e.,  $M^*$  is a perfect matching – it matches all nodes

Exercise: we can assume the above without loss of generality

We are interested in how close  $|M|$  is to  $n$

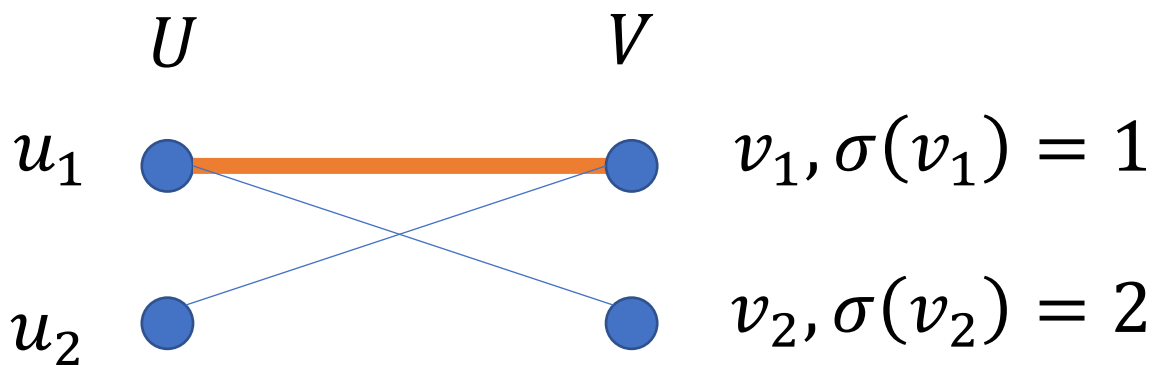
## Theorem

$$\rho_{OBL}(\textit{Ranking}) \leq \frac{e}{e-1} \approx 1.582$$

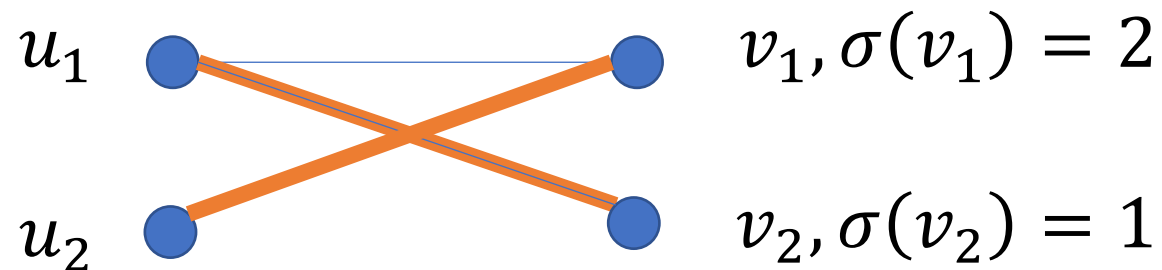
$p_t$  - probability that vertex of rank  $t$  is matched by *Ranking*

Example:

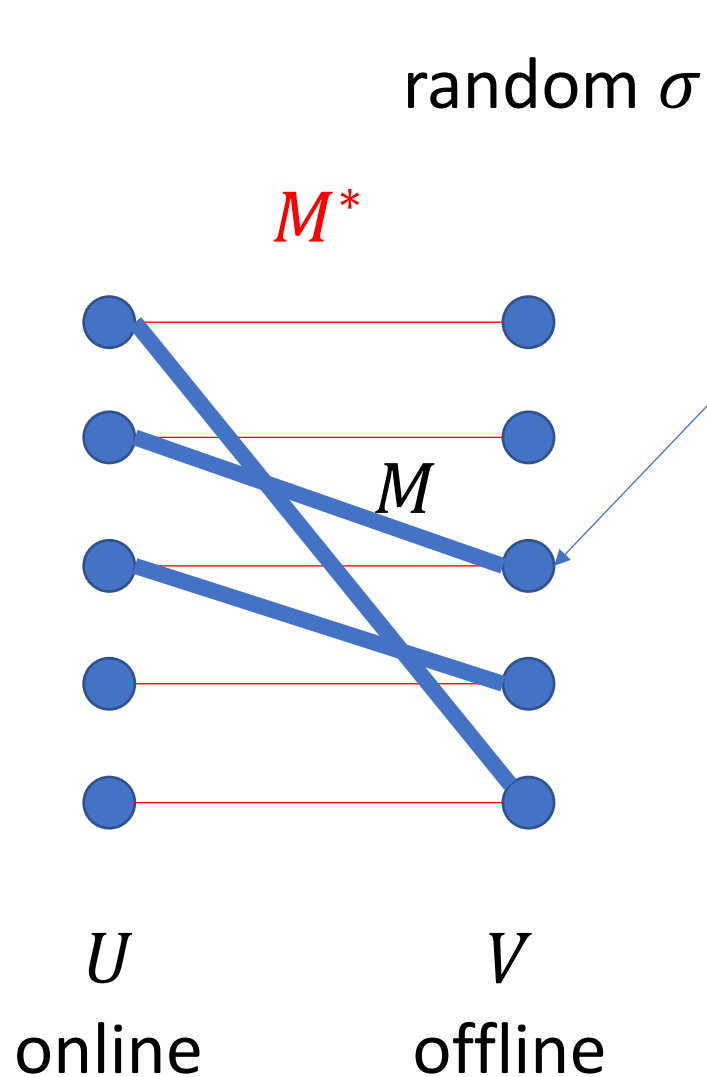
Probability 1/2



Probability 1/2



$$p_1 = ?$$
$$p_2 = ?$$



Vertex of rank  $t$

$p_t$  - probability it is matched

$J_t$  - indicator that vertex of rank  $t$  matched by *Ranking*

$J_t = 1$  if vertex of rank  $t$  is matched

$J_t = 0$  otherwise

$$|M| = \sum_t J_t$$

$$\mathbb{E}(|M|) = \mathbb{E}\left(\sum_t J_t\right)$$

$$= \sum_t \mathbb{E}(J_t) = \sum_t 1 \cdot p_t + 0 \cdot (1 - p_t) = \sum_t p_t$$

## Theorem

$$\rho_{OBL}(\textit{Ranking}) \leq \frac{e}{e-1} \approx 1.582$$

We want to show that  $\mathbb{E}(|M|)$  is large, i.e.,

$$\mathbb{E}(|M|) \geq \frac{e-1}{e}n = \left(1 - \frac{1}{e}\right)n$$

Alternatively

$$\sum_t p_t \geq \left(1 - \frac{1}{e}\right)n$$

**KEY LEMMA:**

For all  $t \in [n]$  we have  $1 - p_t \leq \frac{1}{n} \sum_{1 \leq s \leq t} p_s$

## Theorem

$$\rho_{OBL}(\textit{Ranking}) \leq \frac{e}{e-1} \approx 1.582$$

### KEY LEMMA:

For all  $t \in [n]$  we have  $1 - p_t \leq \frac{1}{n} \sum_{1 \leq s \leq t} p_s$

Assume KEY LEMMA and see how it implies the theorem

$p_1 = 1$  since there exists a perfect matching

$p_t \geq \left(1 - \frac{1}{n}\right) \left(\frac{n}{n+1}\right)^{t-1}$  follows by KEY LEMMA and simple induction

## Theorem

$$\rho_{OBL}(\textit{Ranking}) \leq \frac{e}{e-1} \approx 1.582$$

$$p_1 = 1$$

$$p_t \geq \left(1 - \frac{1}{n}\right) \left(\frac{n}{n+1}\right)^{t-1}$$

$$\begin{aligned} \sum_t p_t &\geq \frac{1}{n} + \left(1 - \frac{1}{n}\right) \sum_{1 \leq t \leq n} \left(\frac{n}{n+1}\right)^{t-1} \geq \frac{1}{n} + \left(1 - \frac{1}{n}\right) \frac{1 - \left(\frac{n}{n+1}\right)^n}{1 - \frac{n}{n+1}} \\ &= \frac{1}{n} + \frac{n^2 - 1}{n} \left(1 - \left(\frac{n}{n+1}\right)^n\right) \geq n \left(1 - \left(1 - \frac{1}{n+1}\right)^n\right) \rightarrow n \left(1 - \frac{1}{e}\right) \end{aligned}$$

## Theorem

$$\rho_{OBL}(\textit{Ranking}) \leq \frac{e}{e-1} \approx 1.582$$

To finish the proof we need to prove

### KEY LEMMA:

For all  $t \in [n]$  we have  $1 - p_t \leq \frac{1}{n} \sum_{1 \leq s \leq t} p_s$

$A_s$  - the set of permutations  $\sigma$  such that *Ranking* **matches** a vertex of rank  $s$

$B_t$  - the set of permutation  $\sigma$  such that *Ranking* **doesn't match** a vertex of rank  $t$



## Theorem

$$\rho_{OBL}(\textit{Ranking}) \leq \frac{e}{e-1} \approx 1.582$$

### KEY LEMMA:

For all  $t \in [n]$  we have  $1 - p_t \leq \frac{1}{n} \sum_{1 \leq s \leq t} p_s$

We construct an injection

$$[n] \times B_t \rightarrow \bigcup_{1 \leq s \leq t} A_s$$

Then  $n|B_t| \leq \sum_{1 \leq s \leq t} |A_s|$

$$\text{So } \frac{|B_t|}{n!} \leq \frac{1}{n} \sum_{1 \leq s \leq t} \frac{|A_s|}{n!}$$

Which is equivalent to  $1 - p_t \leq \frac{1}{n} \sum_{1 \leq s \leq t} p_s$

## Theorem

$$\rho_{OBL}(\textit{Ranking}) \leq \frac{e}{e-1} \approx 1.582$$

Injection

$$[n] \times B_t \rightarrow \bigcup_{1 \leq s \leq t} A_s$$

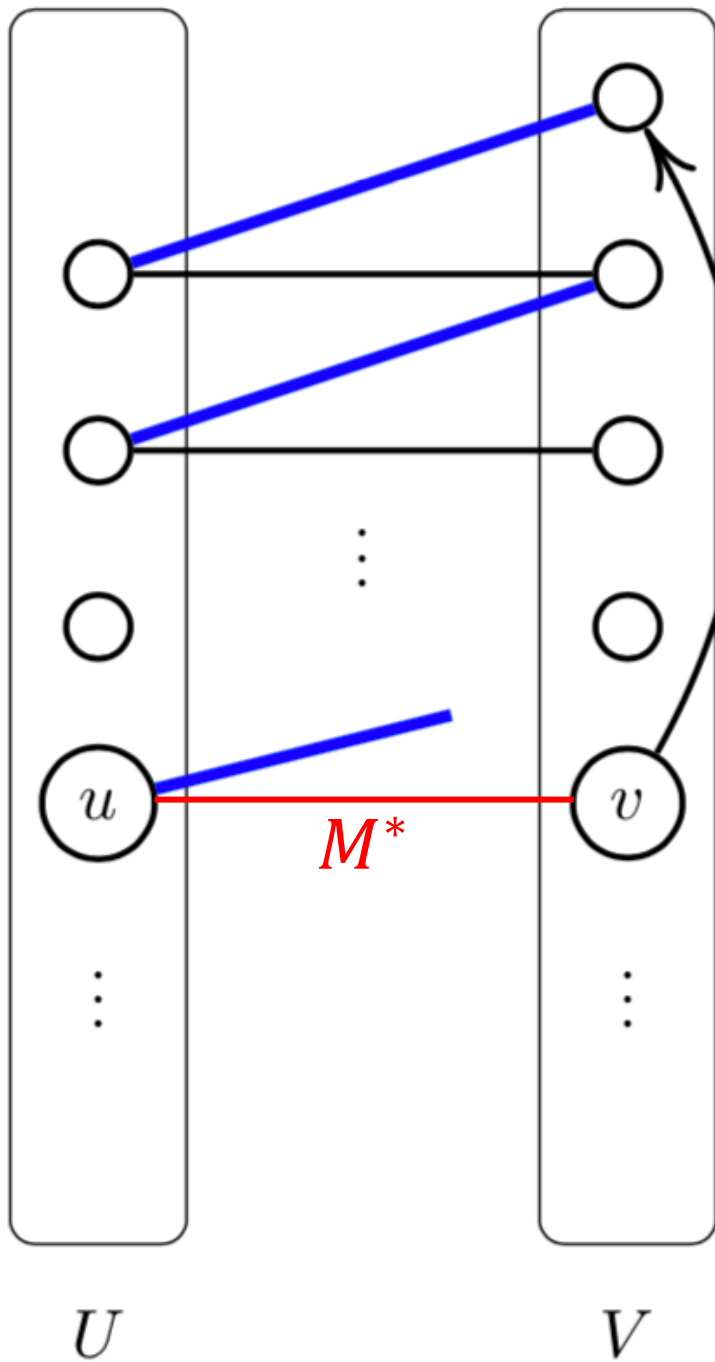
Means that we map an index  $i \in [n]$  and a permutation where a vertex of rank  $t$  is not matched to some permutation where a vertex of rank  $\leq t$  is matched.

$\sigma \in B_t$  and  $i \in [n]$

$v = \sigma^{-1}(t)$  – vertex of rank  $t$

Obtain  $\sigma_i$  from  $\sigma$  by moving  $v$  into position  $i$  and shifting other elements accordingly

It is clearly injective, but is it well defined?



$u$  gets matched to some vertex of rank  $s \leq t$  no matter where  $v$  gets moved

moving  $v$  to a better rank

rank  $t$



Ranking after moving  $v$



Ranking before moving  $v$

This finishes the proof of the upper bound.

We also have a matching lower bound that holds for all randomized algorithms

Theorem

Let  $ALG$  be a randomized algorithm for Maximum Bipartite Matching. Then

$$\rho_{OBL}(ALG) \geq \frac{e}{e-1} \approx 1.582$$

## Theorem

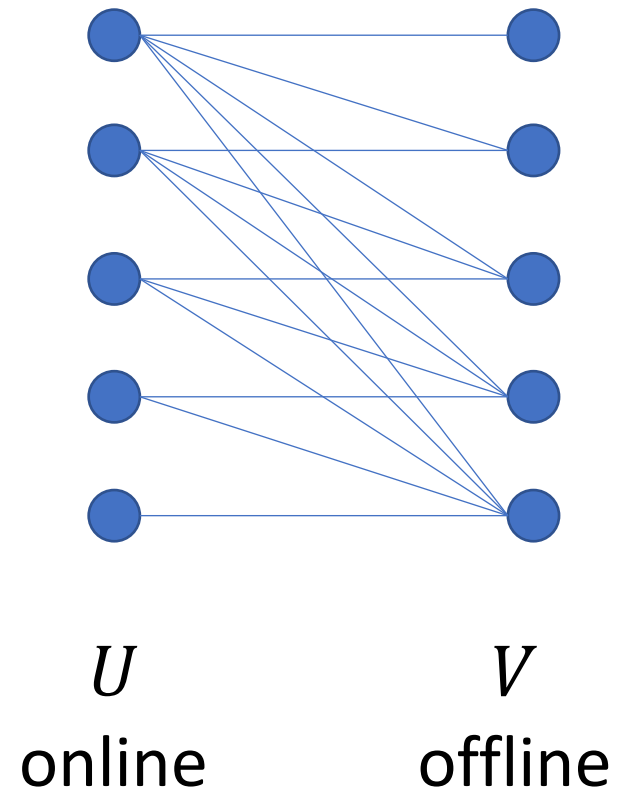
Let  $ALG$  be a randomized algorithm for Maximum Bipartite Matching. Then

$$\rho_{OBL}(ALG) \geq \frac{e}{e-1} \approx 1.582$$

## Proof:

Use Yao's minimax argument

Consider a triangle graph



## Theorem

Let  $ALG$  be a randomized algorithm for Maximum Bipartite Matching. Then

$$\rho_{OBL}(ALG) \geq \frac{e}{e-1} \approx 1.582$$

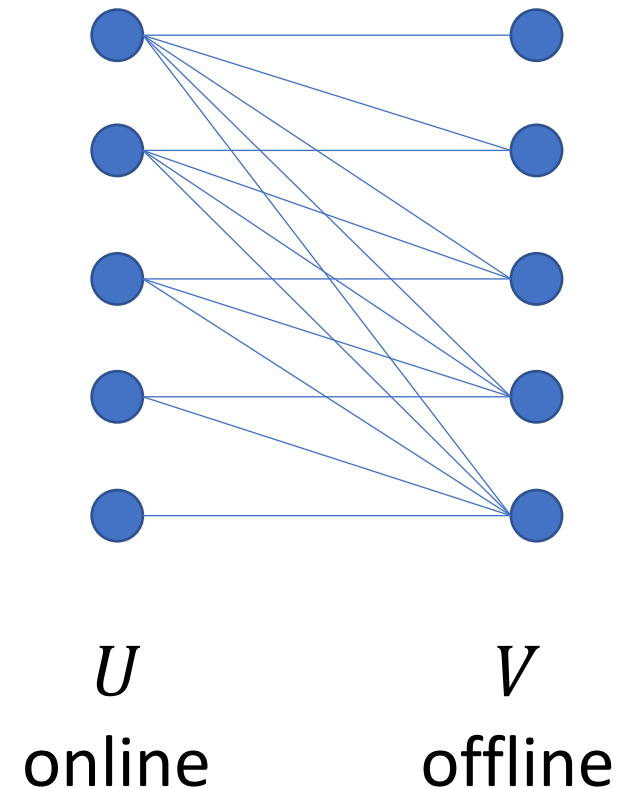
### Proof:

Use Yao's minimax argument

Consider a triangle graph

Generate distribution on inputs by randomly reordering  $V$

Can show that  $ALG$  will miss  $\frac{1}{e}$  fraction on average. See text for details.

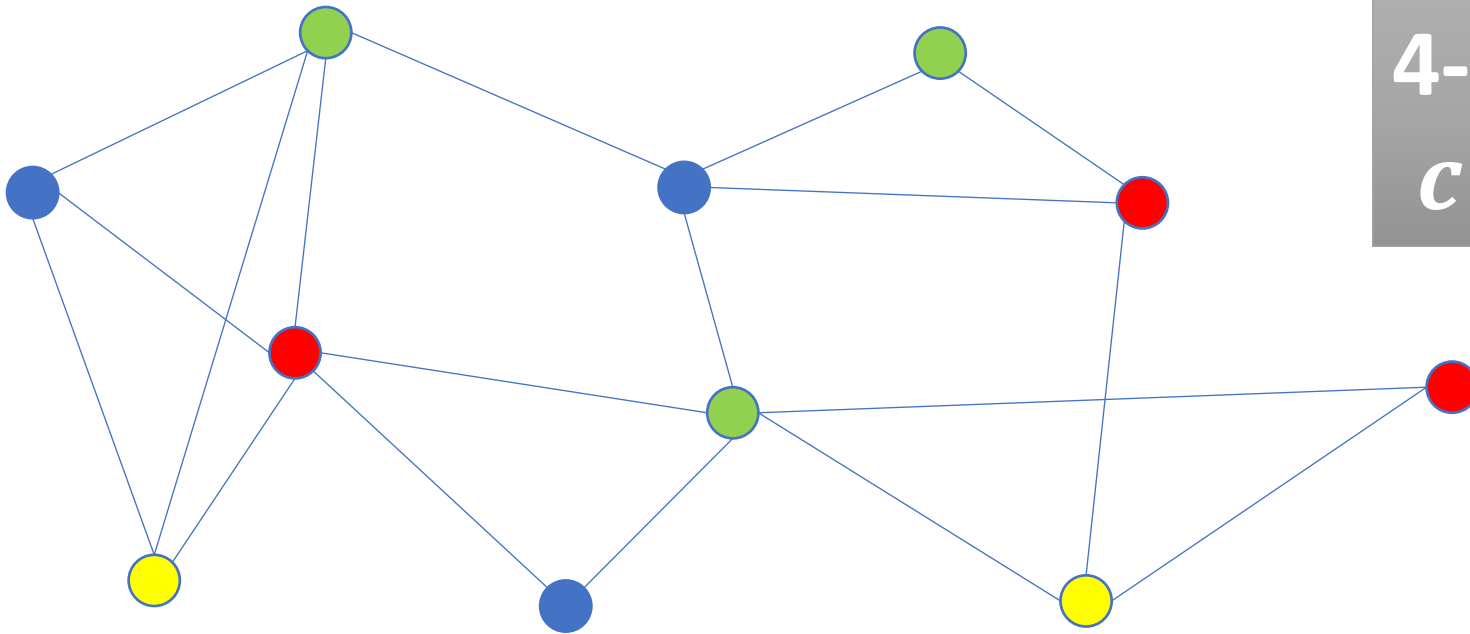




# Graph Coloring

# Graph Coloring

Graph  $k$ -coloring is a function  $c : V \rightarrow \{1, 2, \dots, k\}$  such that for all  $\{u, v\} \in E$  we have  $c(u) \neq c(v)$



4-coloring

$c : V \rightarrow \{1, 2, 3, 4\}$



# Graph Coloring

- Input:**  $G = (V, E, <)$  unweighted undirected graph  
 $<$  is a total order on  $V$   
 $(v_1, N_1), (v_2, N_2), \dots, (v_n, N_n)$  input sequence, where  
 $v_1 < v_2 < \dots < v_n$  and  
 $N_i = N(v_i) \cap \{v_j : j < i\}$
- Output:**  $c : V \rightarrow [k]$  where  $c(v_i)$  indicates the color assigned to vertex  $v_i$
- Objective:** to find  $c$  so as to minimize  $k$  subject to  $c$  being a valid coloring, i.e.,  $\forall \{u, v\} \in E$  we have  $c(u) \neq c(v)$

# Graph Coloring

We are interested in graph coloring of bipartite graphs in VAM-PH

Thus,  $OPT = 2$

Theorem

Let  $ALG$  be a deterministic online algorithm for Graph Coloring of bipartite graphs. Then

$$\rho(ALG) \geq \frac{\log n}{2}$$

Theorem

There is a deterministic online algorithm  $CBIP$  for Graph Coloring of bipartite graphs such that

$$\rho(CBIP) \leq \log n$$

## Theorem

Let  $ALG$  be a deterministic online algorithm for Graph Coloring of bipartite graphs. Then

$$\rho(ALG) \geq \frac{\log n}{2}$$

## Proof:

Fix  $k \in \mathbb{N}$ . Define the following statement  $S(k)$ :

There is an adversarial strategy that presents a sequence of trees

$T_1, \dots, T_k$  such that  $ALG$  uses  $k$  distinct colors

and the combined size of all trees is  $\leq 2^k - 1$

We prove  $S(k)$  by induction on  $k$

$S(k)$ : there is an adversarial strategy that presents a sequence of trees  $T_1, \dots, T_k$  such that  $ALG$  uses  $k$  distinct colors and the combined size of all trees is  $\leq 2^k - 1$

Case  $k = 1$ :



$T_1$  - one tree

One color

Combined size =  $1 = 2^1 - 1$

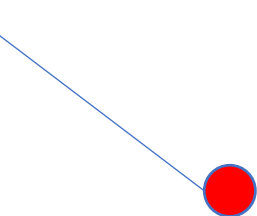
Case  $k = 2$ :



$T_1$



$T_2$



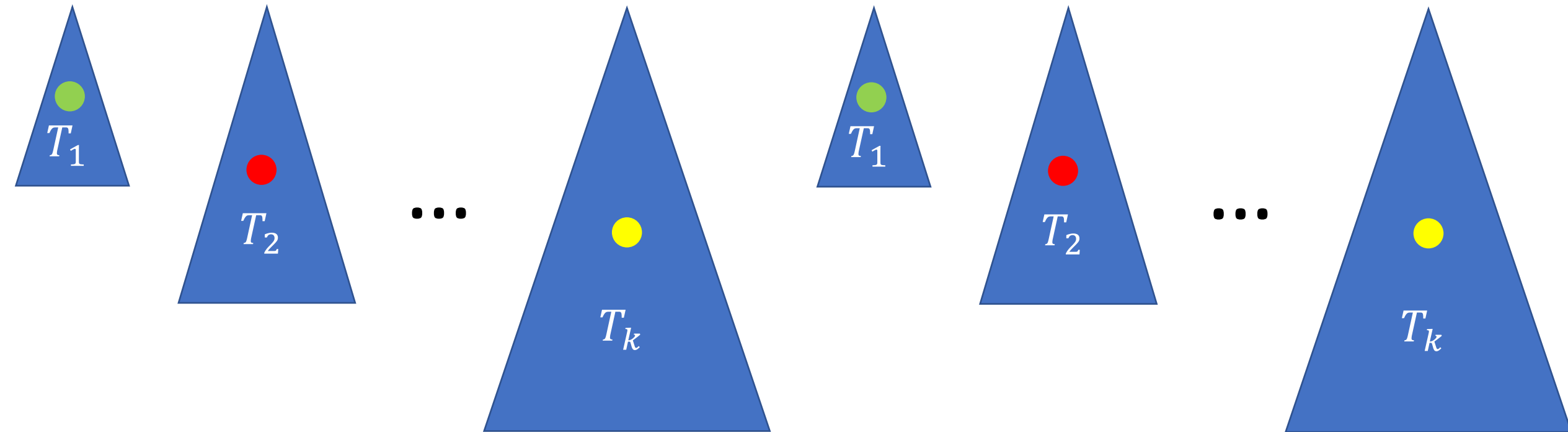
Two trees

Two colors

Combined size =  $3 = 2^2 - 1$

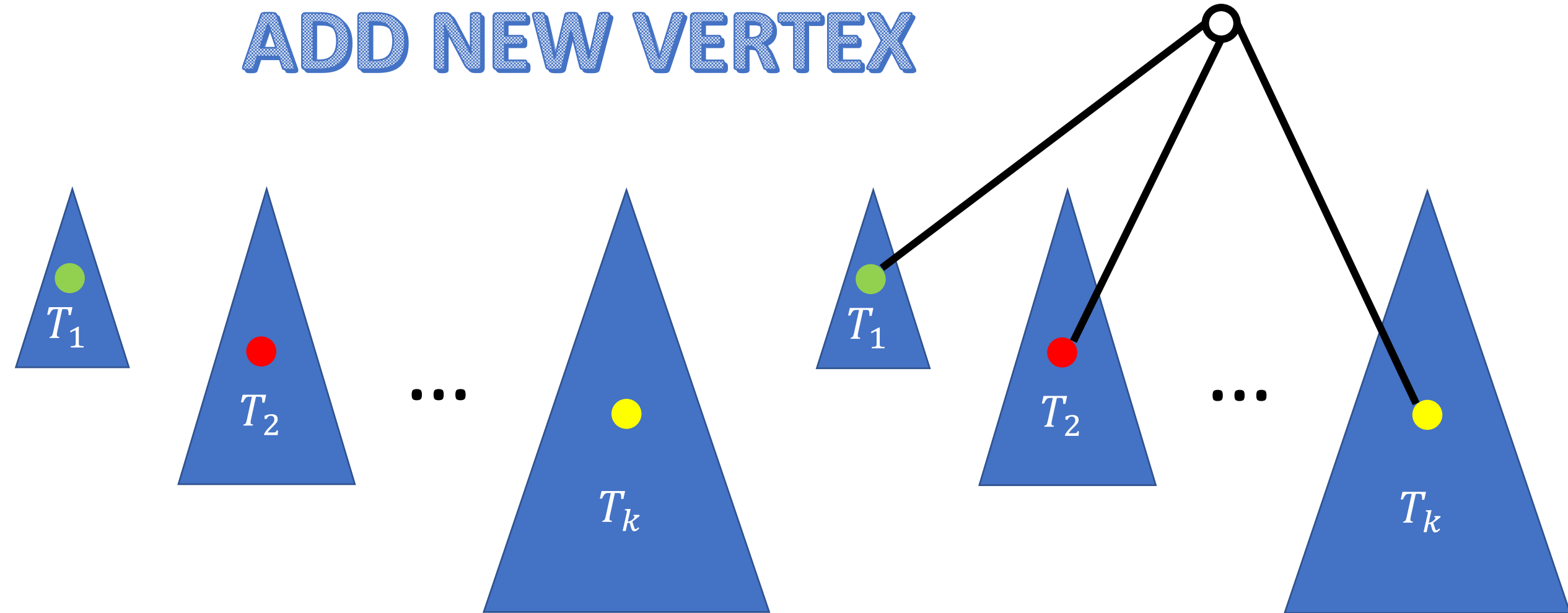
By induction:  $T_1, \dots, T_k$  -  $k$  trees,  $k$  colors, combined size  $\leq 2^k - 1$

**DUPLICATE!**



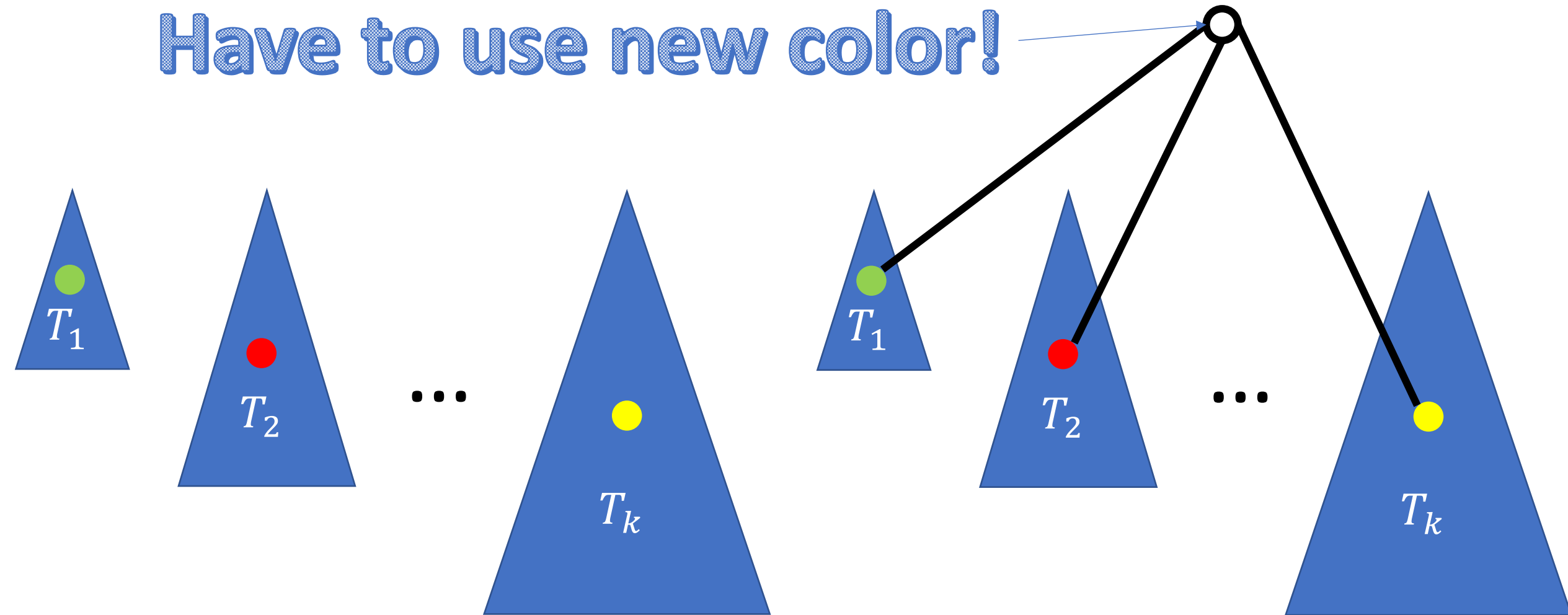
By induction:  $T_1, \dots, T_k$  -  $k$  trees,  $k$  colors, combined size  $\leq 2^k - 1$

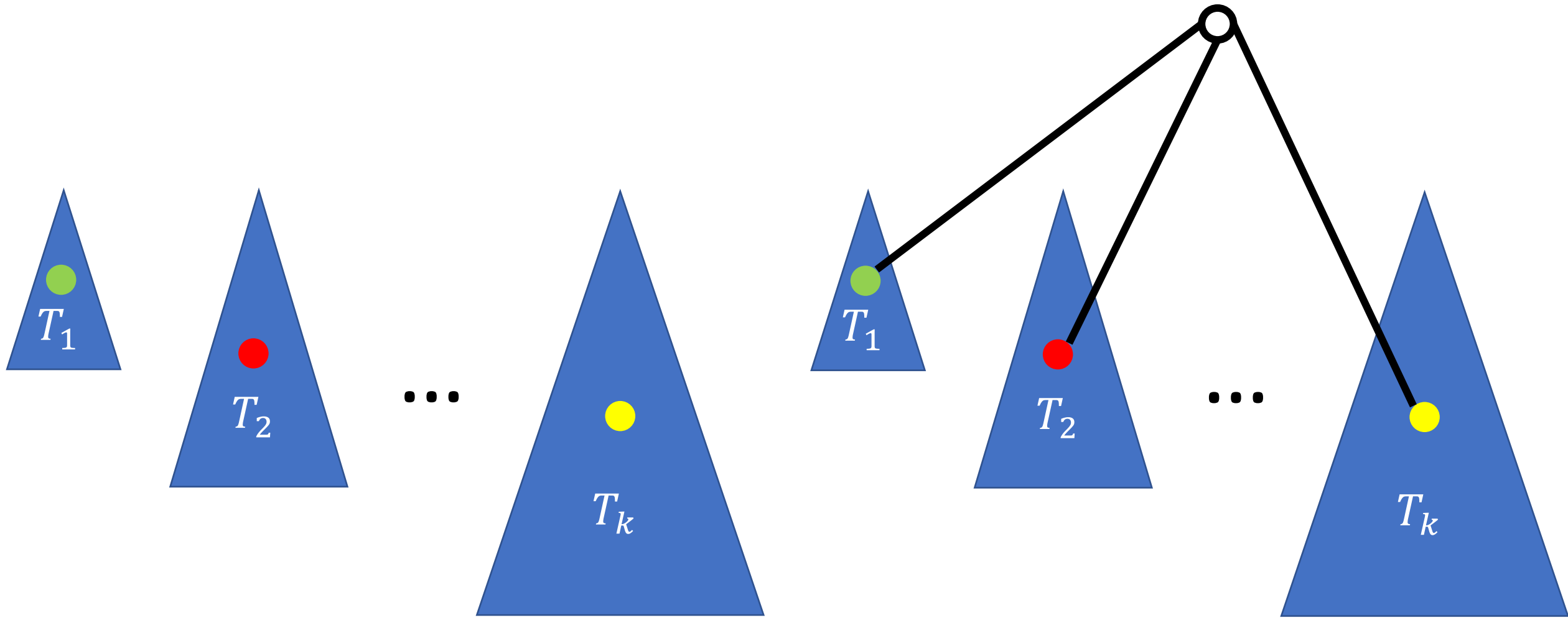
ADD NEW VERTEX



By induction:  $T_1, \dots, T_k$  -  $k$  trees,  $k$  colors, combined size  $\leq 2^k - 1$

Have to use new color!





Get  $T_1, \dots, T_k, T_{k+1}$  -  $(k + 1)$  trees,  $k + 1$  colors,  
 combined size  $\leq (2^k - 1) + (2^k - 1) + 1 = 2^{k+1} - 1$



## Theorem

Let  $ALG$  be a deterministic online algorithm for Graph Coloring of bipartite graphs. Then

$$\rho(ALG) \geq \frac{\log n}{2}$$

Take  $k = \log n$

Then adversary presents input  $T_1, \dots, T_k$  trees of combined size  $2^k - 1 = n - 1$

Trees are bipartite so  $OPT = 2$

$ALG$  uses  $k = \log n$  colors

QED

# *CBIP* algorithm

When a vertex  $v$  arrives

compute  $C_v$  - connected component of  $v$

compute bipartition of  $C_v = S_v \cup \widetilde{S}_v$

$$v \in S_v \text{ and } N(v) \subseteq \widetilde{S}_v$$

let  $i$  be the smallest color **not** in  $\widetilde{S}_v$

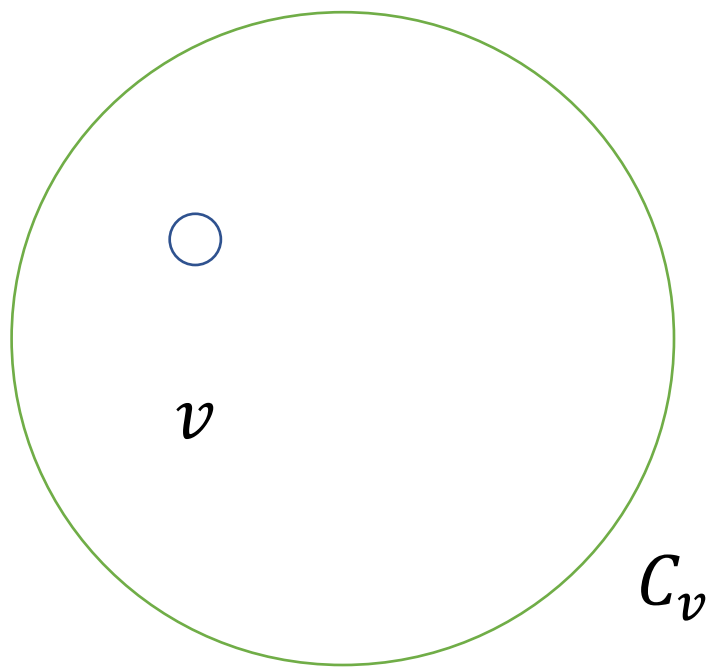
color  $v$  with  $i$

# *CBIP* example

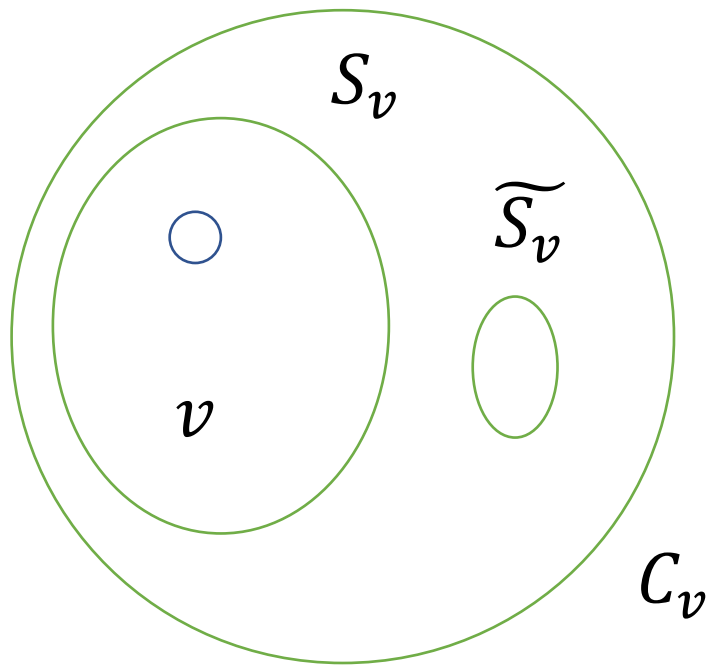


$v$

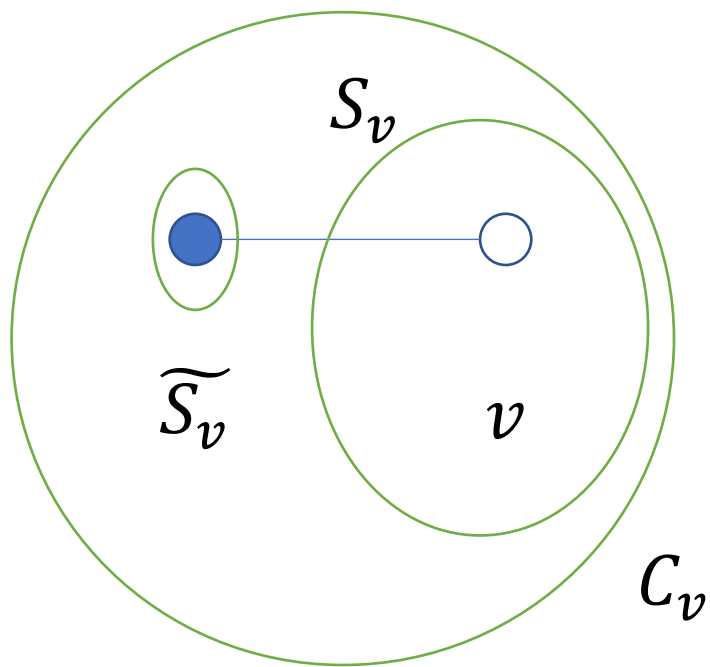
# *CBIP* example



# *CBIP* example



# *CBIP* example

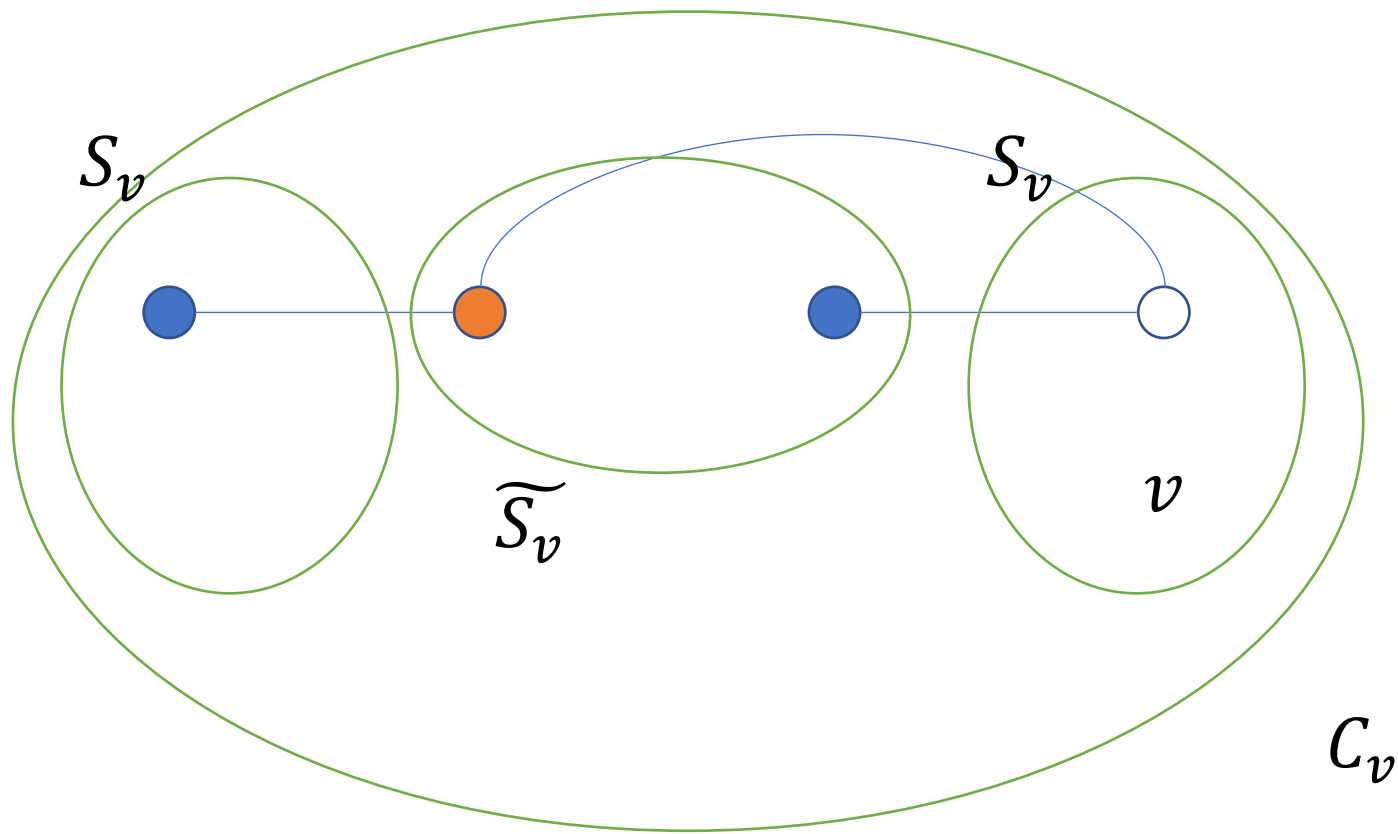


# *CBIP* example



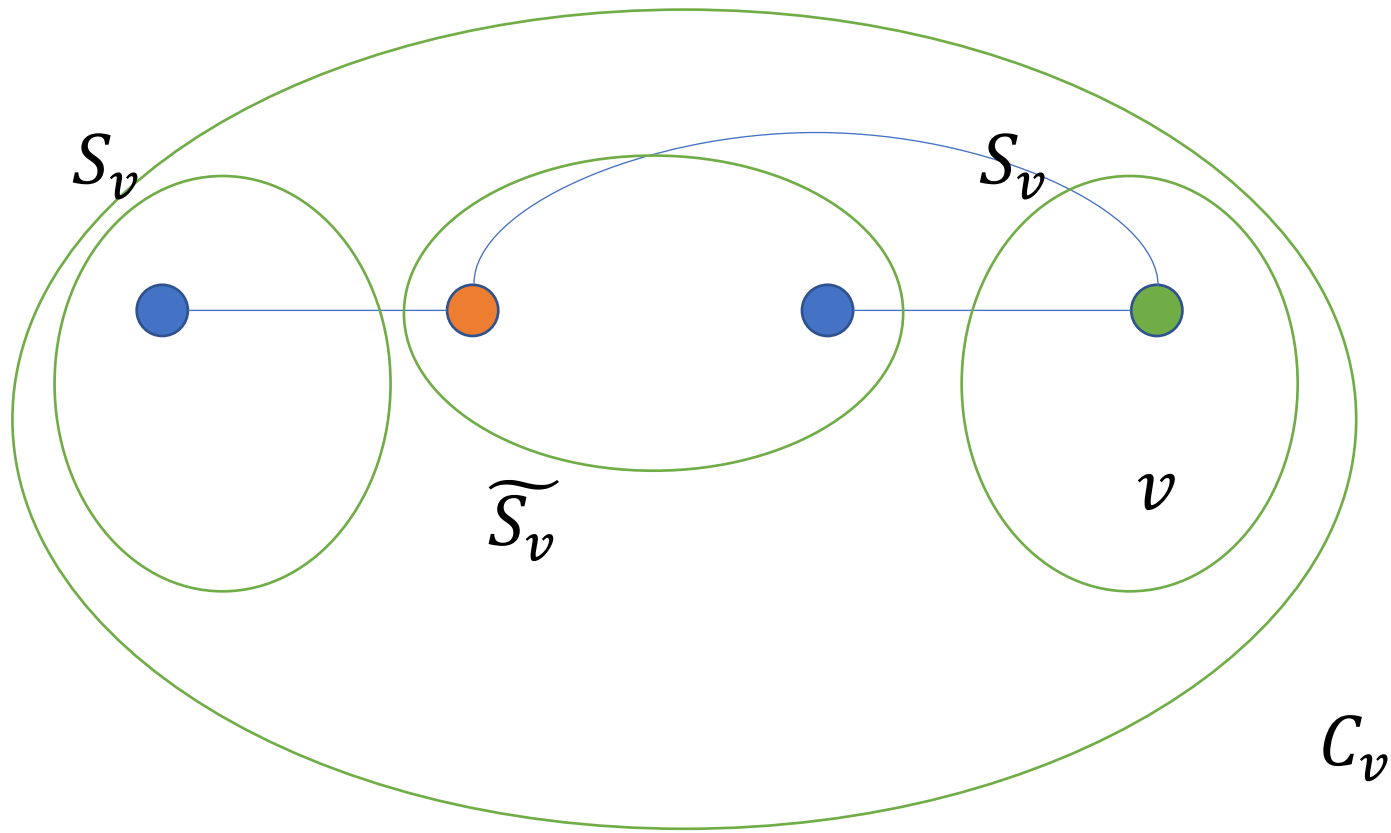
$v$

# $CBIP$ example





# $CBIP$ example



## Theorem

$$\rho(CBIP) \leq \log n$$

## Proof

$n(i)$  – minimum number of nodes needed to force  $CBIP$  use  $i$  distinct colors

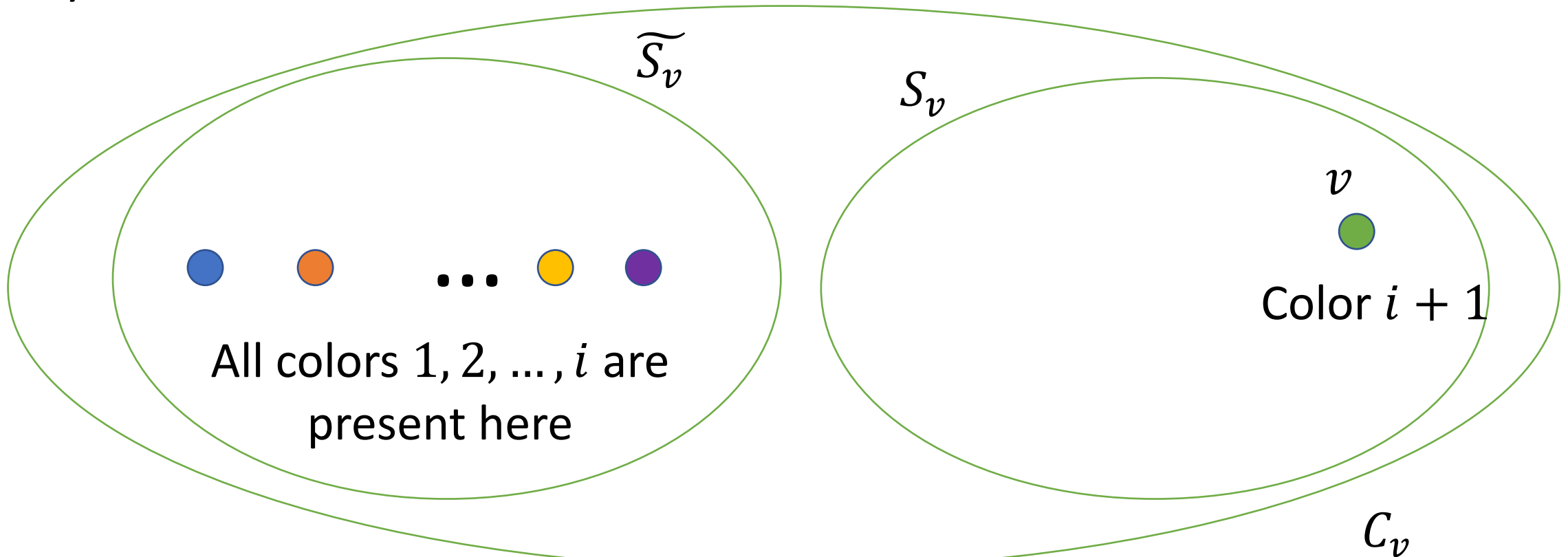
We will prove  $n(i) \geq \lceil 2^{i/2} \rceil$  by induction on  $i$

Clearly  $n(1) = 1$  and  $n(2) = 2$

## Theorem

$$\rho(CBIP) \leq \log n$$

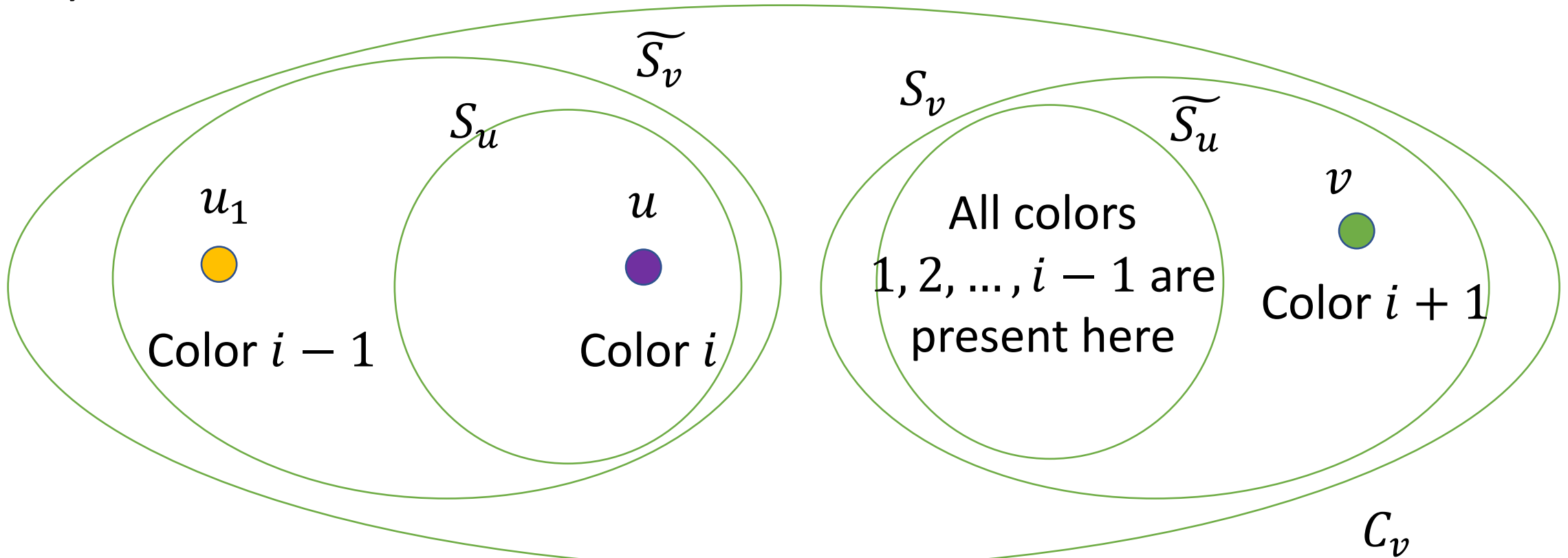
Inductive step: let  $v$  be the first vertex that is colored with color  $i + 1$  by CBIP



# Theorem

$$\rho(CBIP) \leq \log n$$

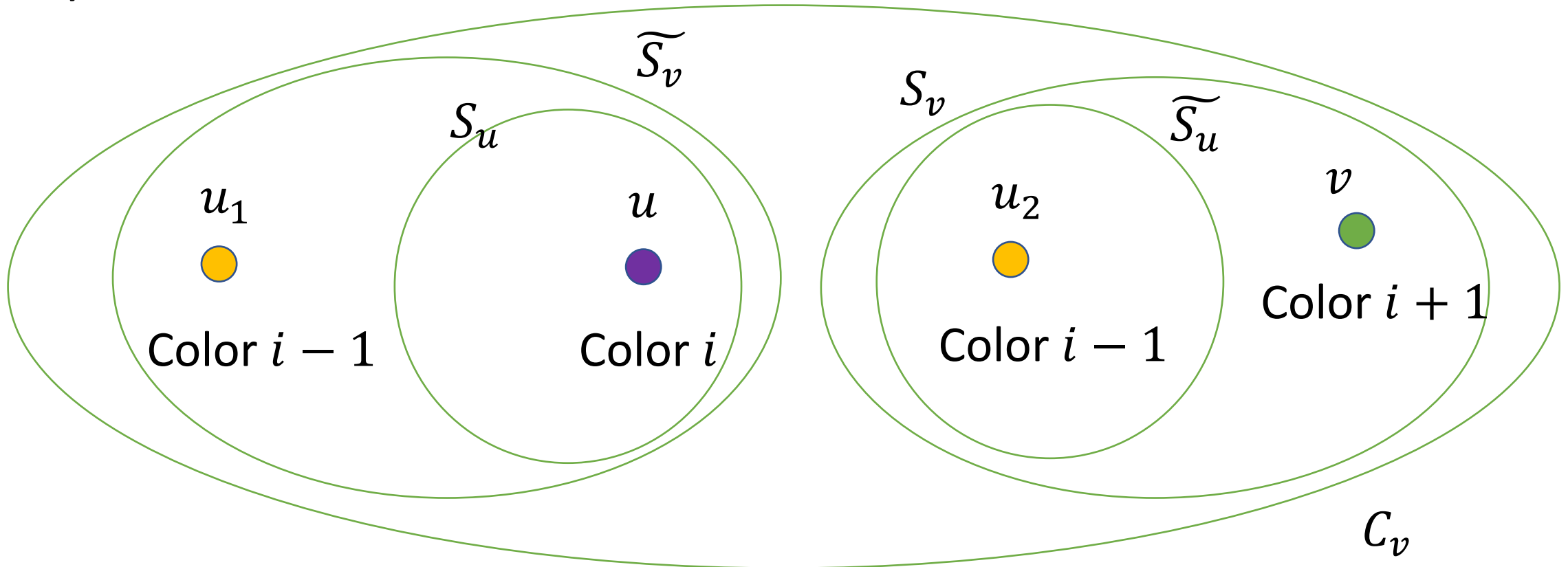
Inductive step: let  $v$  be the first vertex that is colored with color  $i + 1$  by CBIP



# Theorem

$$\rho(CBIP) \leq \log n$$

Inductive step: let  $v$  be the first vertex that is colored with color  $i + 1$  by CBIP

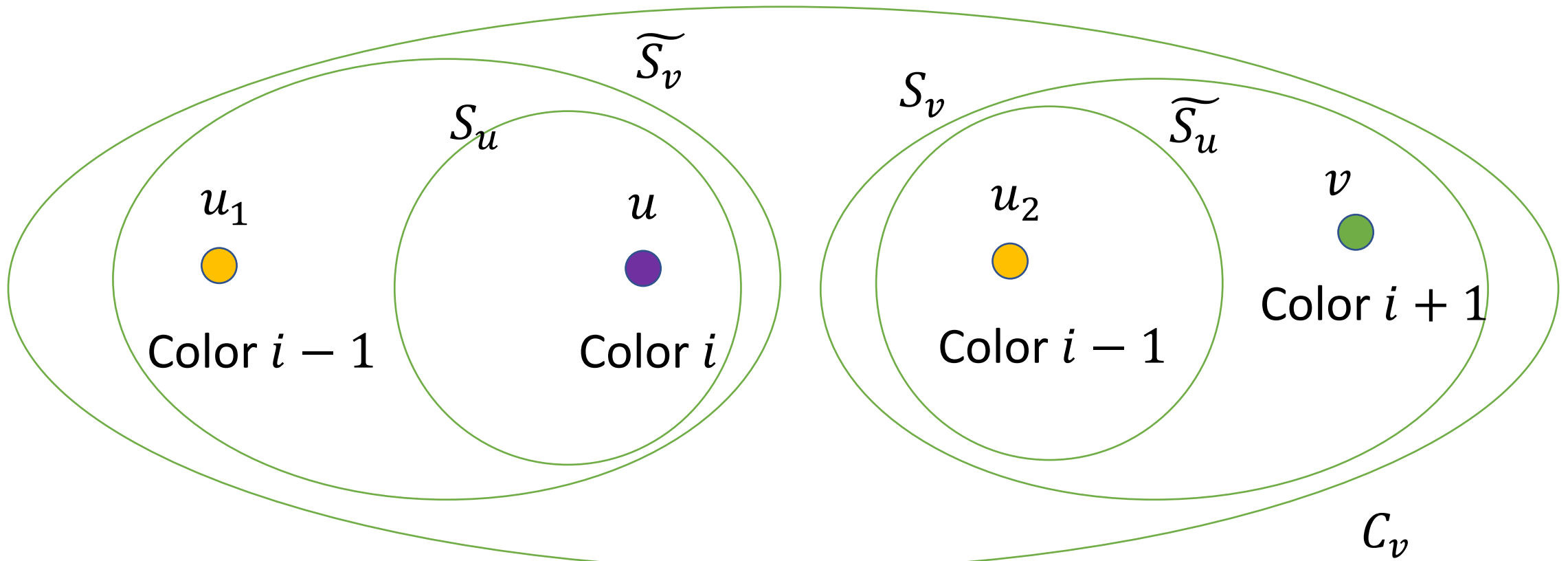


# Theorem

$$\rho(CBIP) \leq \log n$$

Found two vertices  $u_1$  and  $u_2$  colored with  $i - 1$

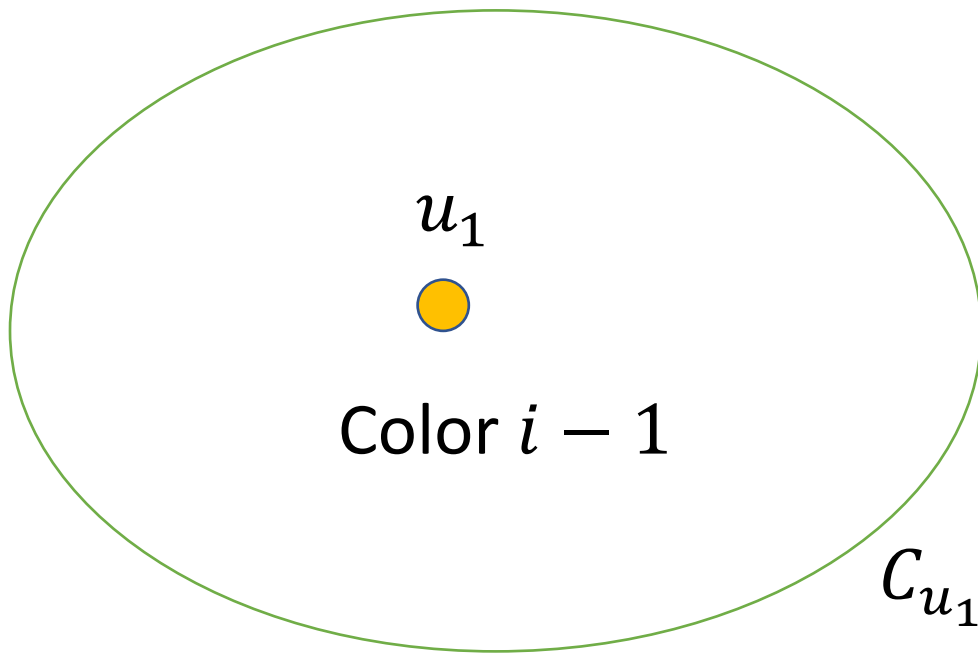
$$C_{u_1} \cap C_{u_2} = \emptyset$$



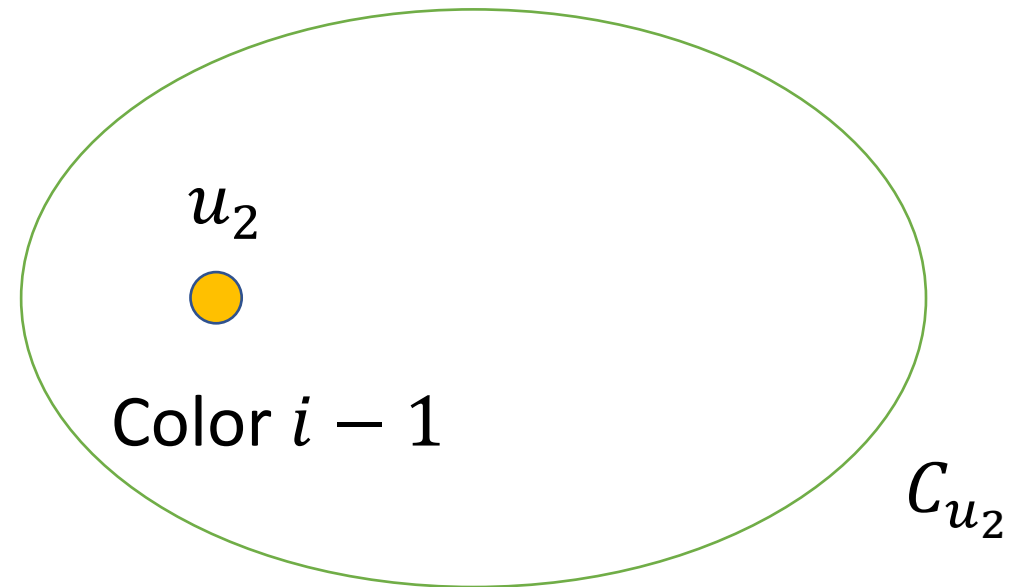
## Theorem

$$\rho(CBIP) \leq \log n$$

Found two vertices  $u_1$  and  $u_2$  colored with  $i - 1$   
 $C_{u_1} \cap C_{u_2} = \emptyset$



By induction  $|C_{u_1}| \geq \lceil 2^{(i-1)/2} \rceil$



By induction  $|C_{u_2}| \geq \lceil 2^{(i-1)/2} \rceil$

## Theorem

$$\rho(CBIP) \leq \log n$$

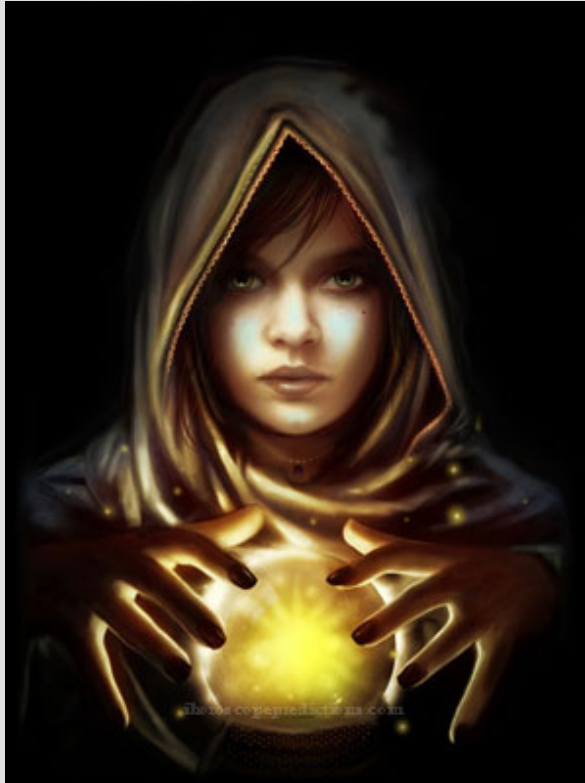
Therefore we have  $n(i + 1) \geq |C_{u_1}| + |C_{u_2}| \geq 2 \lceil 2^{(i-1)/2} \rceil \geq \lceil 2^{(i+1)/2} \rceil$

Set  $i = 2 \log n$  then we get  $n(i) = n$

Hence we get that on all bipartite graphs of size  $n$   $CBIP$  uses  $\leq 2 \log n$  colors, while  $OPT = 2$

QED





# Advice Complexity

# Online Algorithms with Advice

In practice, algorithm designers often have side knowledge about input

Example:

- Shipping company packs boxes of items into large containers
- Based on past, collects statistics of how many boxes and of roughly what size to expect each month



# Other Examples

## Time-Series Search:

the bounds  $L$  and  $U$  can be considered advice

knowing  $L$  and  $U$  allows to design better algorithms

## Graph problems:

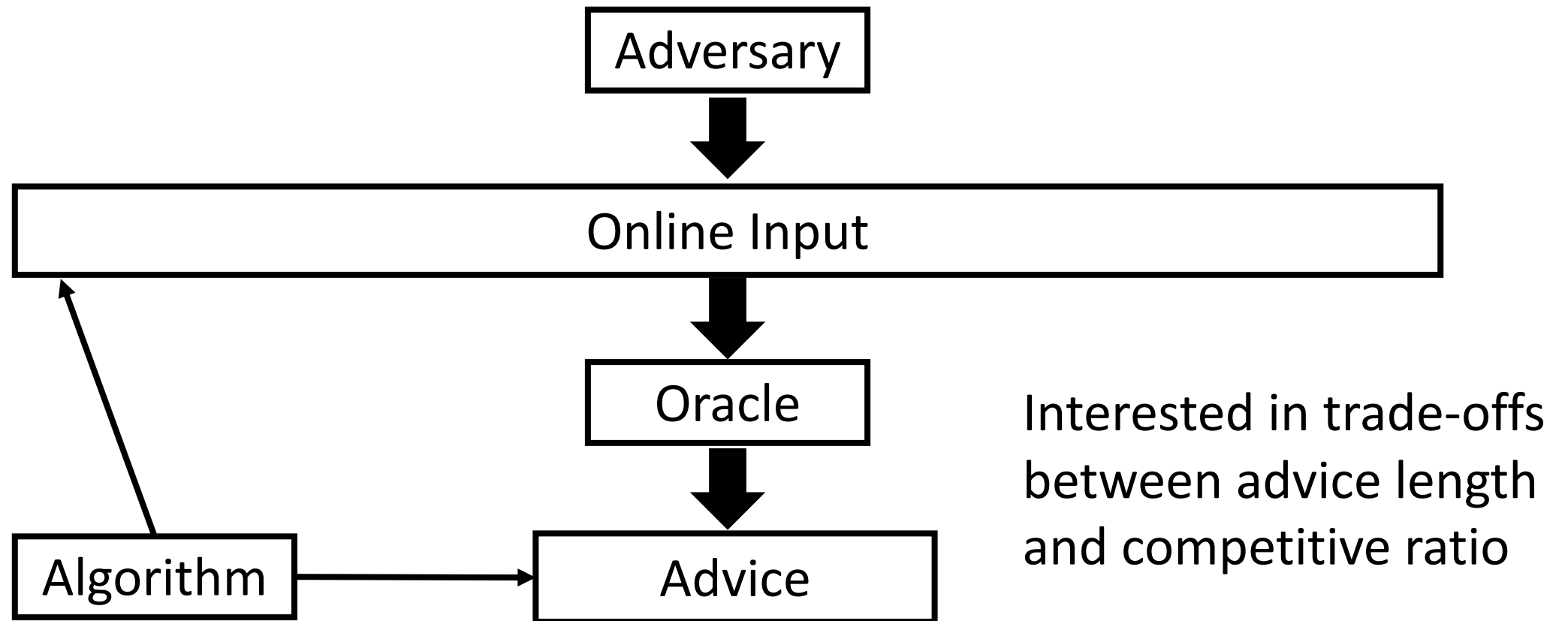
you sometimes know  $n$ , or  $m$ , or a bound on maximum degree

## Scheduling problems:

knowing the value of  $OPT$  (but not its decisions) can improve competitive ratio

# Online algorithms with advice

Model this side-knowledge information-theoretically with advice



# Advice models notes

3 players: algorithm, adversary, oracle

Oracle and algorithm cooperate and trust each other

Oracle is all powerful and sees the entire input

Oracle compresses some relevant information about input into short advice

If you fix advice to a specific string you get a deterministic algorithm

# Two advice models: Per Request and Tape

## **Per Request** model:

oracle fixes universe of answers  $U$

adversary presents  $x_i$

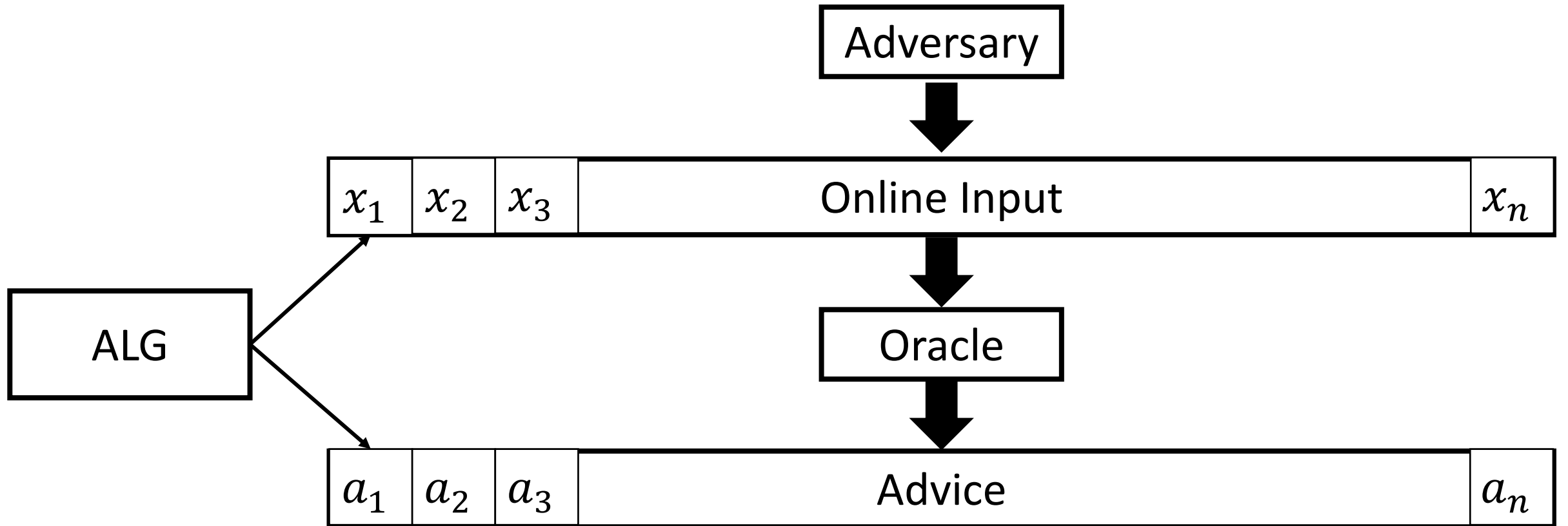
oracle presents  $a_i \in U$

algorithm makes a decision  $d_i$  based on all previous  $x_j$  and  $a_j$

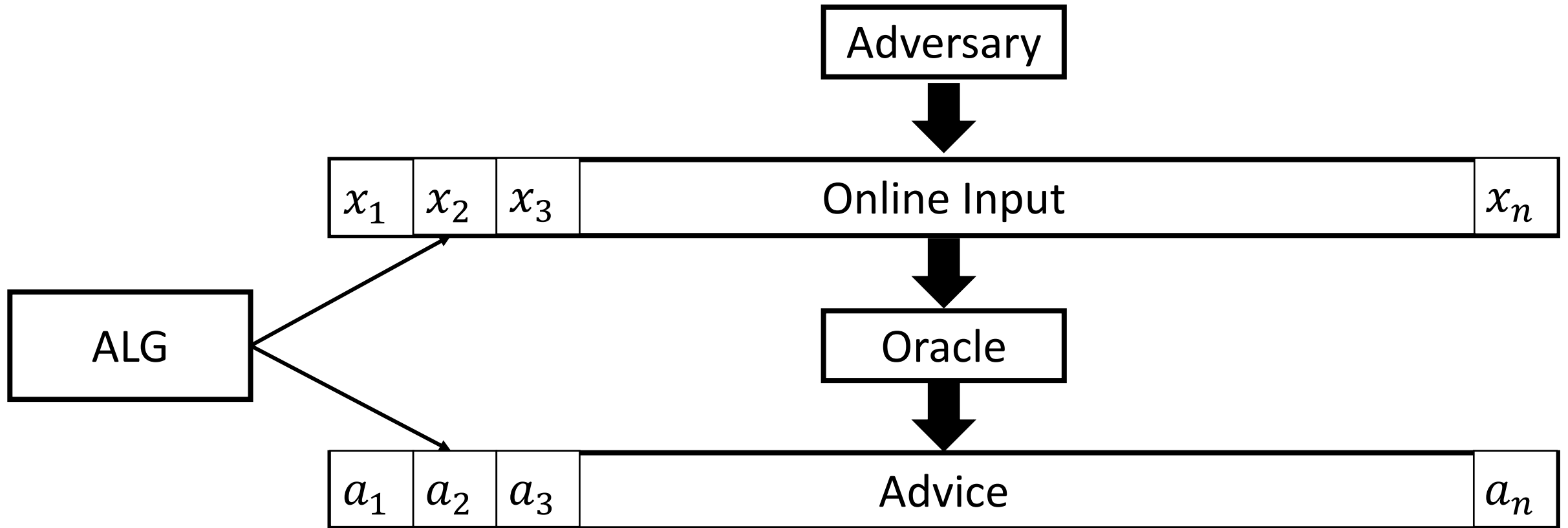
**Advice length:**  $\lceil \log |U| \rceil n$

Note that advice length is necessarily  $\Omega(n)$  in this model

# Per request model

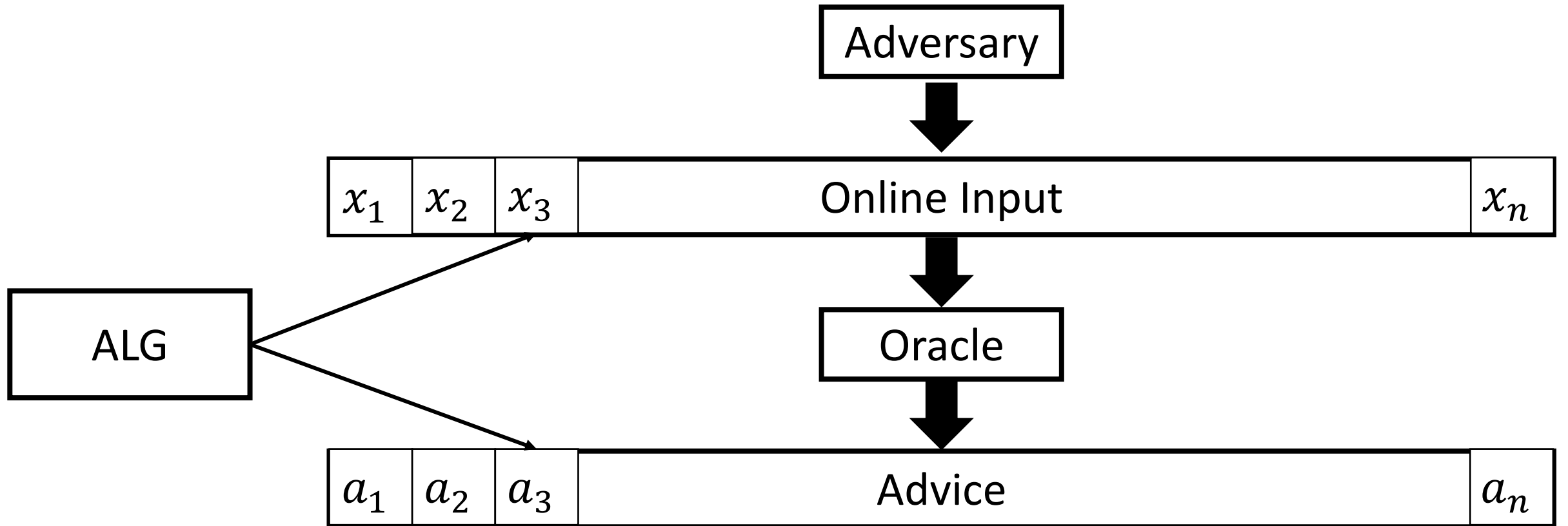


# Per request model

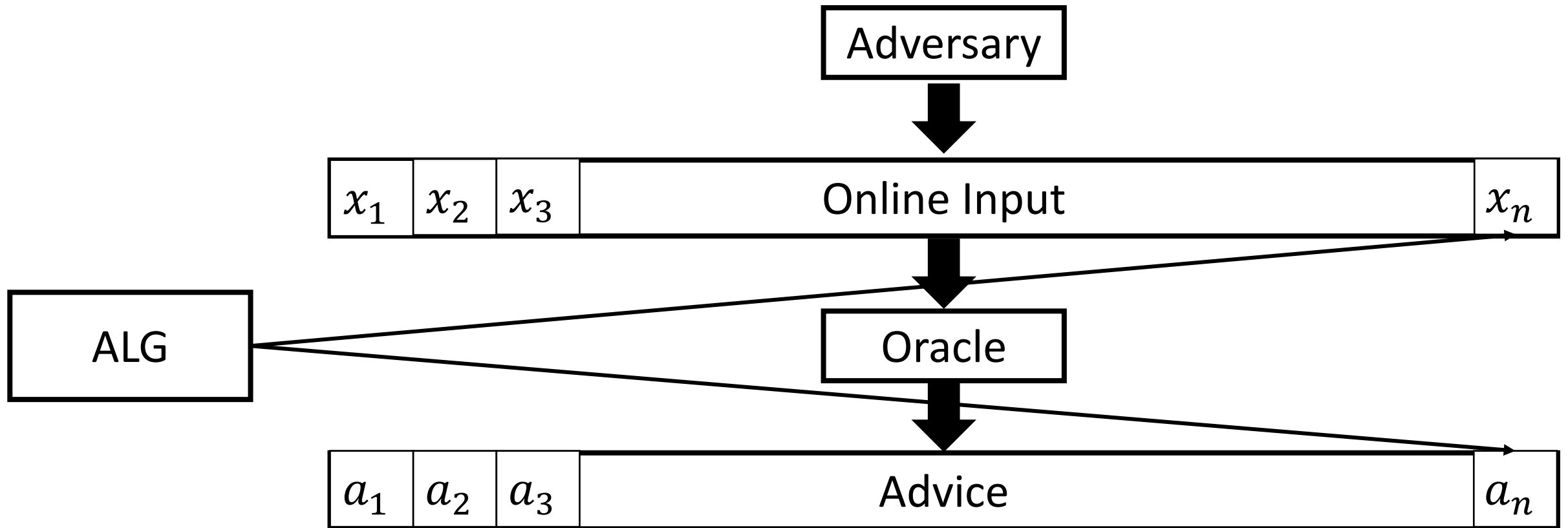




# Per request model



# Per request model



# Two advice models: Per Request and Tape

## **Tape** model:

- oracle writes advice on infinite tape

- adversary presents input sequence  $x_1, \dots, x_n$

- algorithm decides how many bits of advice it wants to read

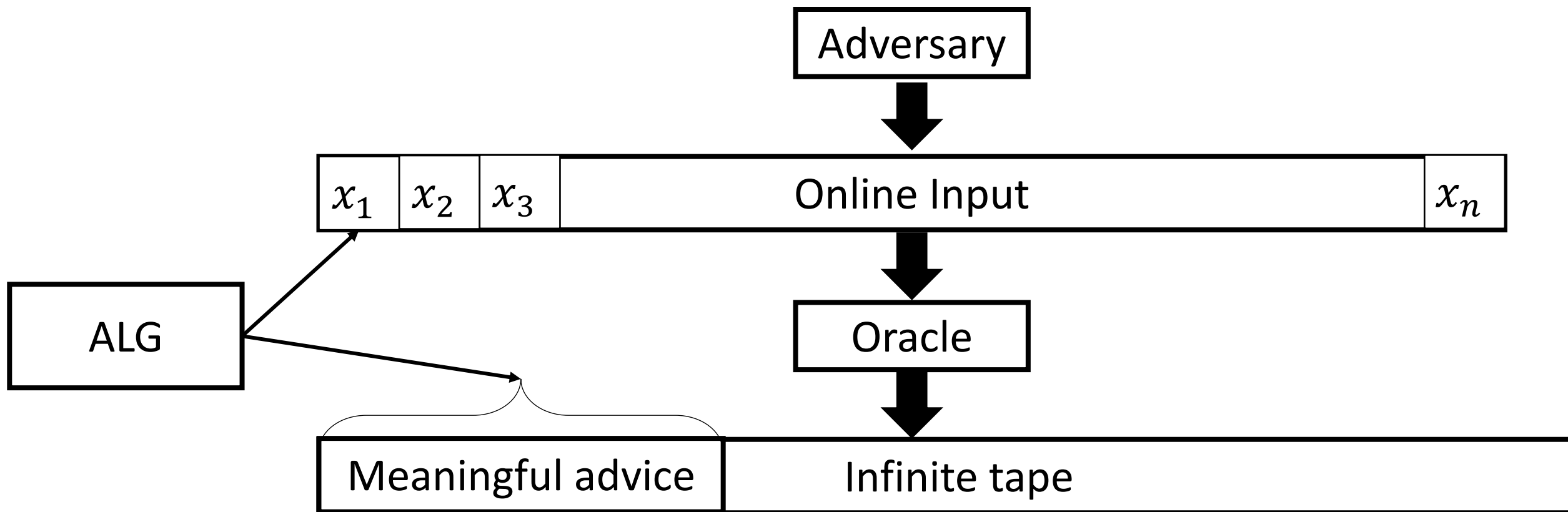
- algorithm makes decisions  $d_1, \dots, d_n$  based on advice and online input

**Advice length:** variable, not fixed a priori

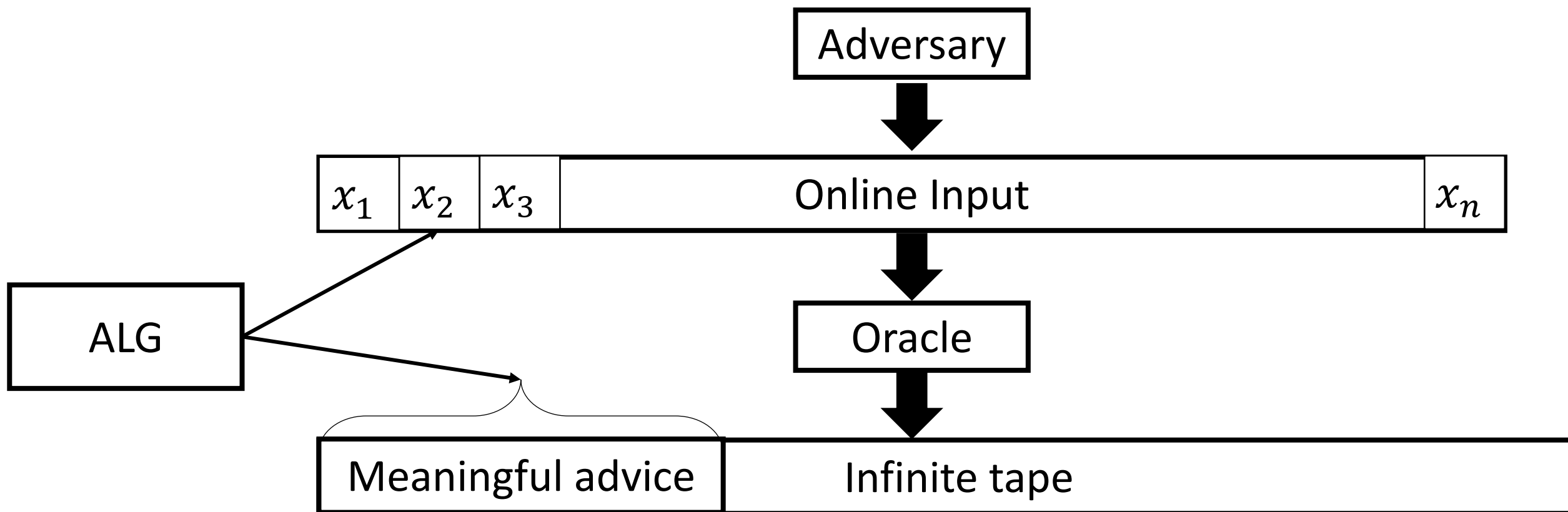
Note that this model allows us to consider  $o(n)$  length advice

Even constant advice is sometimes helpful

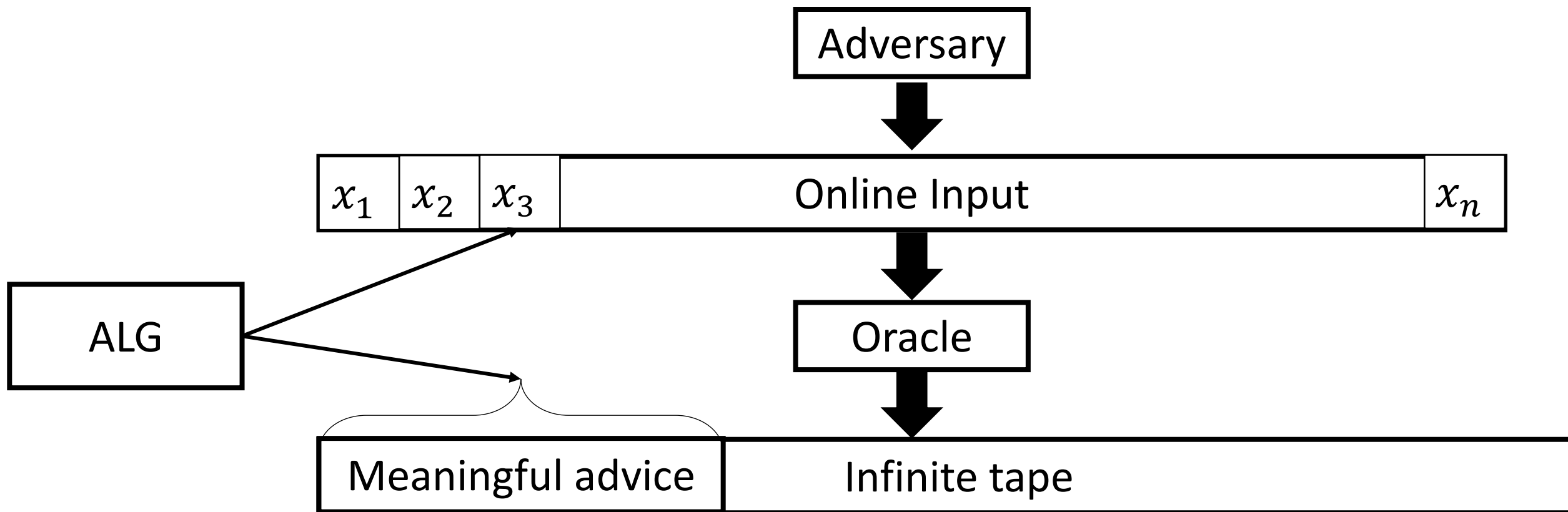
# Tape model



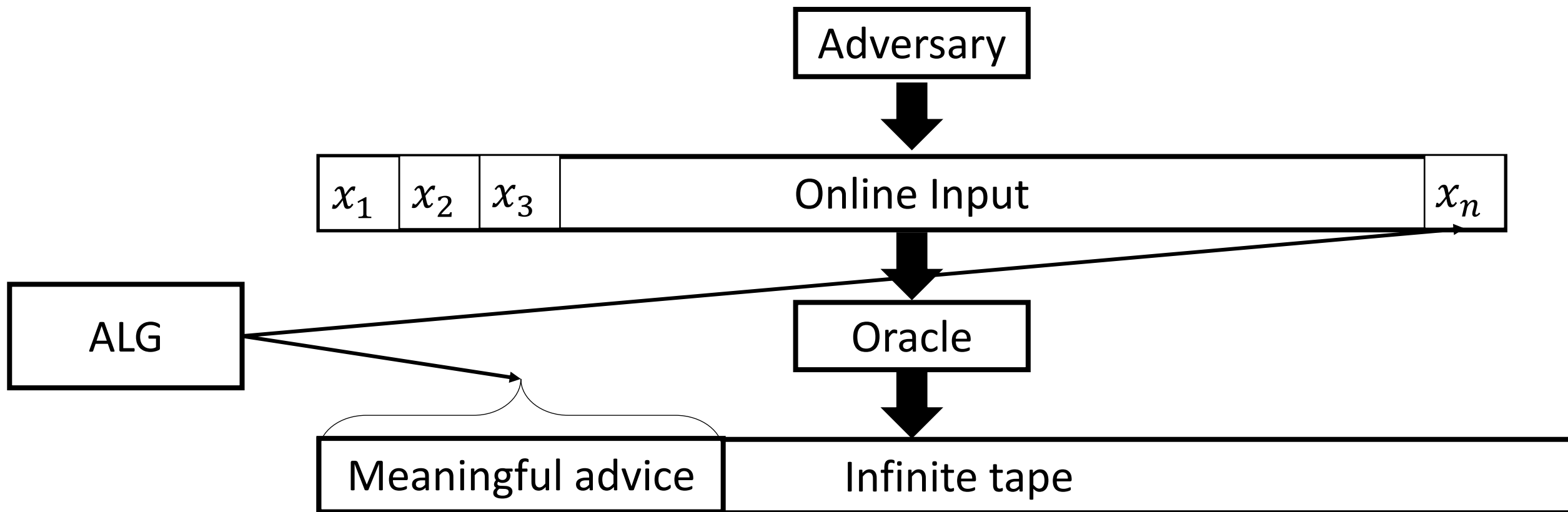
# Tape model



# Tape model



# Tape model



# Note on advice length in the Tape model

Meaningful advice	Infinite tape
-------------------	---------------

In the tape advice model, how does an algorithm know when meaningful advice ends?

There are several possibilities:

- (1) oracle and algorithm can agree on a fixed advice length, e.g., 5
- (2) oracle and algorithm can agree on a protocol such that based on  $(x_1, \dots, x_i)$  and already read advice  $a$ , how many more bits to read for the input  $x_i$
- (3) oracle can encode advice together with its length
- (4) etc...



# Note on encoding

Suppose that an oracle wants to specify the string  
11001

11001 in binary is the same thing as  $2^0 + 2^3 + 2^4 = 1 + 8 + 16 = 25$

Since the tape is infinite, the oracle could populate the tape with  
11001000000000000000 ...

Unless oracle and algorithm has agreed on the length of advice, then  
algorithm doesn't know where meaningful advice ends

# Note on encoding

# The oracle could represent 11001 in unary

**1111111111111111111111100000000000 ..**

That is 25 ones followed by a zero

Algorithm knows exactly where the advice ends – at the first zero


However this increases the length of advice exponentially!

So if advice string has length  $b$  its representation in unary could be of length  $2^b$

# Note on encoding

Better strategy: the oracle specifies the length of advice in unary followed by advice itself

1111101100100000000000 ...



Length of advice    Advice itself  
(in unary)        (in binary)

Thus, encoding advice of length  $b$  requires

$$b + 1 + b = 2b + 1$$

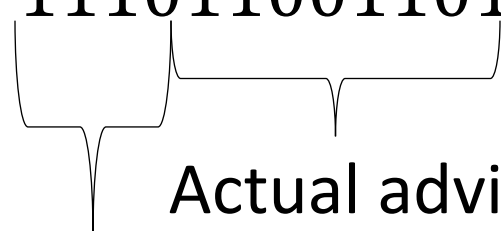
many bits on the advice tape

# Note on encoding

Even better strategy: the oracle specifies the **logarithm** of the length of advice in unary followed by advice itself

Suppose you want to encode 11001101

Then it becomes

1110110011010000000000 ...  
The diagram shows the bit string 1110110011010000000000 followed by an ellipsis. A bracket is drawn under the first three '1's, and a vertical line extends from the center of this bracket to the text 'Actual advice' below it.

3 in unary, means read  $2^3 = 8$  following bits

Similar encodings of length  $b$  can be shown to require  
 $\approx b + 2 \log b + O(1)$

many bits on the advice tape

# Note on encoding

We refer to  $b$  as the **logical length of advice**

The actual length of advice is the length of its encoding  $b + 2 \log b + O(1)$

Since  $\log b \ll b$  we often ignore this difference and simply speak of logical length of advice

# Different views of advice algorithms

## **Oracle view**

All-powerful oracle provides advice bits (how we defined the models)

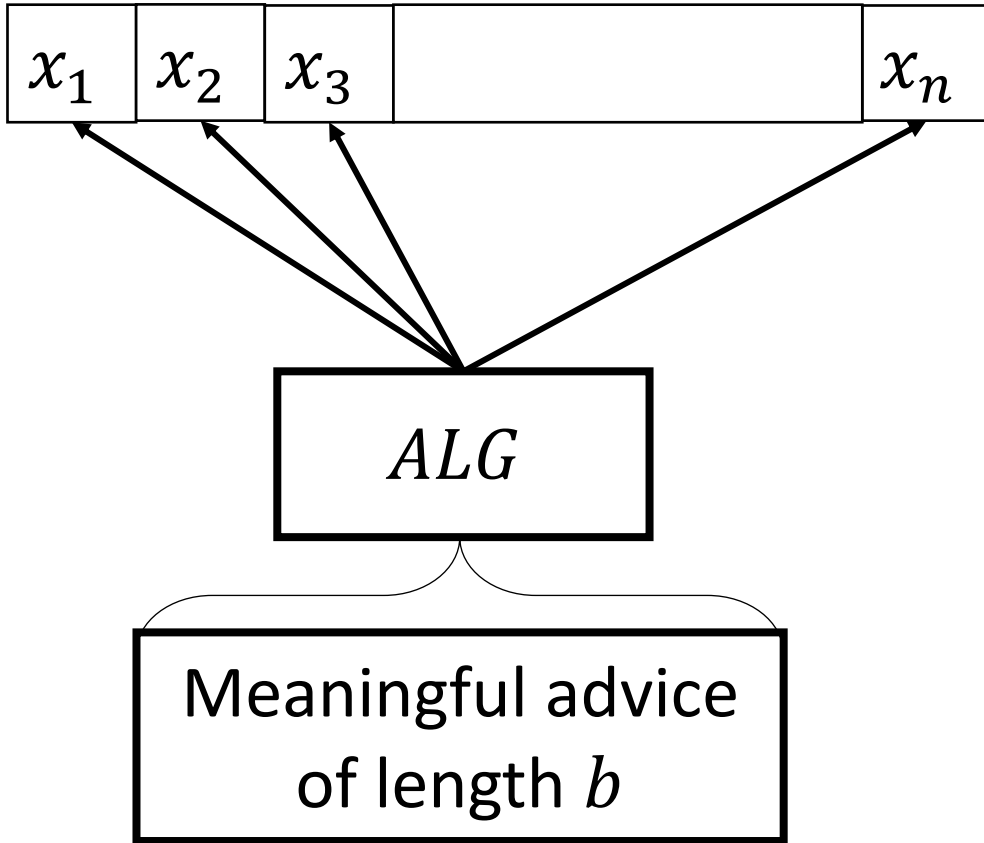
## **Multiple algorithms view (for Tape model)**

*ALG* with  $b$  bits of advice is equivalent to running  $2^b$  deterministic algorithms on a given input and picking the best one.

## **Nondeterministic view**

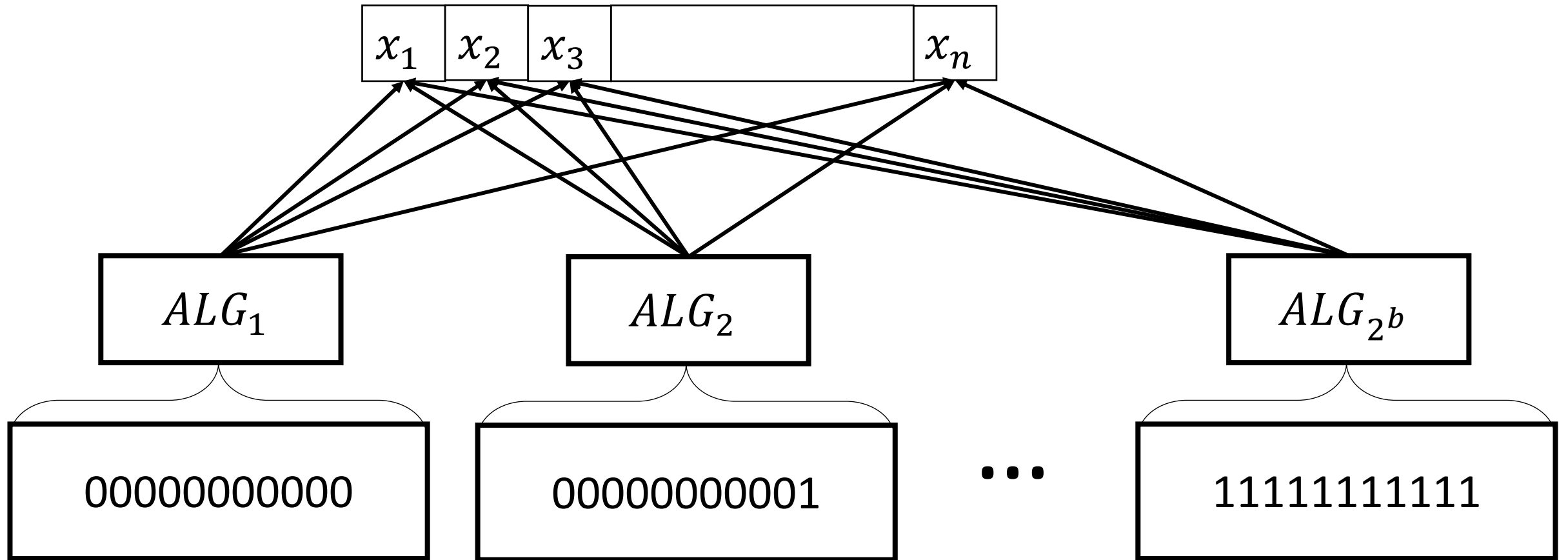
Similar to non-deterministic Turing machines, the algorithm “guesses” the best possible advice string.

# Multiple algorithms view



For any fixed choice of advice string you get a deterministic algorithm  
Enumerate all choices!

# Multiple algorithms view



Oracle knows everything and picks the best advice string for this input



Algorithm with  $b$  bits of advice (in Tape model) is the same thing as

A collection of  $2^b$  deterministic algorithms  $\{ALG_1, ALG_2, \dots, ALG_{2^b}\}$   
without advice

Executing an algorithm with  $b$  bits of advice on input  $x$  is the same as

Running  $ALG_1, \dots, ALG_{2^b}$  on  $x$  in parallel and selecting the best  
performing algorithm for that input

# Plan, part I

We will study the following techniques (with examples)

## Upper bound techniques:

1. Now-or-later (e.g., Paging)
2. Follow  $OPT$  (e.g.,  $k$ -Server)
3. Adapting randomized algorithms (e.g., Proportional Knapsack)
4. Adapting Offline Algorithms (e.g., Bin Packing)
5. Combinatorial Designs (e.g., Matching on trees)

# Plan, part II

We will study the following techniques (with examples)

## Lower bound techniques:

1. Pigeonhole argument (e.g.,  $k$ -Server)
2. String Guessing and Reductions (e.g., graph problems)
3.  $\Sigma$ -repeatable problems (e.g., Paging)
4.  $V$ -repeatable problems (e.g., Graph Coloring)

Upper bound  
techniques:  
Now-or-Later

# Now-or-Later

Applies in the Per Request model

Main idea: specify a constant number of bits for each item, which indicate whether the item can be used for the decision immediately, or if it can be deferred to future

# Paging

One way to get optimality (= strict competitive ratio 1) is to specify  $\log k$  bits of advice per request

$p_i$  - request  $i$

$a_i \in [k]$  – which page to evict if  $p_i$  is not in cache

Total advice length  $n \log k$

# Paging

Can we do better?

Yes, we can achieve optimality with  $n$  bits of advice

$\tilde{a}_i \in \{0,1\}$  indicates if page  $p_i$  has to be kept in cache until next time it is requested

Evict only those pages that have  $\tilde{a}_i = 0$

Example: cache size  $k = 4$

Input Sequence

5	4	3	2	1	7	6	4	5	2	3	1
---	---	---	---	---	---	---	---	---	---	---	---

OPT cache

	5	5	5	5	5	5	5	5	5	3	1
		4	4	4	4	4	4	4	4	4	4
			3	3	1	7	6	6	6	6	6
				2	2	2	2	2	2	2	2

Advice  $a_i$

1	2	3	4	3	3	*	*	*	*	1	1
---	---	---	---	---	---	---	---	---	---	---	---



Example: cache size  $k = 4$

Input Sequence

5	4	3	2	1	7	6	4	5	2	3	1
---	---	---	---	---	---	---	---	---	---	---	---

OPT cache

	5	5	5	5	5	5	5	5	5	3	1
		4	4	4	4	4	4	4	4	4	4
			3	3	1	7	6	6	6	6	6
				2	2	2	2	2	2	2	2

Advice  $\tilde{a}_i$

1	1	0	1	0	0	1	1	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---

Example: cache size  $k = 4$

Input Sequence

5	4	3	2	1	7	6	4	5	2	3	1
---	---	---	---	---	---	---	---	---	---	---	---

Advice  $\tilde{a}_i$

1	1	0	1	0	0	1	1	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---

ALG cache

	5	5	5	5	5	5	5	5	5	3	1
		4	4	4	4	4	4	4	4	4	4
			3	3	1	7	6	6	6	6	6
				2	2	2	2	2	2	2	2

# Describing algorithm with advice

**(1) What does advice encode?**

**(2) What is the size of advice?**

**(3) How does the algorithm work with advice?**

**(4) Why is this algorithm optimal?**

# Describing algorithm with advice

## **(1) What does advice encode?**

$\tilde{a}_i = 1$  if page  $p_i$  should remain in cache until next time it's requested

## **(2) What is the size of advice?**

1 bit per request, or  $n$  total bits

## **(3) How does the algorithm work with advice?**

To accommodate a new page, evict a page from cache with  $\tilde{a}_i = 0$

## **(4) Why is this algorithm optimal?**

It only evicts those pages that are “safe” to evict. Formal proof can be done by induction

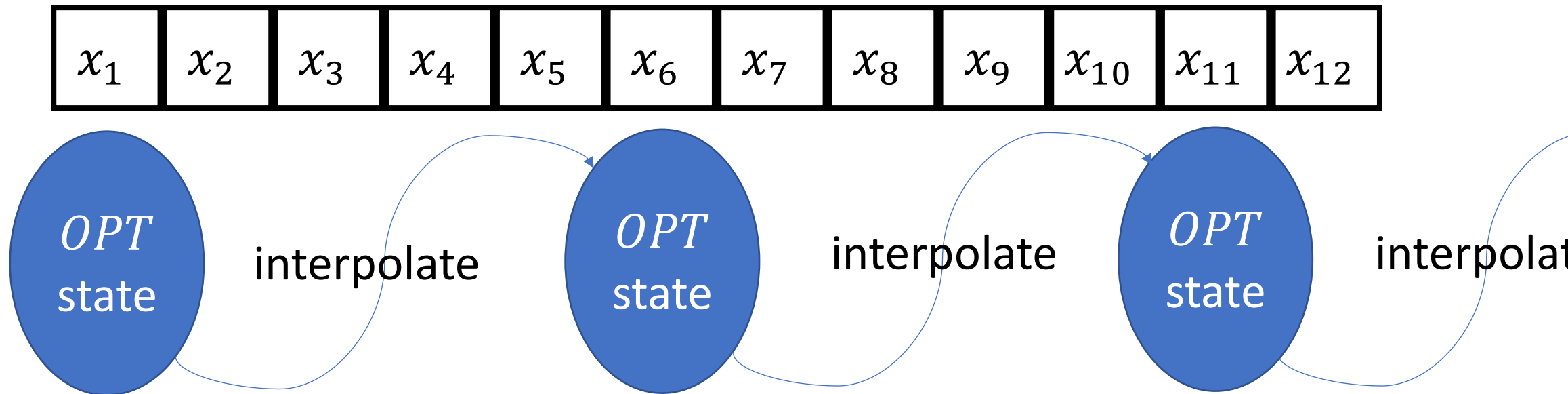
Upper bound  
techniques: Follow  
*OPT*

# Follow *OPT* technique

Provide “snapshots” of the state of *OPT* at certain requests (not at every request)

*ALG* recovers the exact state of *OPT* at those snapshot points

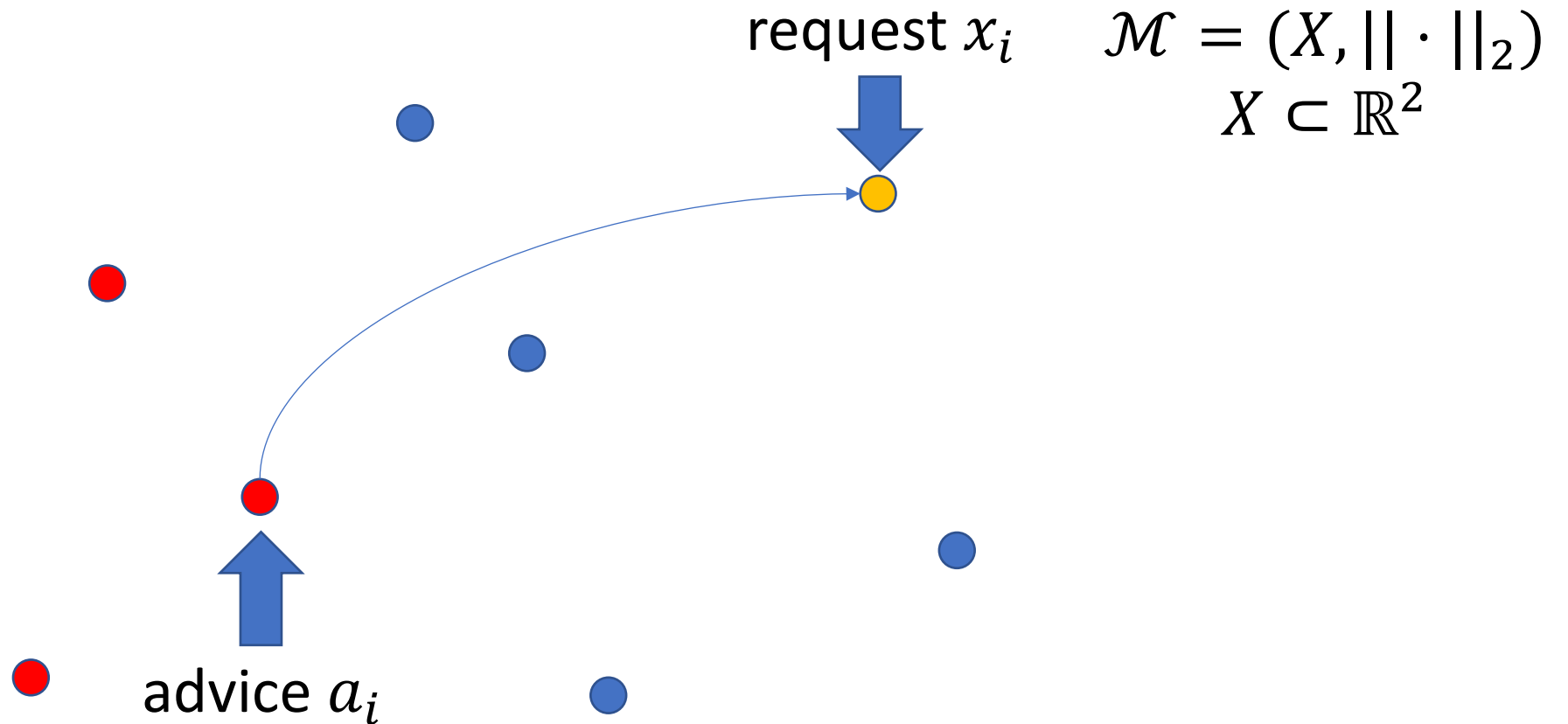
*ALG* tries to “interpolate” between the snapshots



# $k$ -Server

One way to achieve optimality with advice is to specify which server  $OPT$  uses to process each request

$k = 3$



# $k$ -Server

One way to achieve optimality with advice is to specify which server  $OPT$  uses to process each request

$a_i$  - name of the server to use to process request  $x_i$

$\log k$  bits of advice **per request**

$n \log k$  - **total number of advice bits**

Also, recall  $\rho(ALG) \geq k$  for any deterministic algorithm  $ALG$  and metric space with at least  $k + 1$  points



# $k$ -Server with advice

Our goal is to improve upon competitive ratio  $k$  with advice of total length  $\ll n \log k$

We will use "follow  $OPT$ " technique to design an online algorithm  $CHASE$  in **Tape Advice Model** such that

- Advice length of  $CHASE$  is  $3n$
- Competitive ratio of  $CHASE$  is  $\log k$

# *CHASE* algorithm

Input sequence for  $k$ -Server:

$$x_1, x_2, x_3, \dots, x_n$$

Fix some offline optimal algorithm  $OPT$  that is lazy

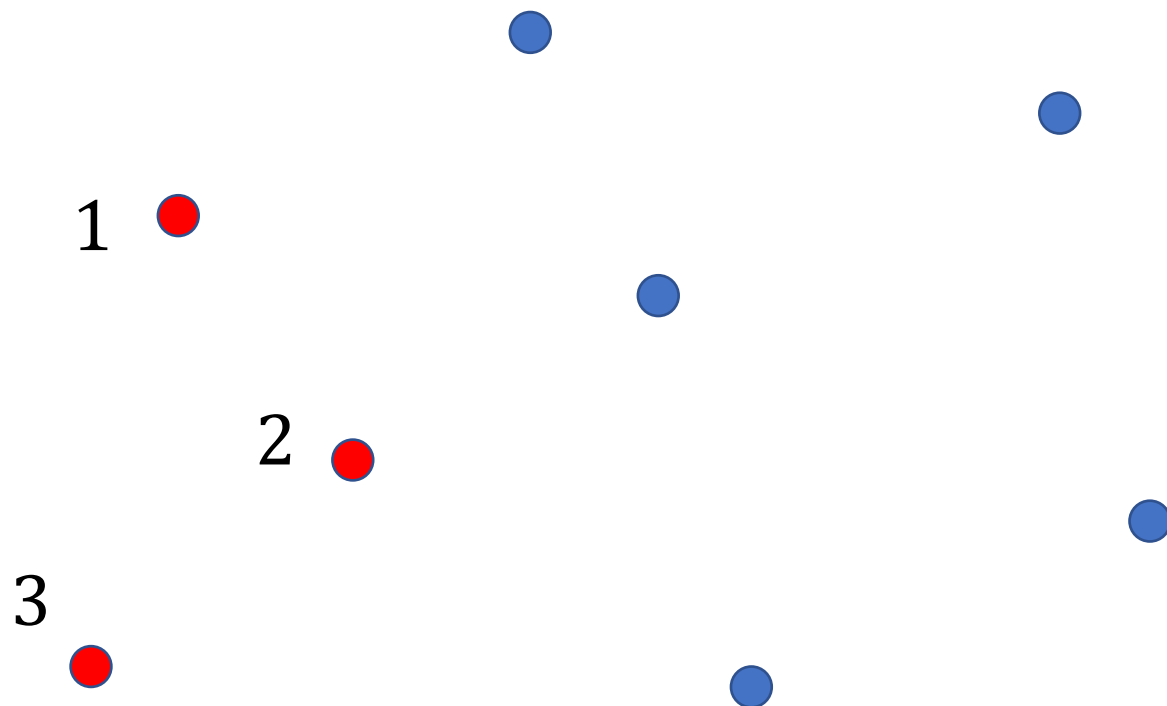
Split input sequence into  $k$  subsequences  $x^{(1)}, x^{(2)}, \dots, x^{(k)}$

$x^{(s)}$  - those requests  $x_i$  that are served by server  $s$  in  $OPT$

# Initial configuration

$k = 3$

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$



$x^{(1)} :$

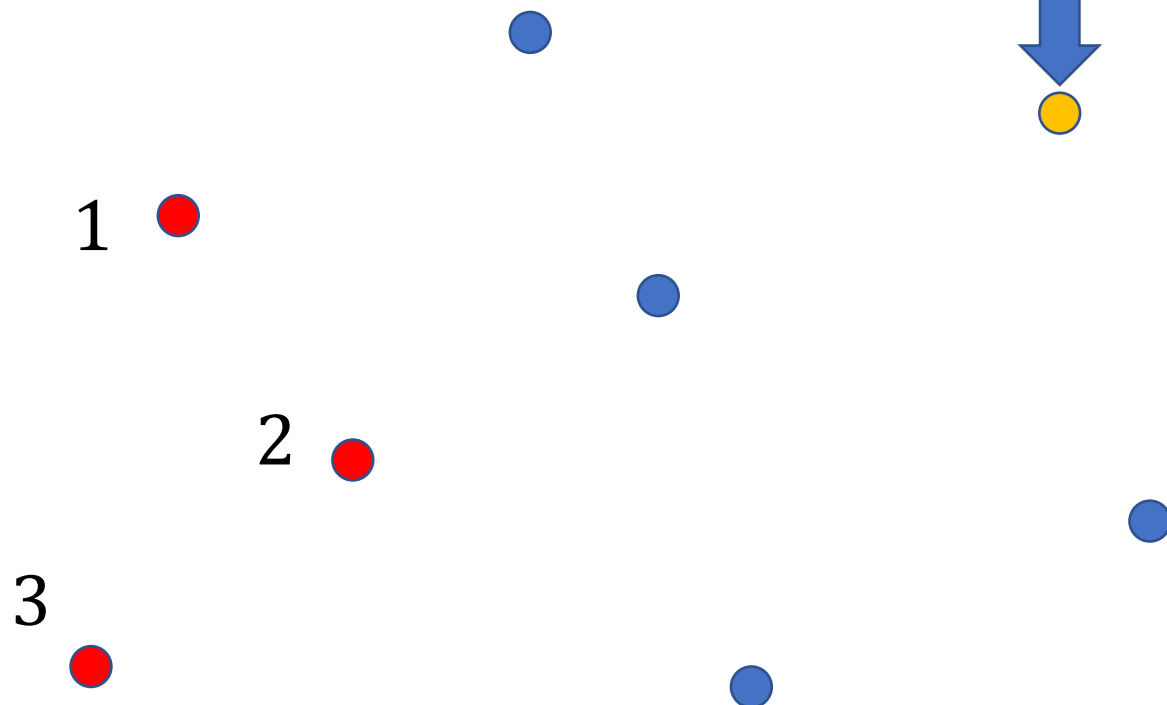
$x^{(2)} :$

$x^{(3)} :$

# New request

$k = 3$

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$



$x^{(1)} :$

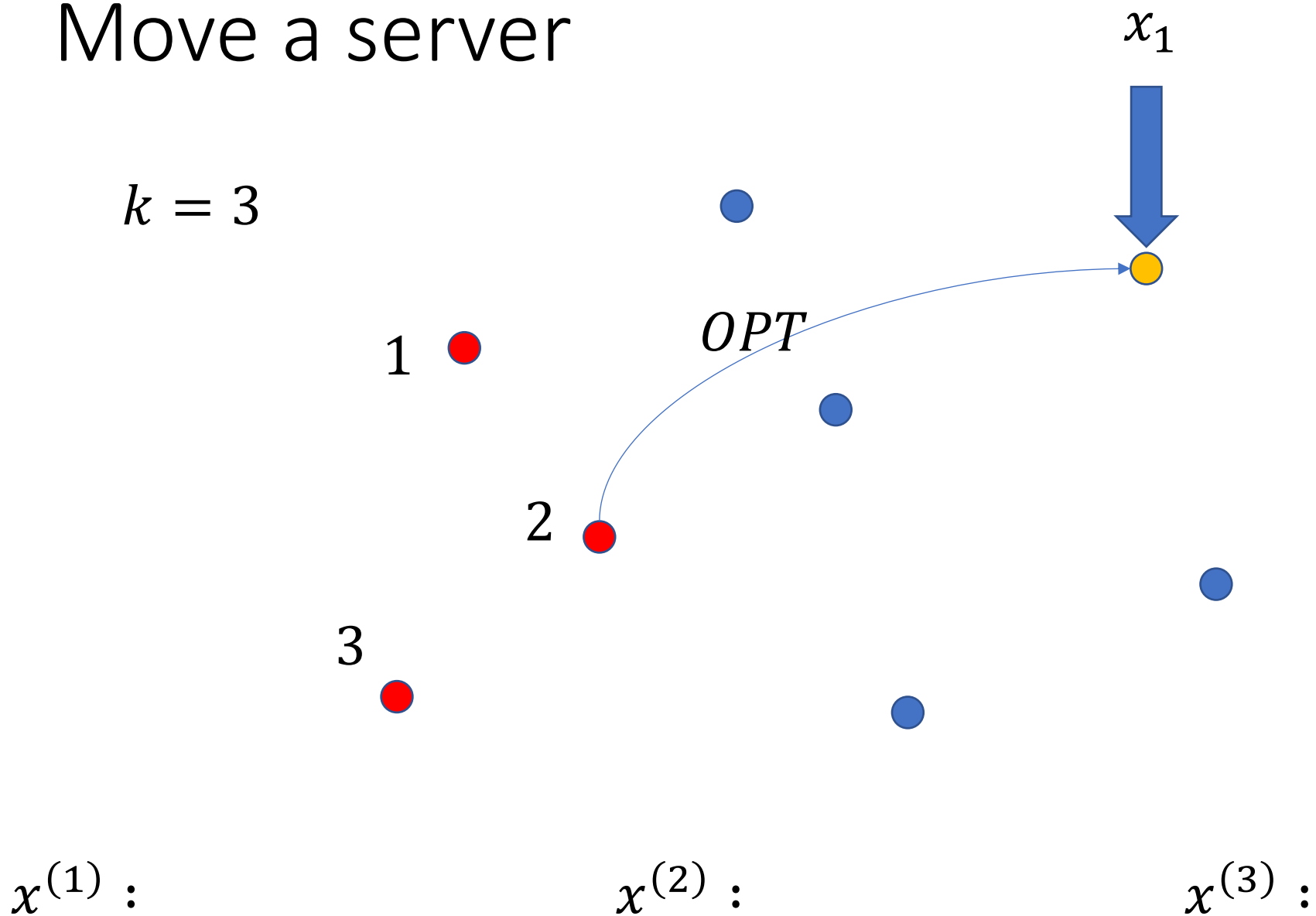
$x^{(2)} :$

$x^{(3)} :$

# Move a server

$k = 3$

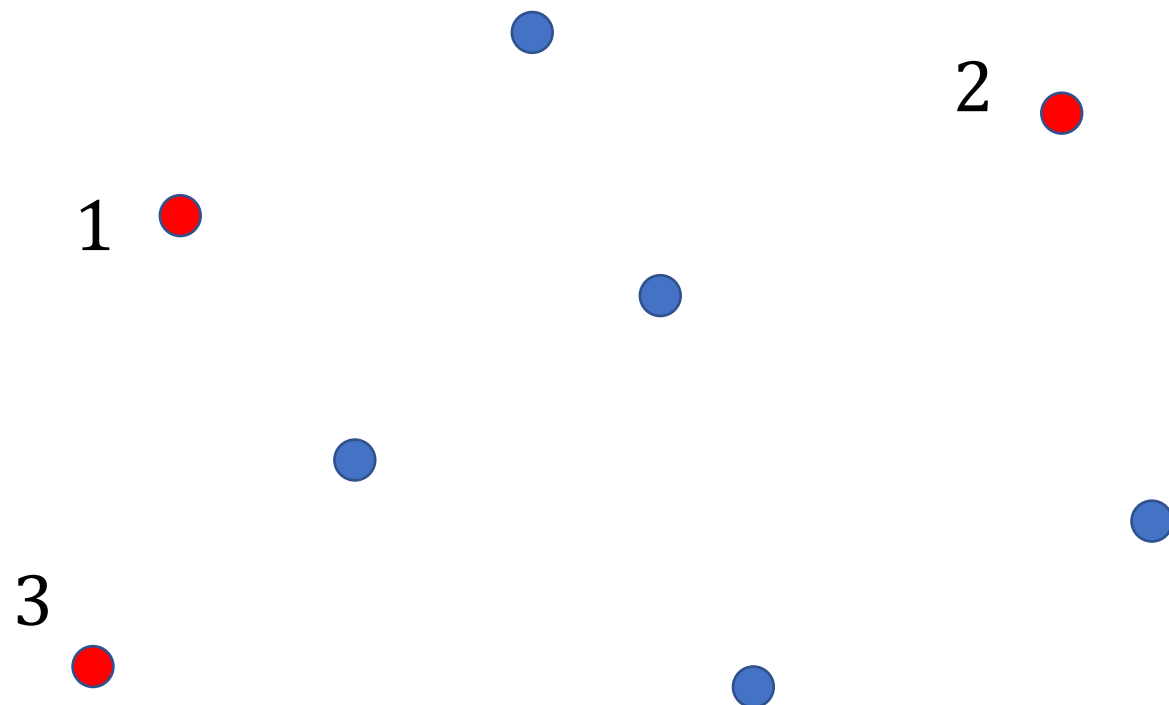
$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$



# Move a server

$k = 3$

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$



$x^{(1)} :$

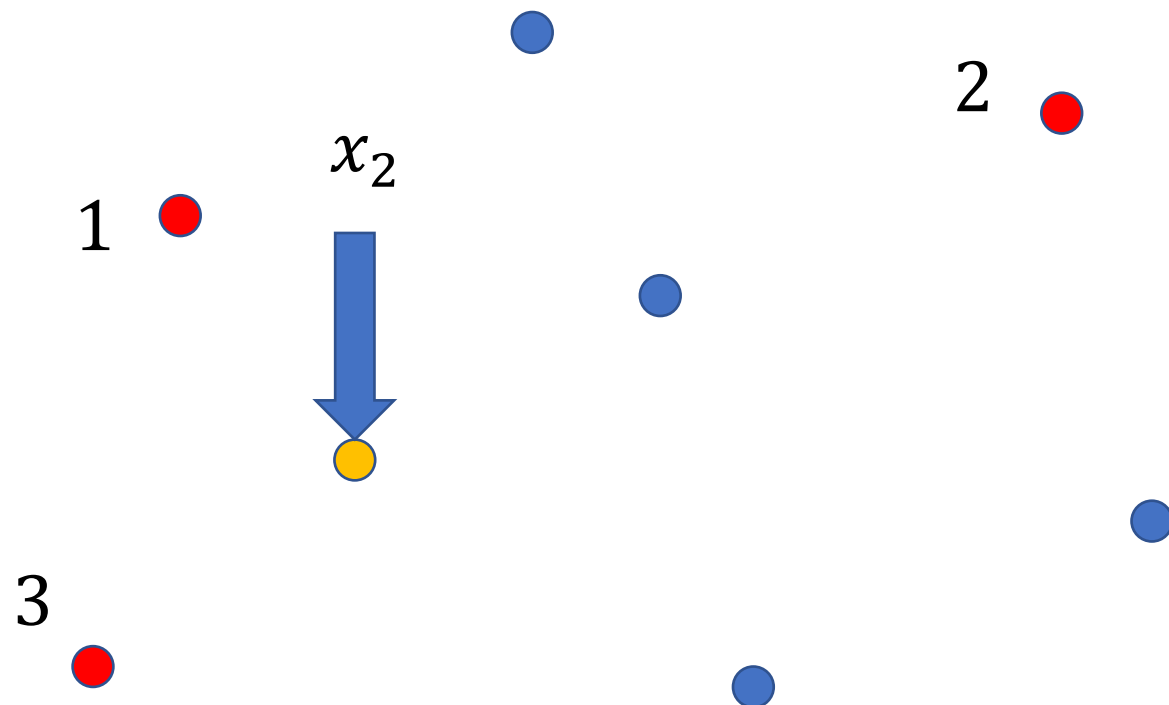
$x^{(2)} : x_1$

$x^{(3)} :$

# New request

$k = 3$

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$



$x^{(1)} :$

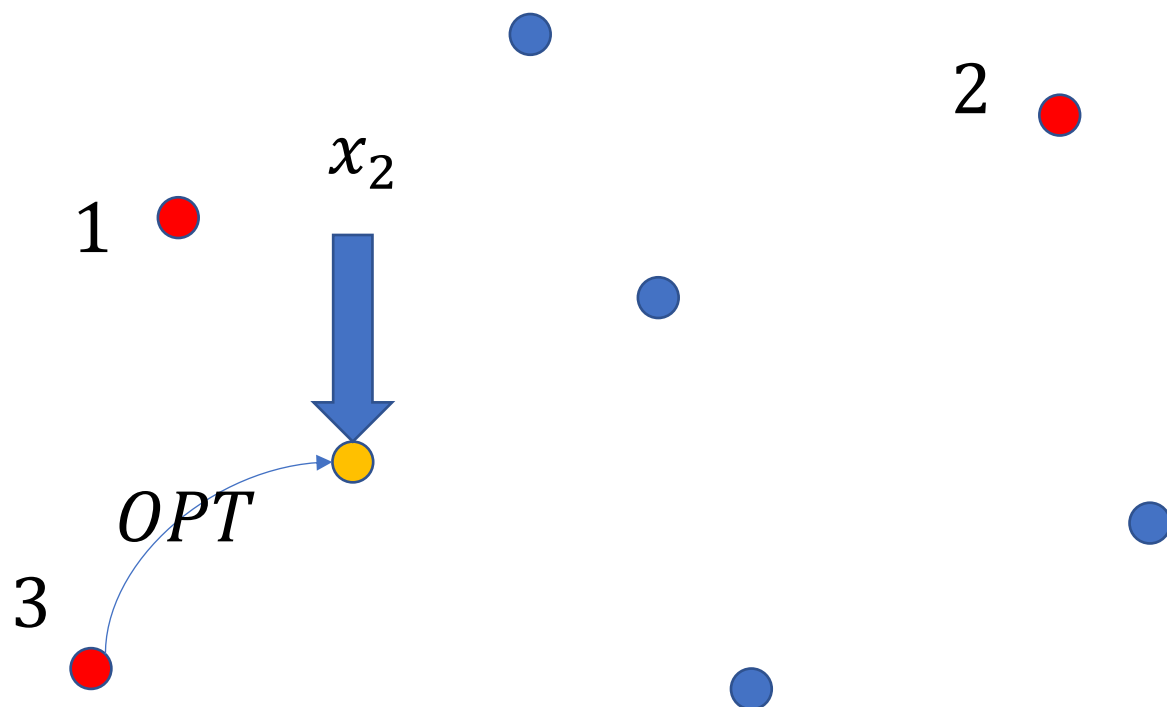
$x^{(2)} : x_1$

$x^{(3)} :$

# Move a server

$k = 3$

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$



$x^{(1)} :$

$x^{(2)} : x_1$

$x^{(3)} :$



$k = 3$

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$

1

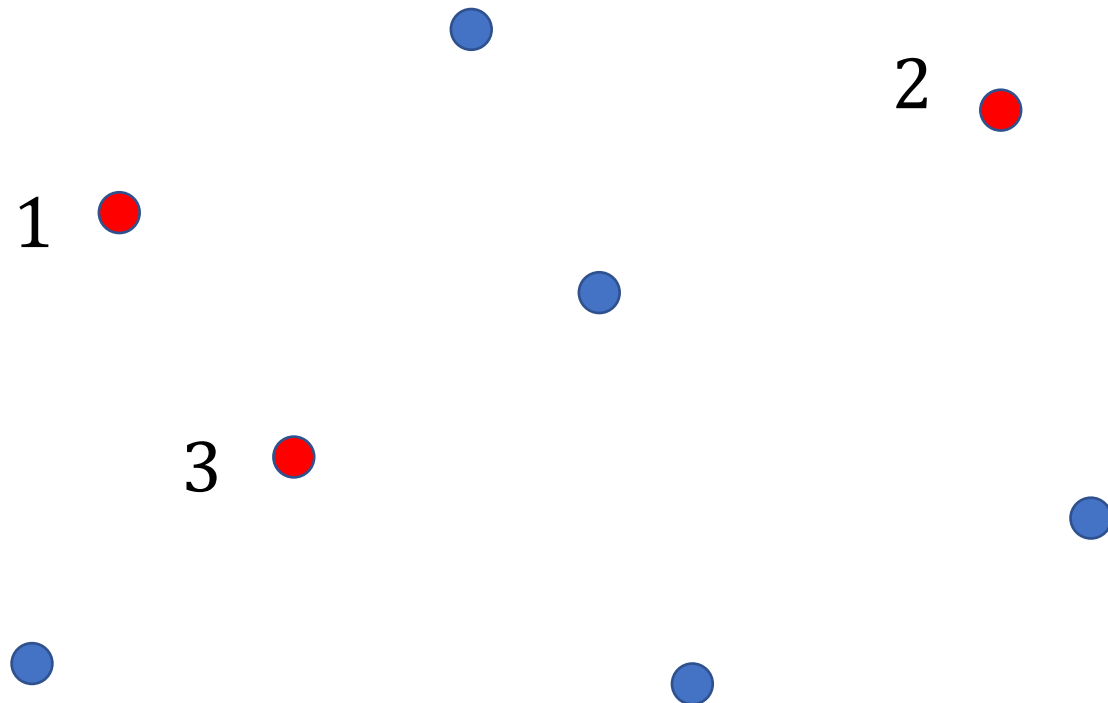
2

3

$x^{(1)} :$

$x^{(2)} : x_1$

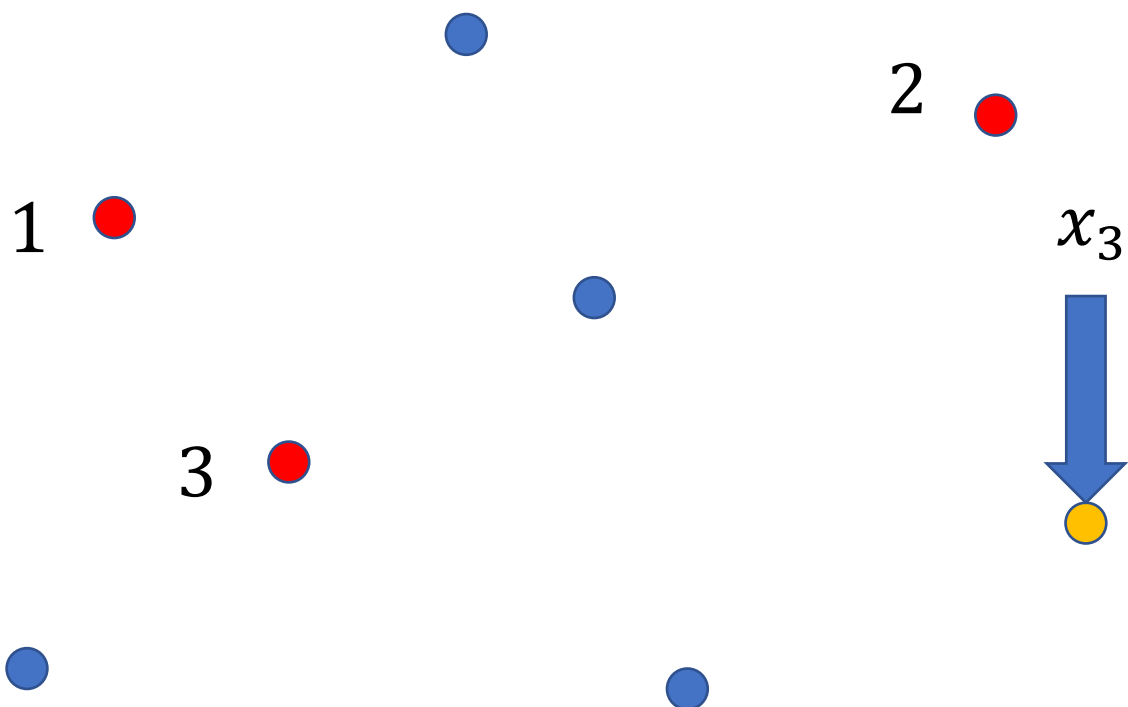
$x^{(3)} : x_2$



# New request

$k = 3$

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$



$x^{(1)} :$

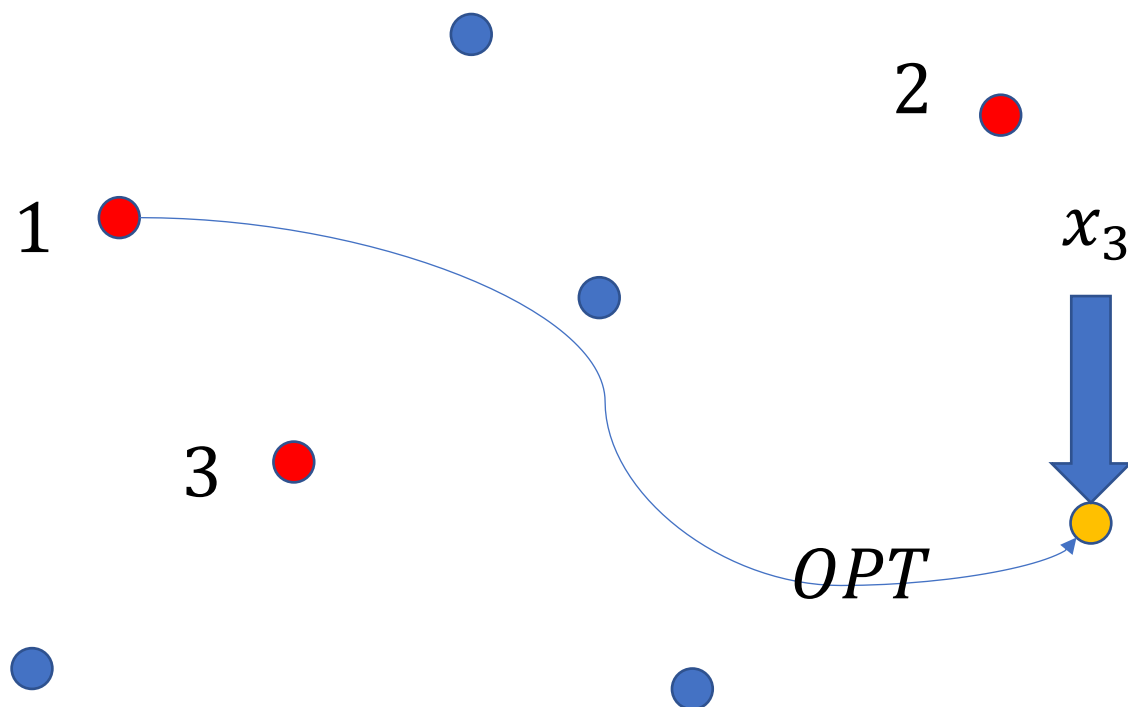
$x^{(2)} : x_1$

$x^{(3)} : x_2$

# Move a server

$k = 3$

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$



$x^{(1)} :$

$x^{(2)} : x_1$

$x^{(3)} : x_2$

$k = 3$

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$

2

3

1

$x^{(1)} : x_3$

$x^{(2)} : x_1$

$x^{(3)} : x_2$

$k = 3$

$x_4$



$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$

2





3





1



$x^{(1)} : x_3$



$x^{(2)} : x_1$



$x^{(3)} : x_2$

$k = 3$

$x_4$

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$

*OPT*

2

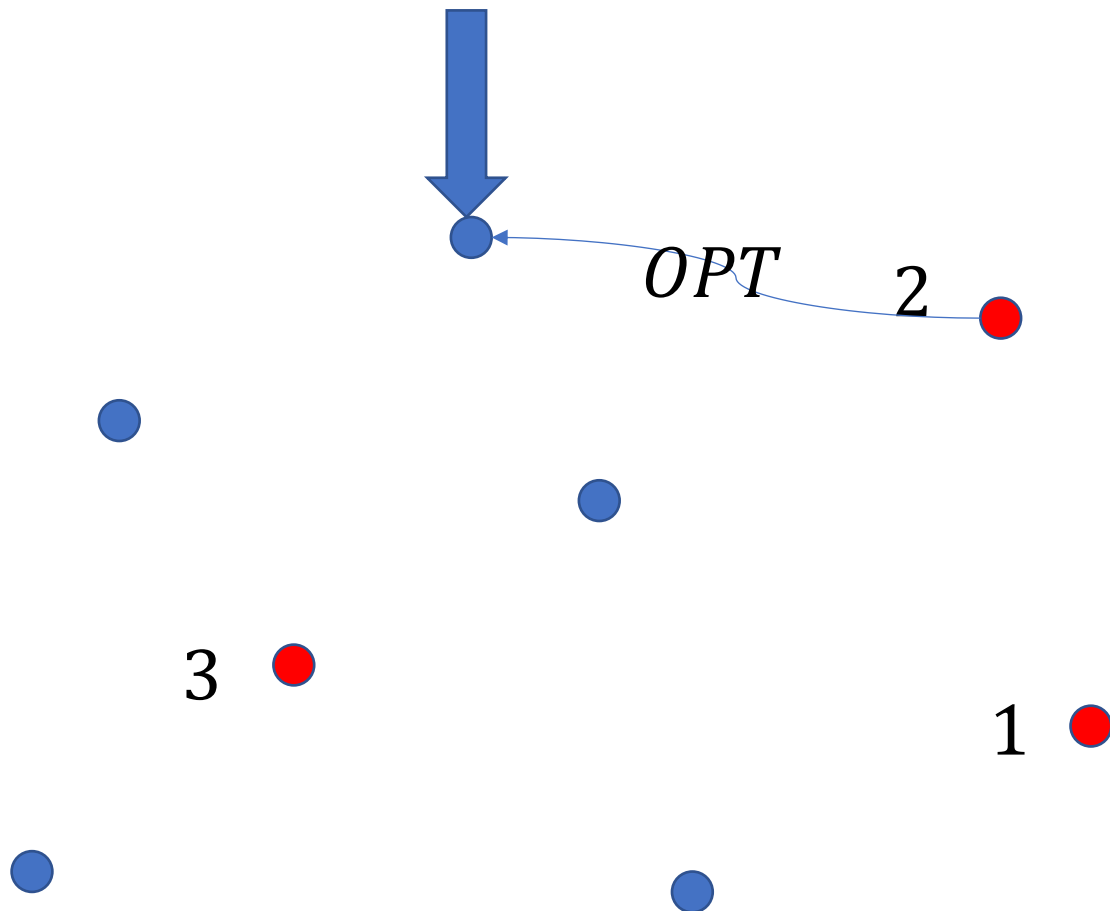
3

1

$x^{(1)} : x_3$

$x^{(2)} : x_1$

$x^{(3)} : x_2$



$$\mathcal{M} = (X, \|\cdot\|_2)$$

$$X \subset \mathbb{R}^2$$

$$k = 3$$

2

3

1

$$x^{(1)} : x_3$$

$$x^{(2)} : x_1, x_4$$

$$x^{(3)} : x_2$$

$k = 3$

2

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$

$x_5$

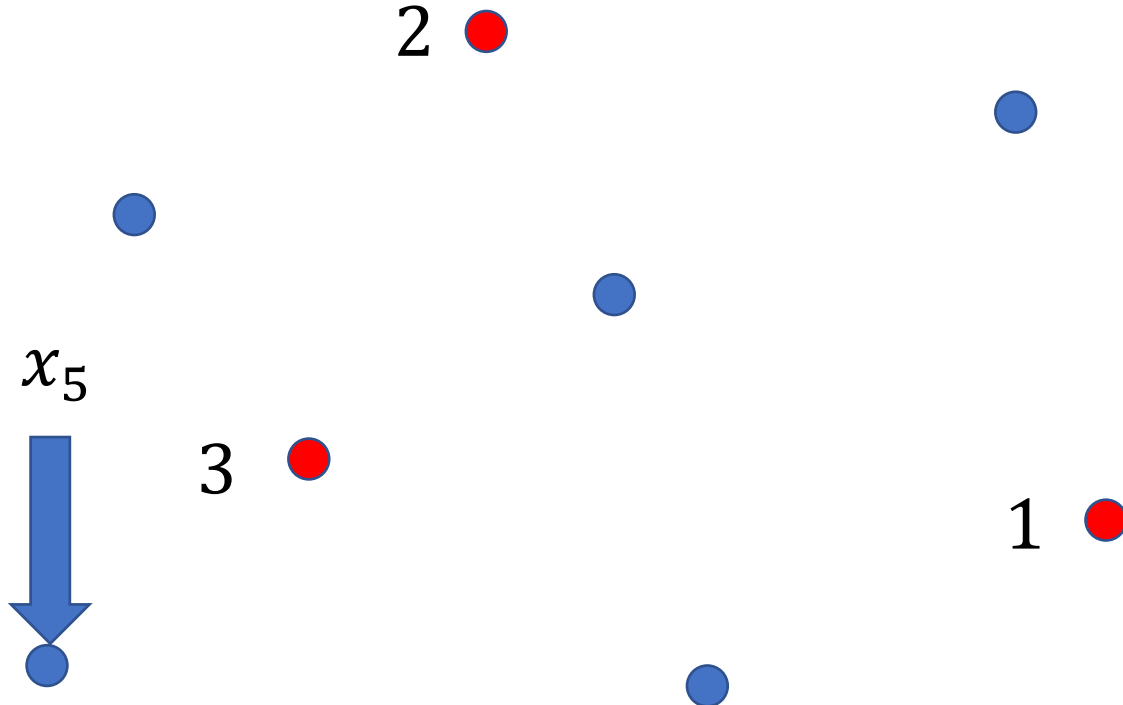
3

1

$x^{(1)} : x_3$

$x^{(2)} : x_1, x_4$

$x^{(3)} : x_2$





$k = 3$

$$\mathcal{M} = (X, \|\cdot\|_2)$$
$$X \subset \mathbb{R}^2$$

2



$x_5$



3



*OPT*



1



$x^{(1)} : x_3$

$x^{(2)} : x_1, x_4$

$x^{(3)} : x_2$

$k = 3$

2



1

3



$x^{(1)} : x_3$

$x^{(2)} : x_1, x_4$

$x^{(3)} : x_2, x_5$

$\mathcal{M} = (X, || \cdot ||_2)$   
 $X \subset \mathbb{R}^2$

# *CHASE* algorithm

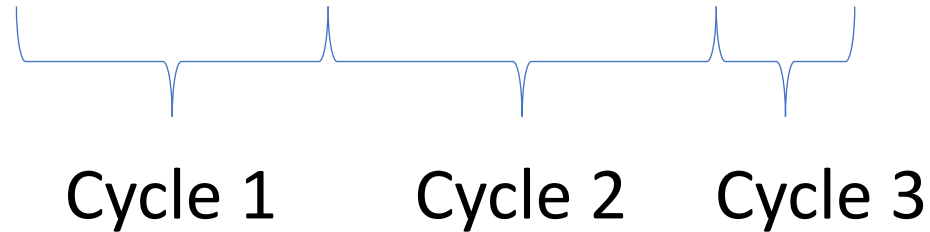
For simplicity assume that  $k$  is an even power of 2

$$\alpha := \frac{\log k}{2} \in \mathbb{N}$$

Subsequence  $x^{(i)} = x_{j_1}, x_{j_2}, \dots, x_{j_\ell}$  of length  $\ell$  is further subdivided into contiguous subsequences of length  $\alpha$  each, called cycles

# *CHASE* algorithm

Example:  $\alpha = 3$  and  $x^{(1)} : x_2, x_5, x_8, x_{10}, x_{11}, x_{23}, x_{48}$



Note: last cycle may be incomplete

# *CHASE* algorithm

**ORACLE** on request  $x_i$ :

- Find subsequence  $x^{(s)}$  such that  $x_i \in x^{(s)}$
- Find cycle  $C$  inside  $x^{(s)}$  such that  $x_i \in C$
- If  $x_i$  is the last element of  $C$ 
  - Oracle writes  $1s$  (1 followed by name of the server  $s$ ) on the advice tape
- Otherwise
  - Oracle writes 0 on the advice tape

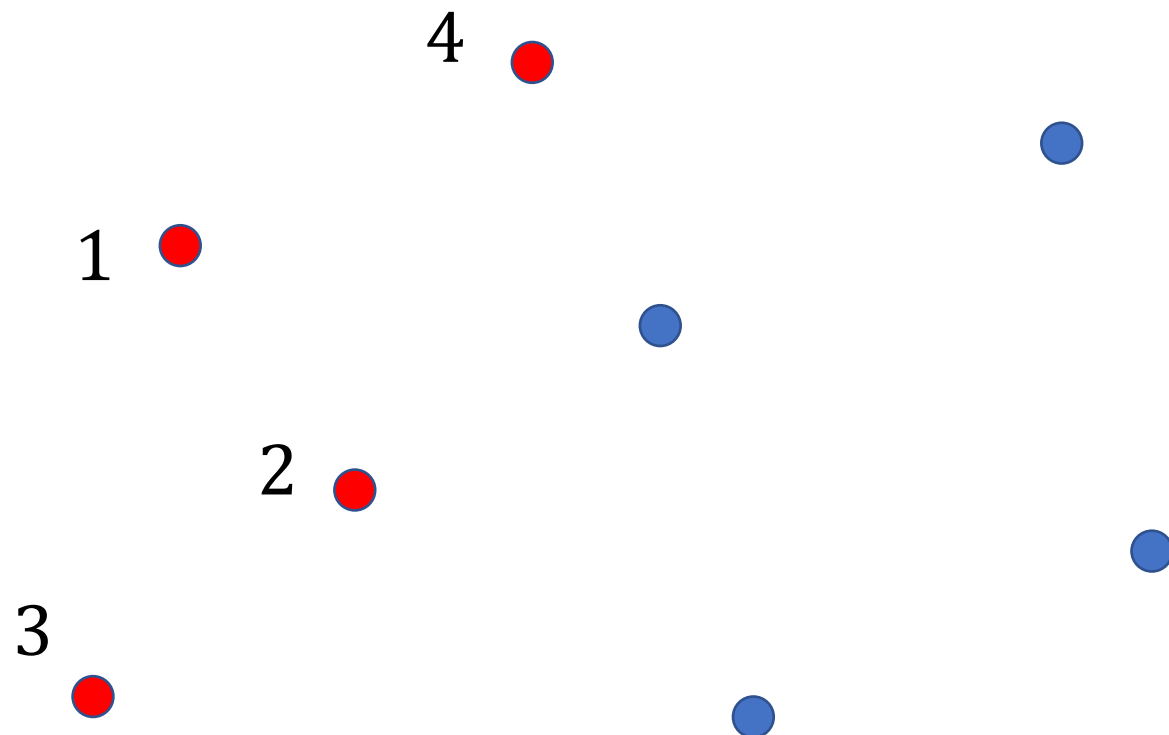
Key idea: specify which server  $OPT$  uses to process last elements of cycles

# *CHASE* algorithm

*CHASE* algorithm on request  $x_i$ :

- Read next bit of advice from the tape
- If the bit is 1
  - Read  $\log k$  more bits from the advice tape
  - Interpret those bits as the name of a server  $s$
  - Move  $s$  to  $x_i$
- Otherwise
  - Find a server  $s$  that is **closest** to  $x_i$
  - Move  $s$  to  $x_i$
  - After processing  $x_i$  **move  $s$  back to its original location**

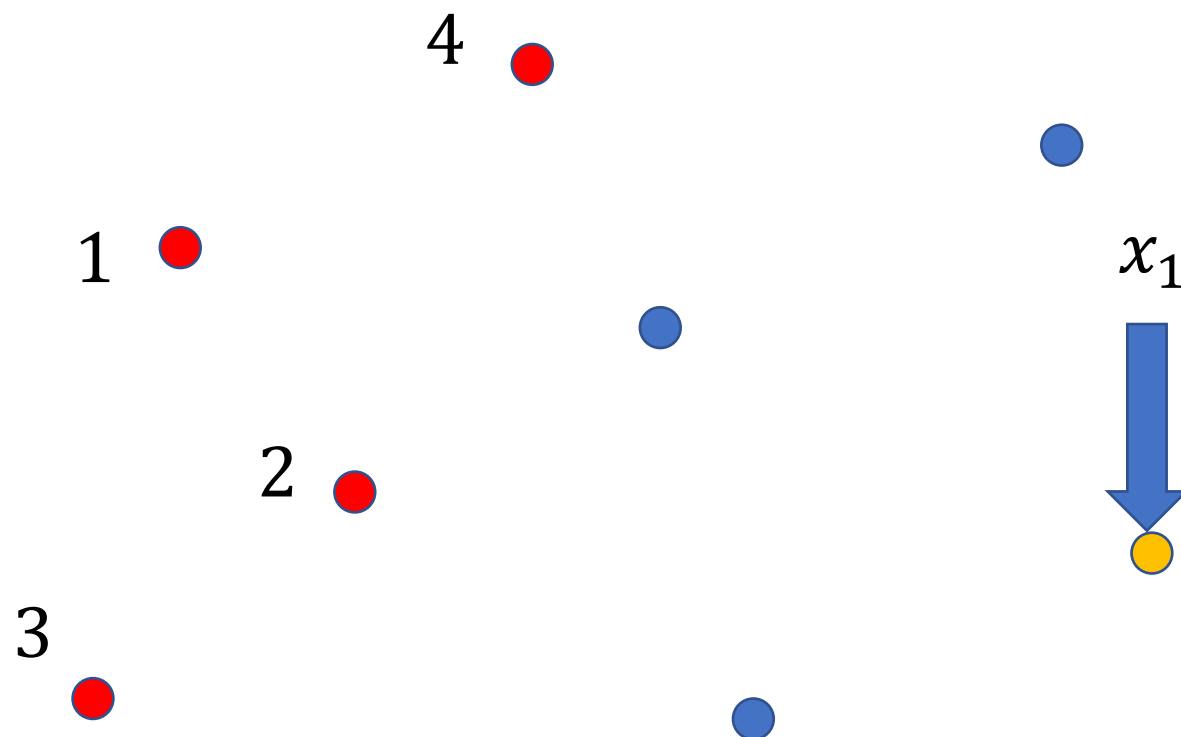
# *CHASE* example



Advice tape:

0	0	1	0	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---

# *CHASE* example

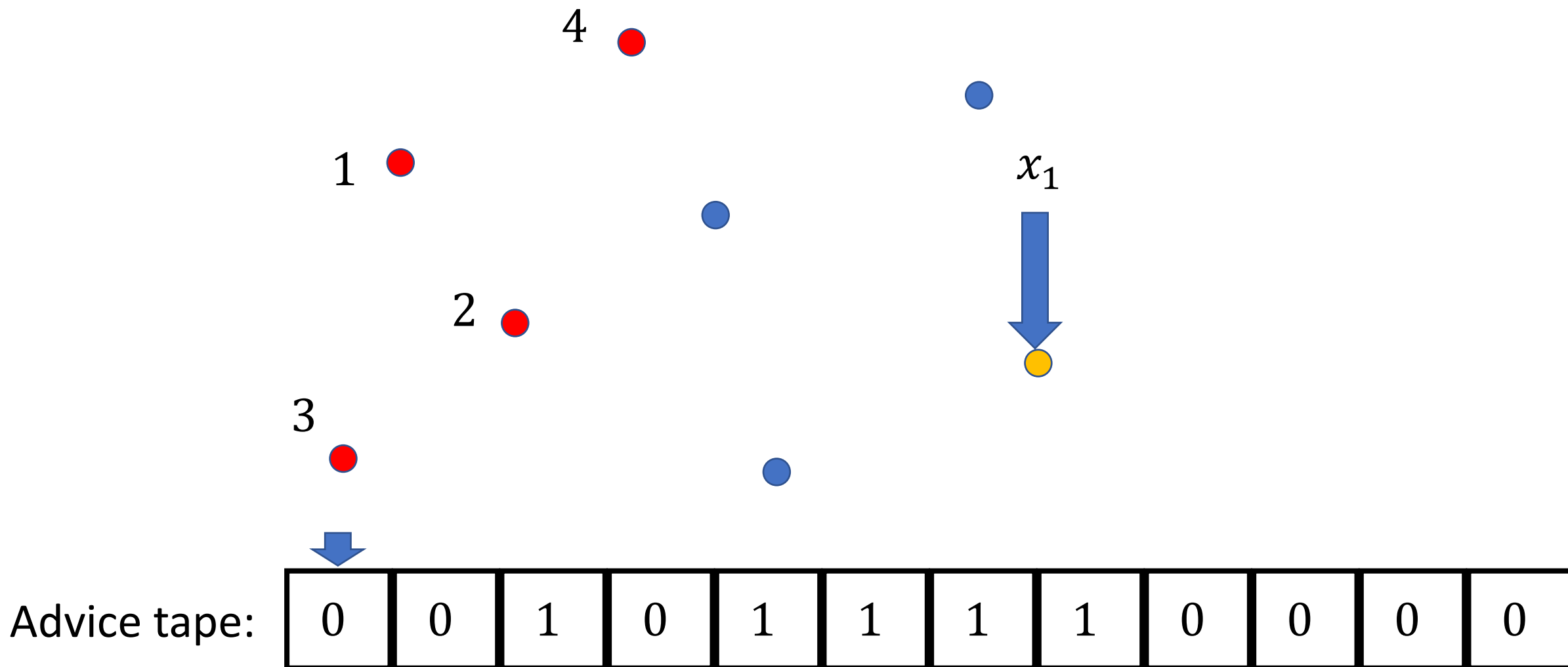


Advice tape:

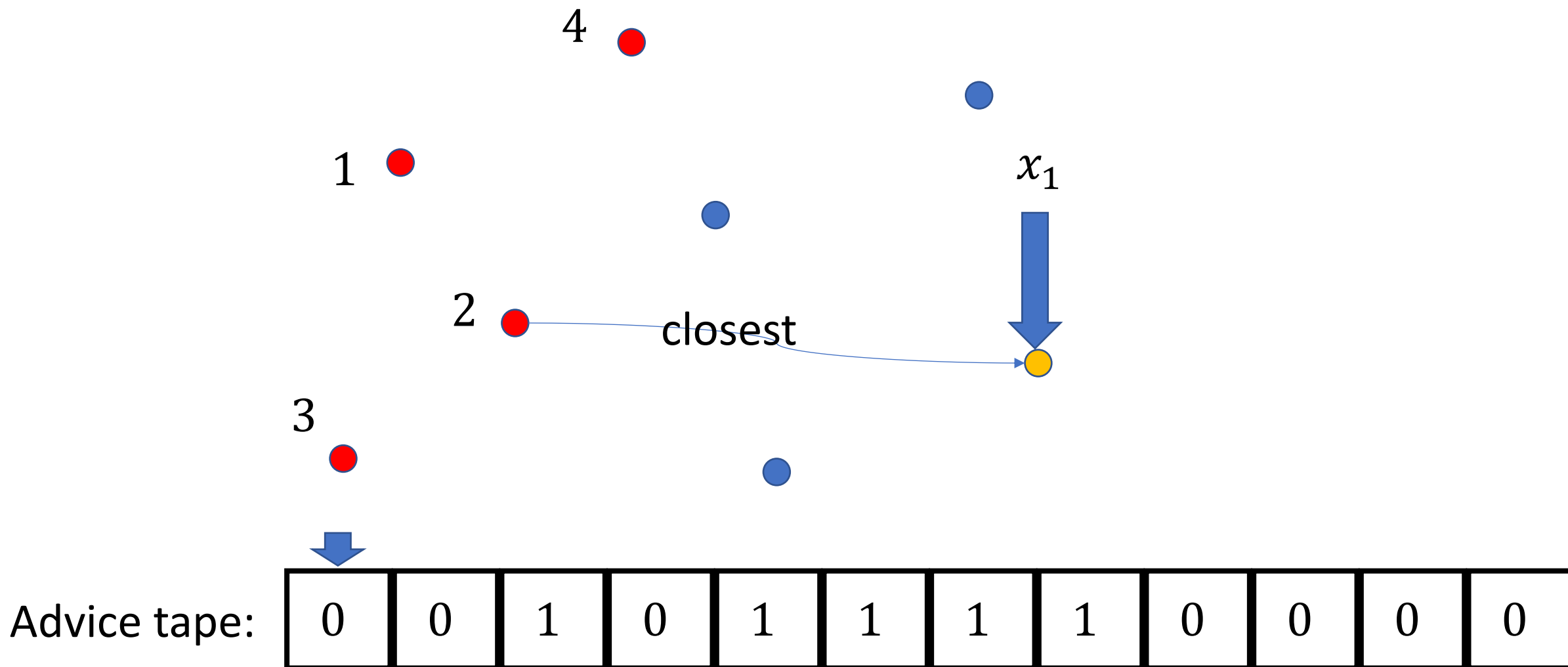
0	0	1	0	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---



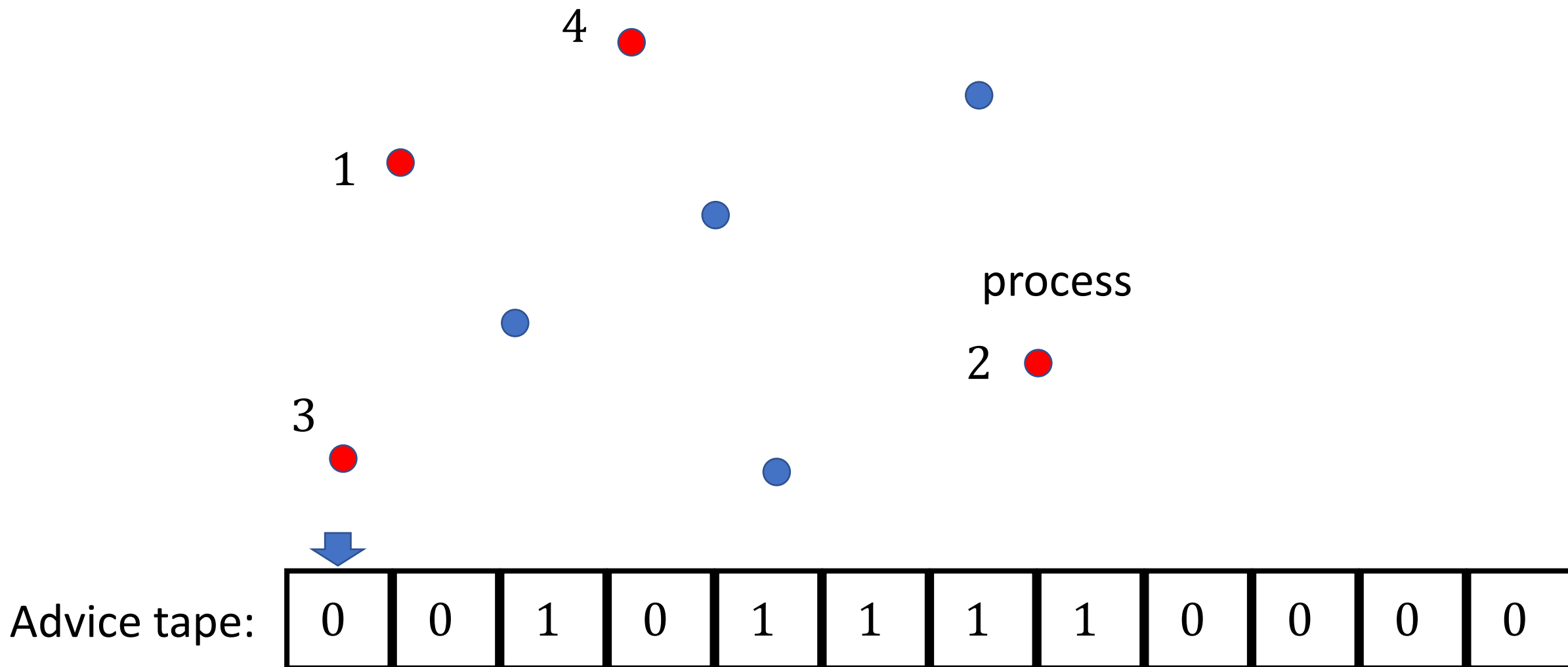
# *CHASE* example



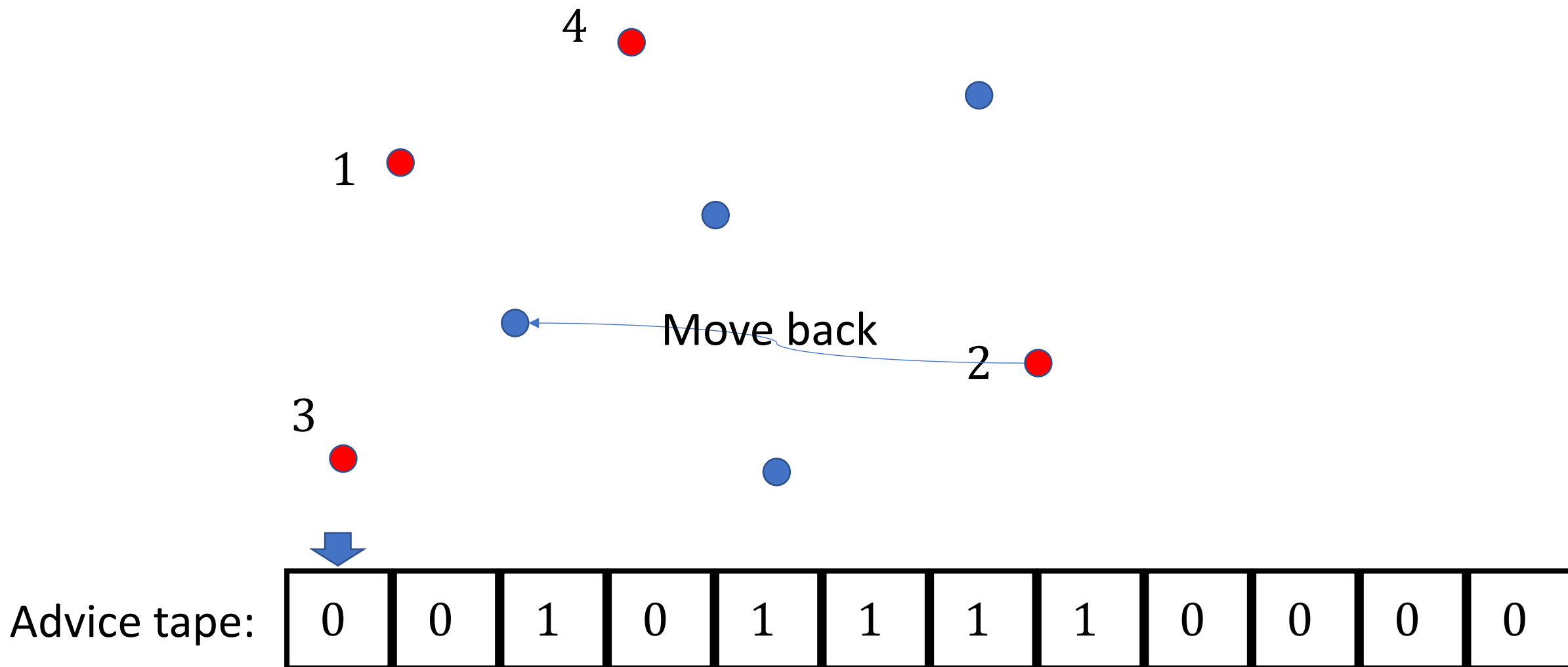
# *CHASE* example



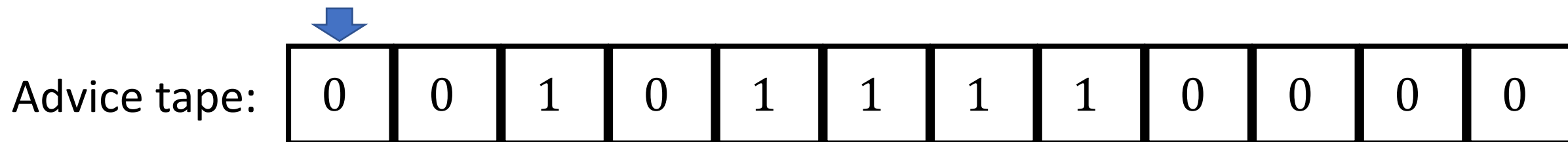
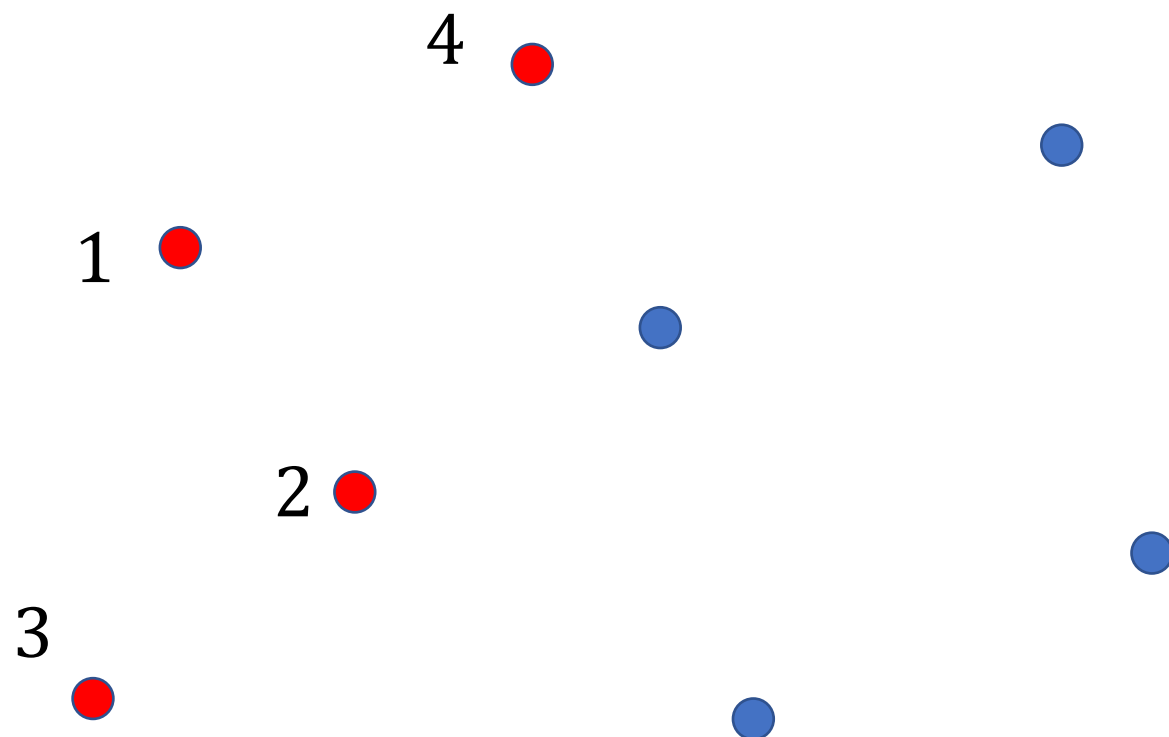
# *CHASE* example



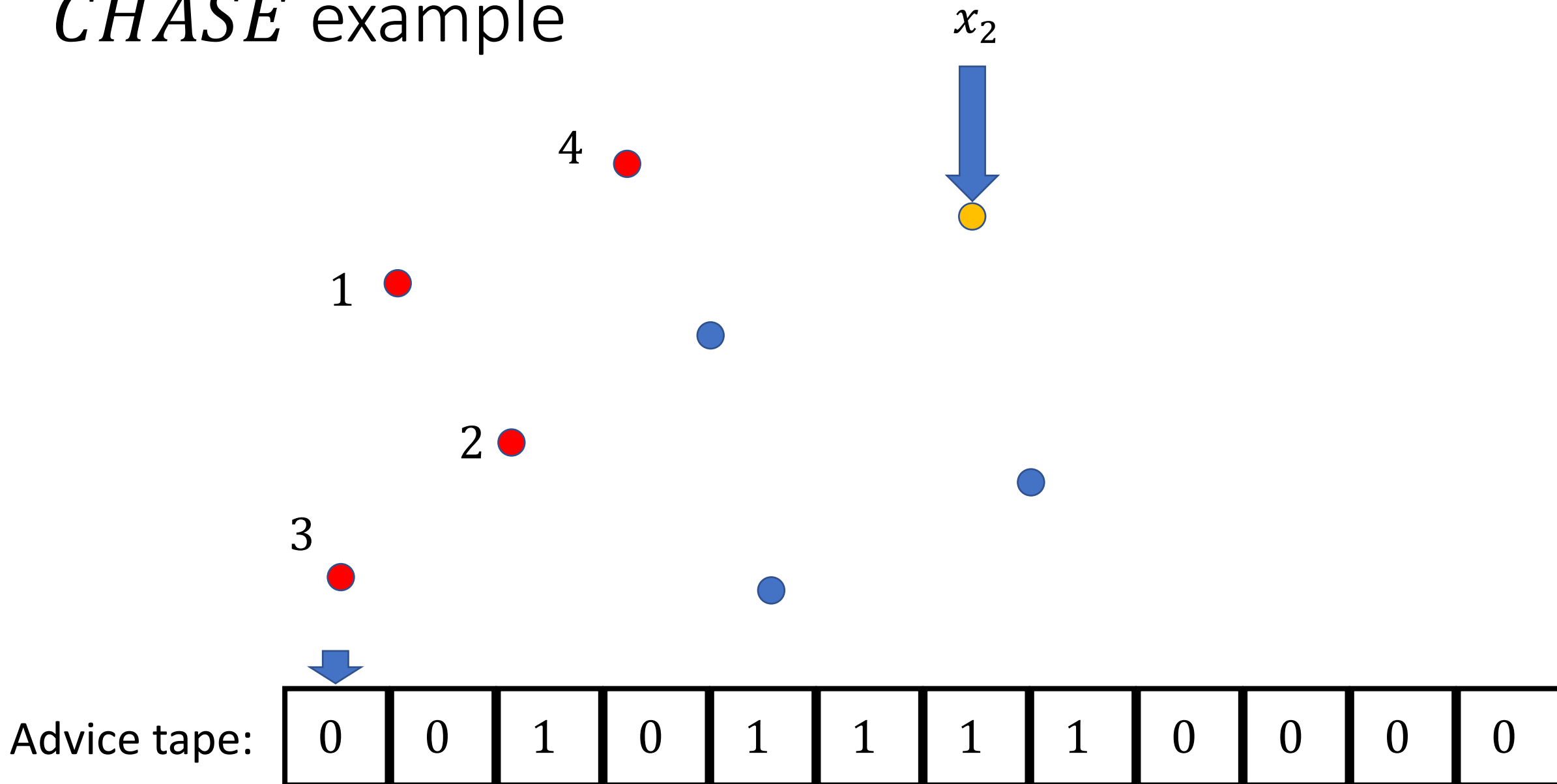
# *CHASE* example



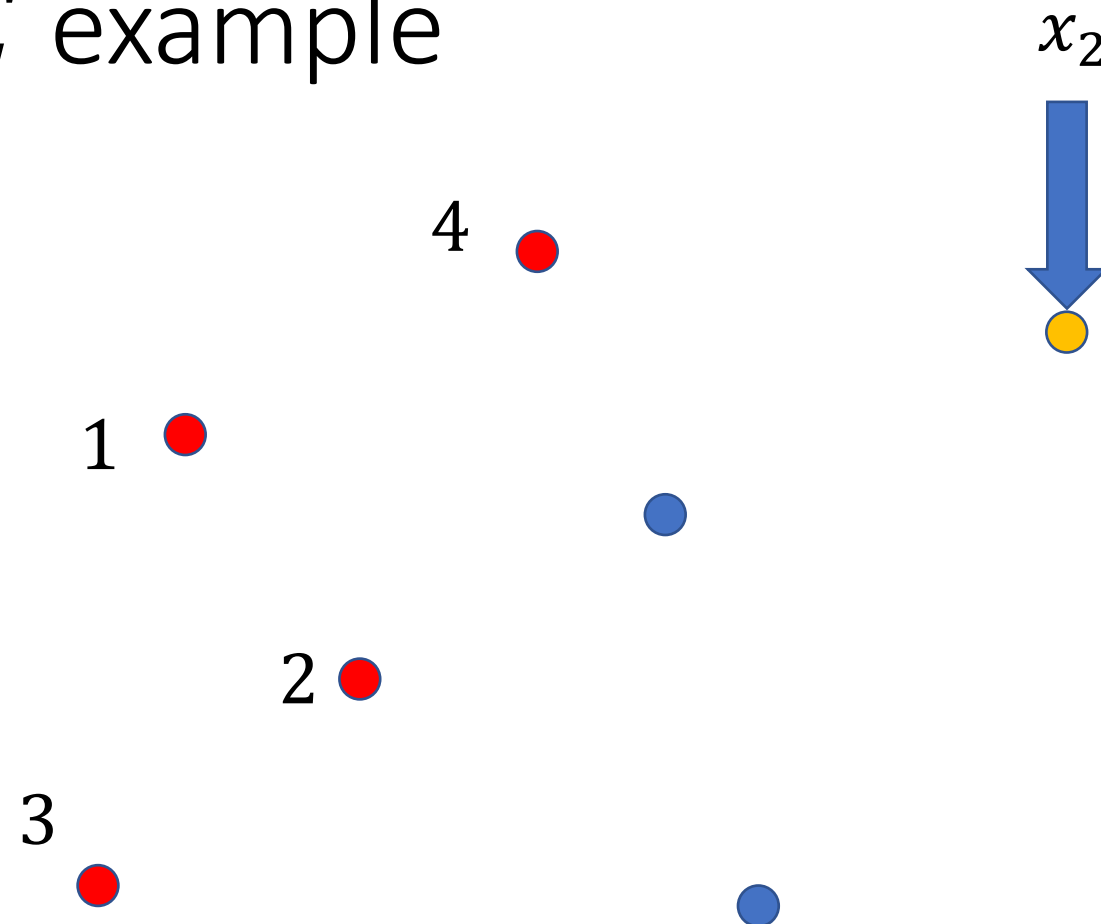
# *CHASE* example



# *CHASE* example



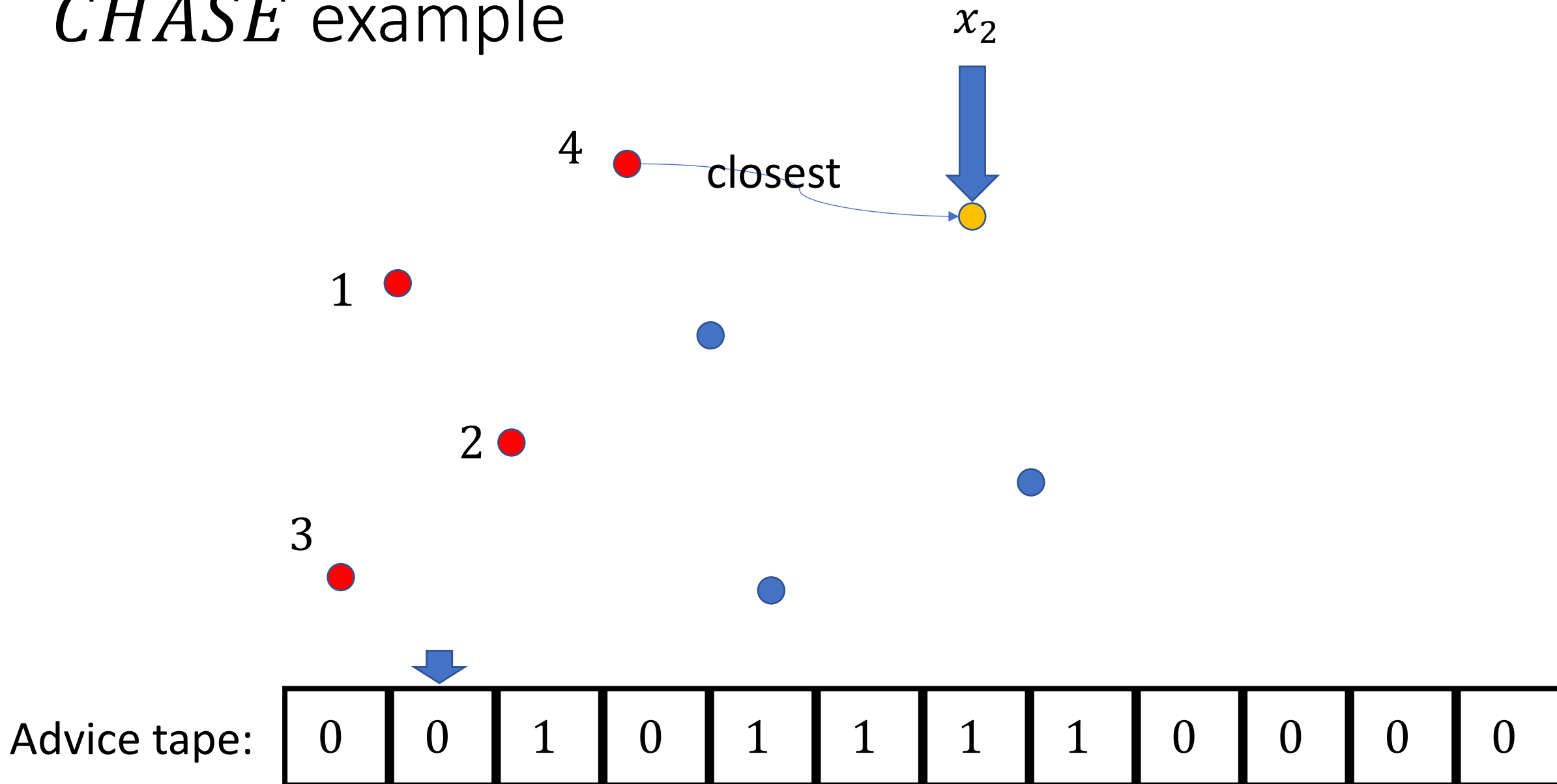
# *CHASE* example



Advice tape:

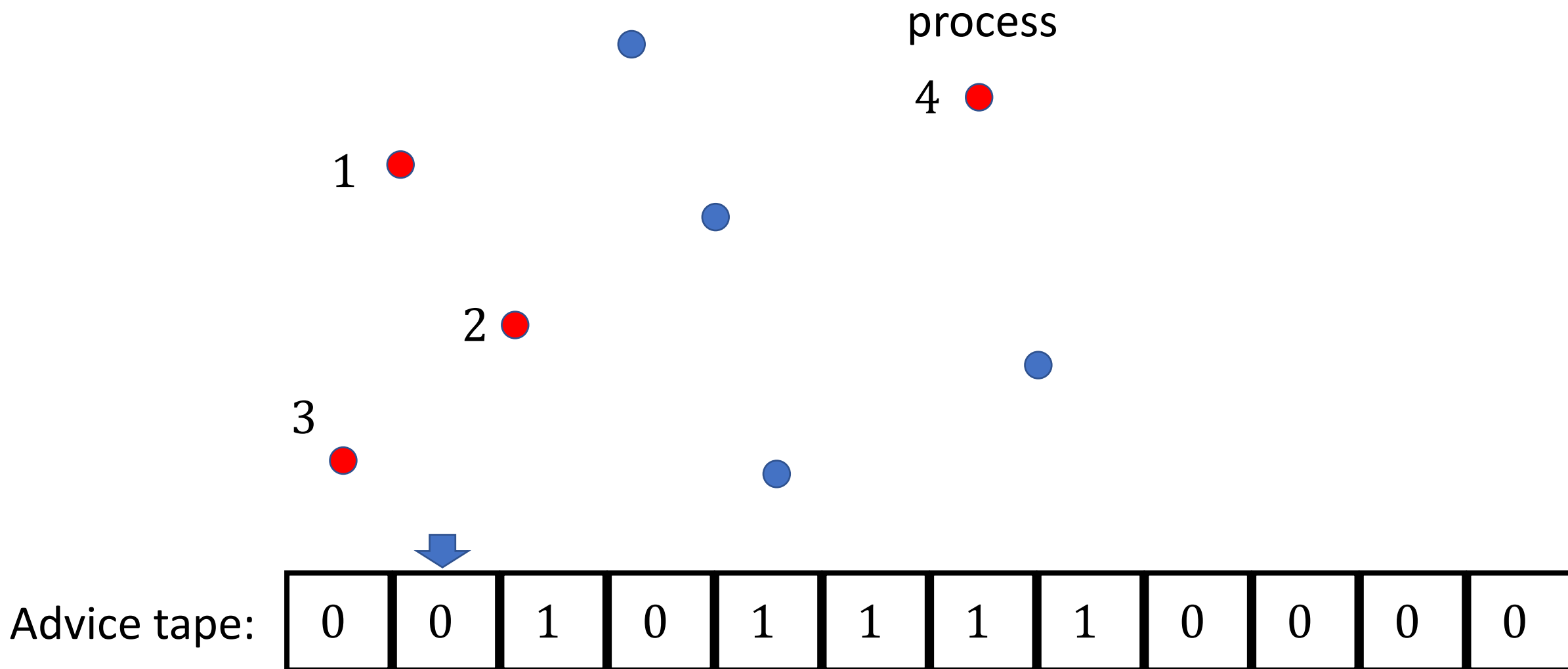
0	0	1	0	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---

# *CHASE* example

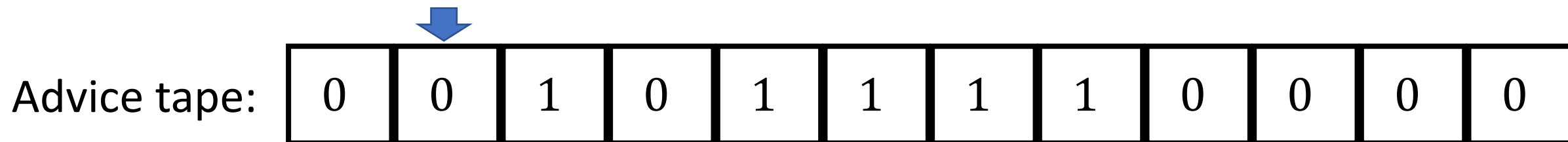
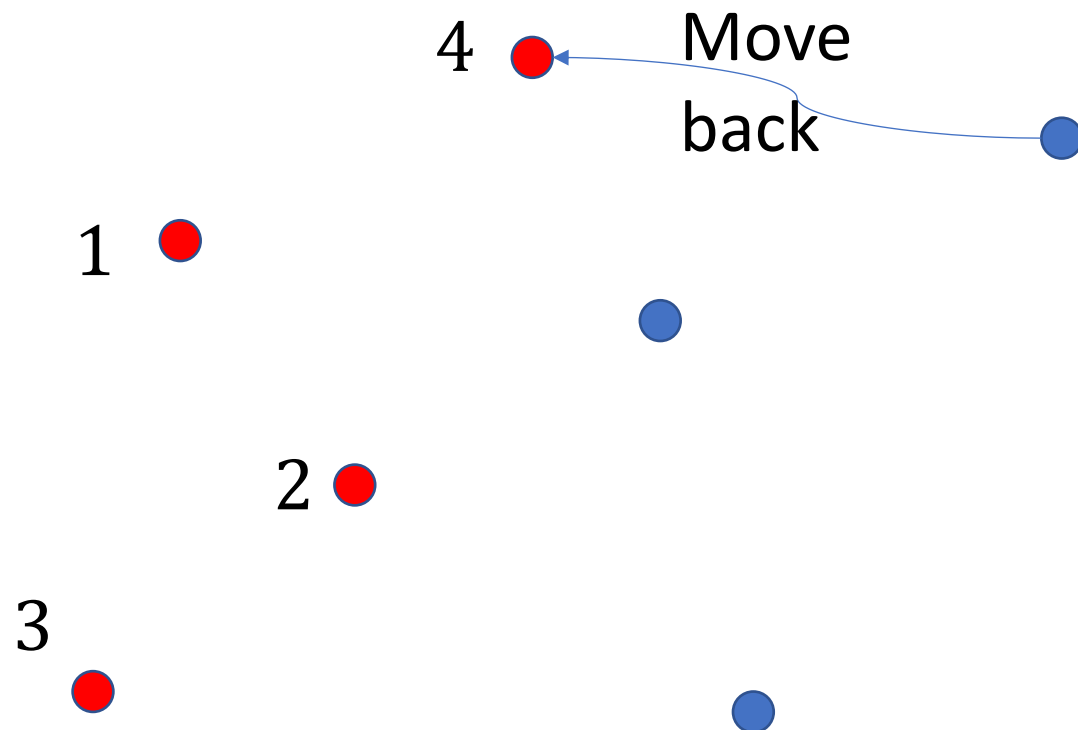




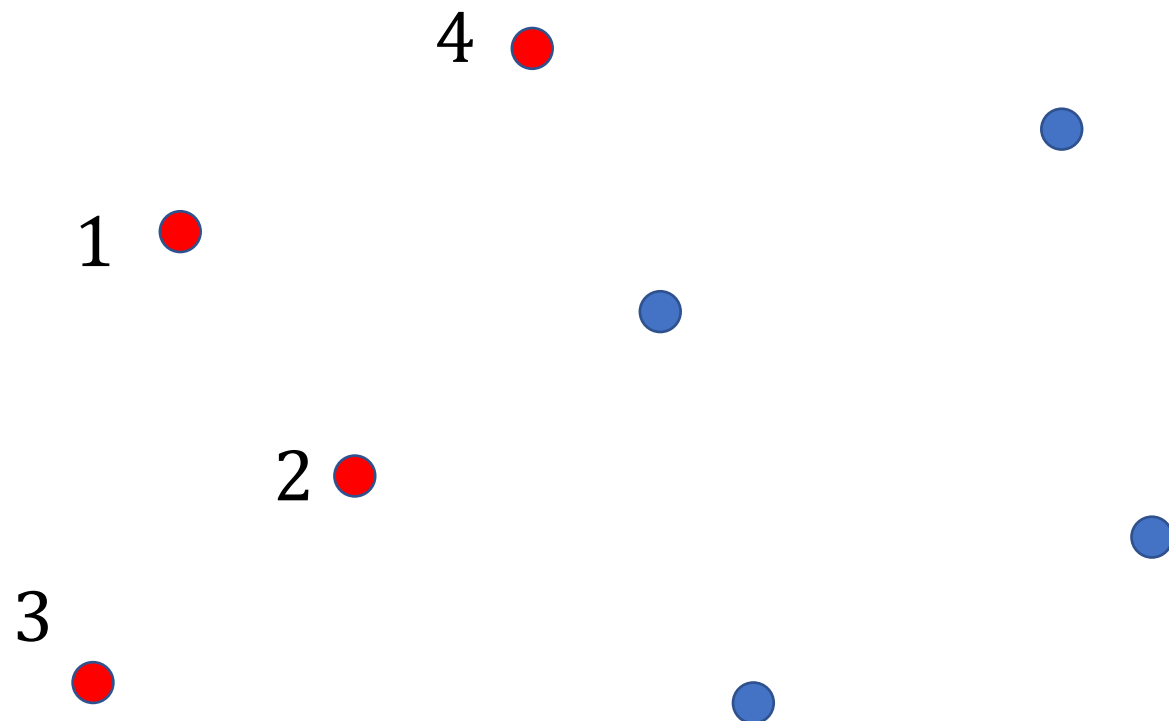
# *CHASE* example



# *CHASE* example



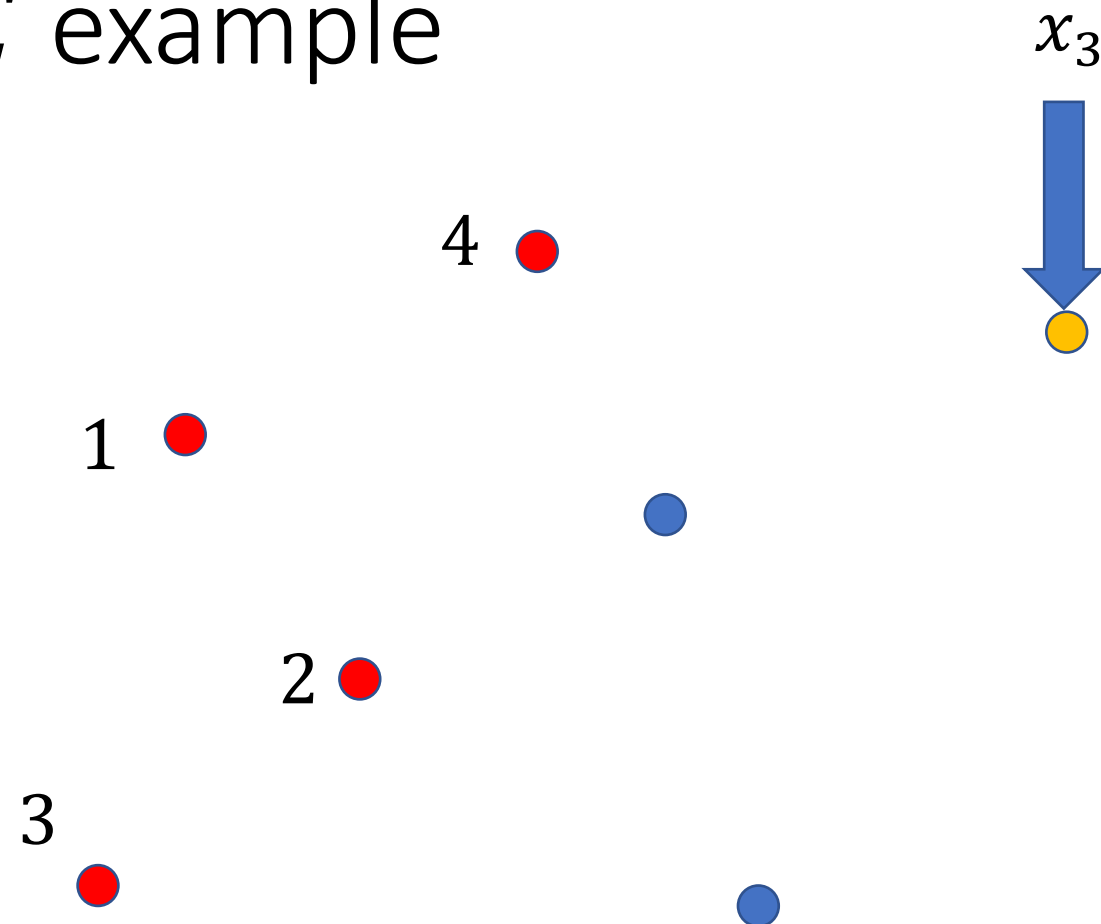
# *CHASE* example



Advice tape:

0	0	1	0	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---

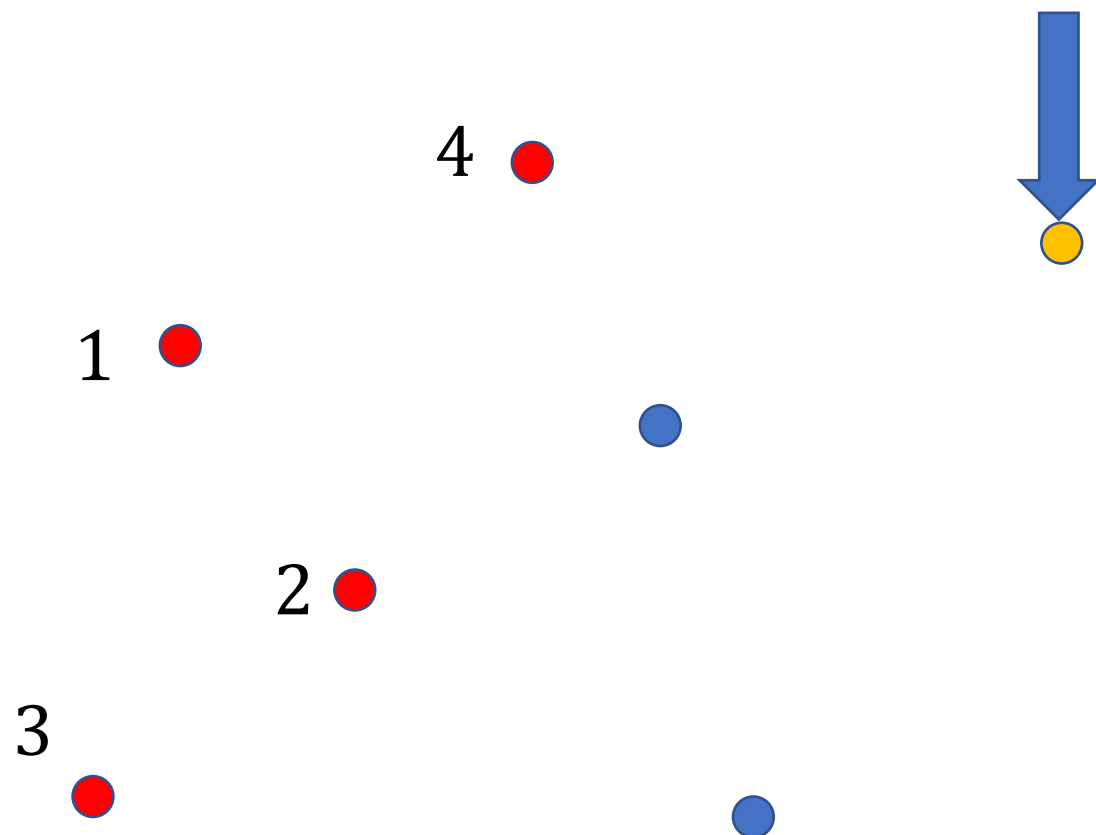
# *CHASE* example



Advice tape:

0	0	1	0	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---

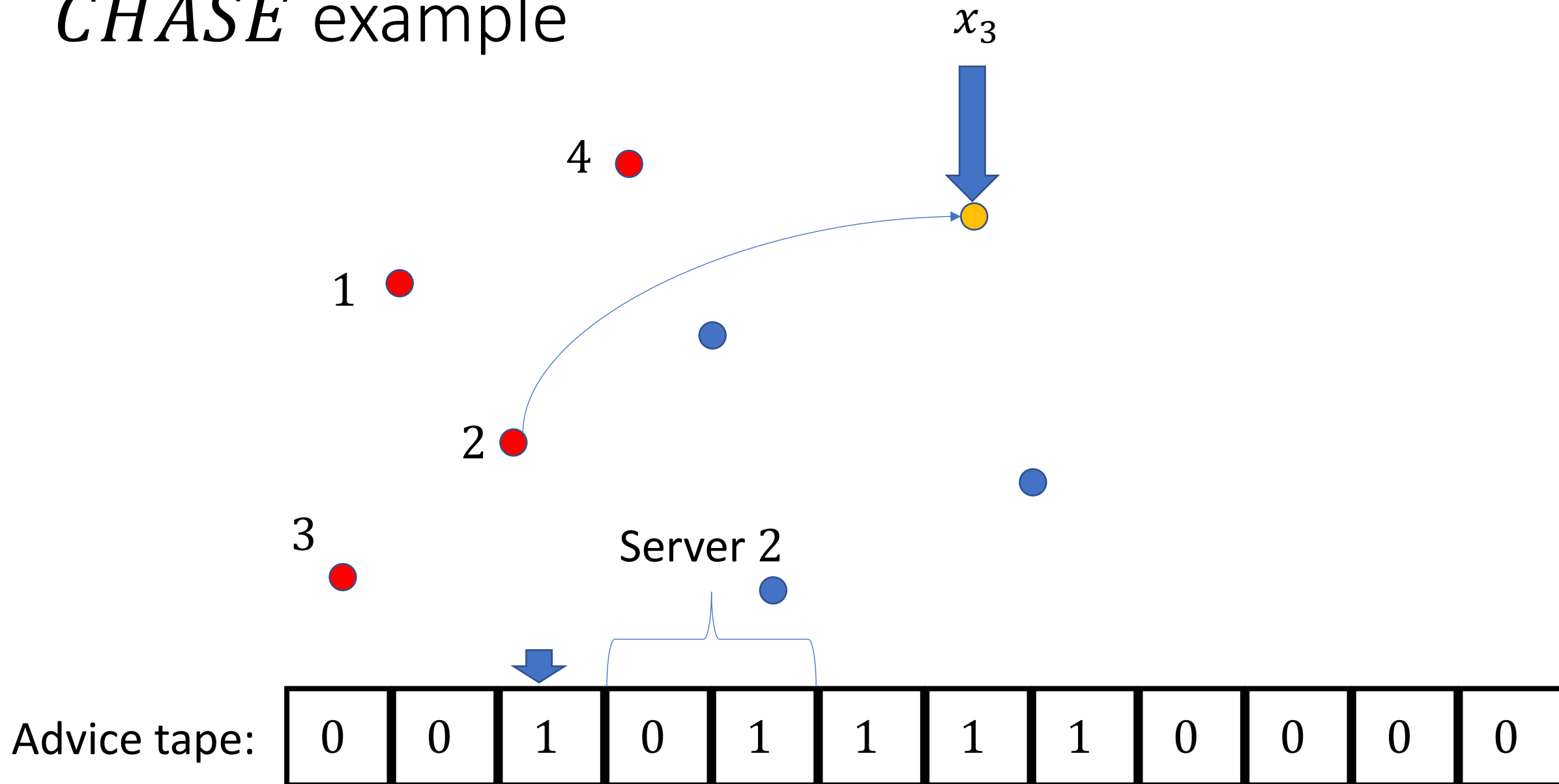
# *CHASE* example



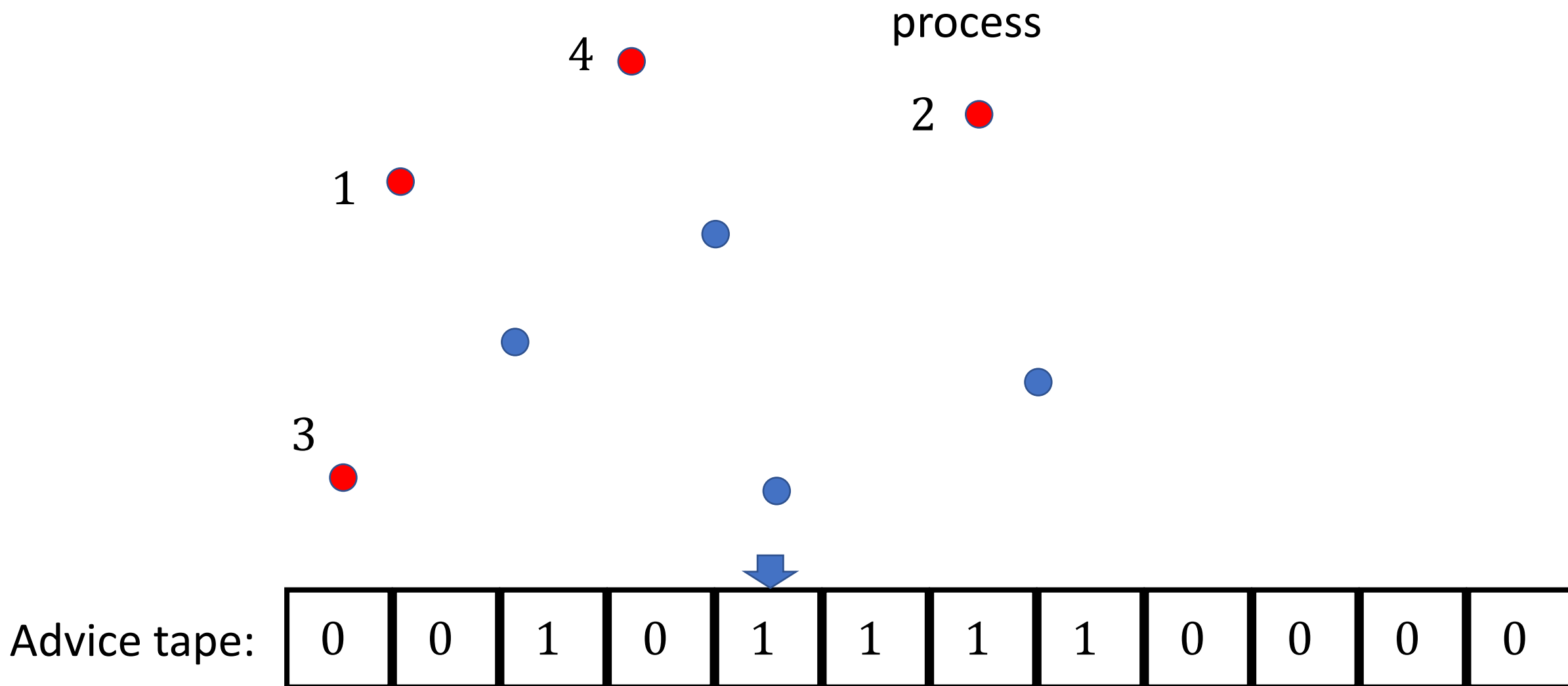
Advice tape:

0	0	1	0	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---

# *CHASE* example

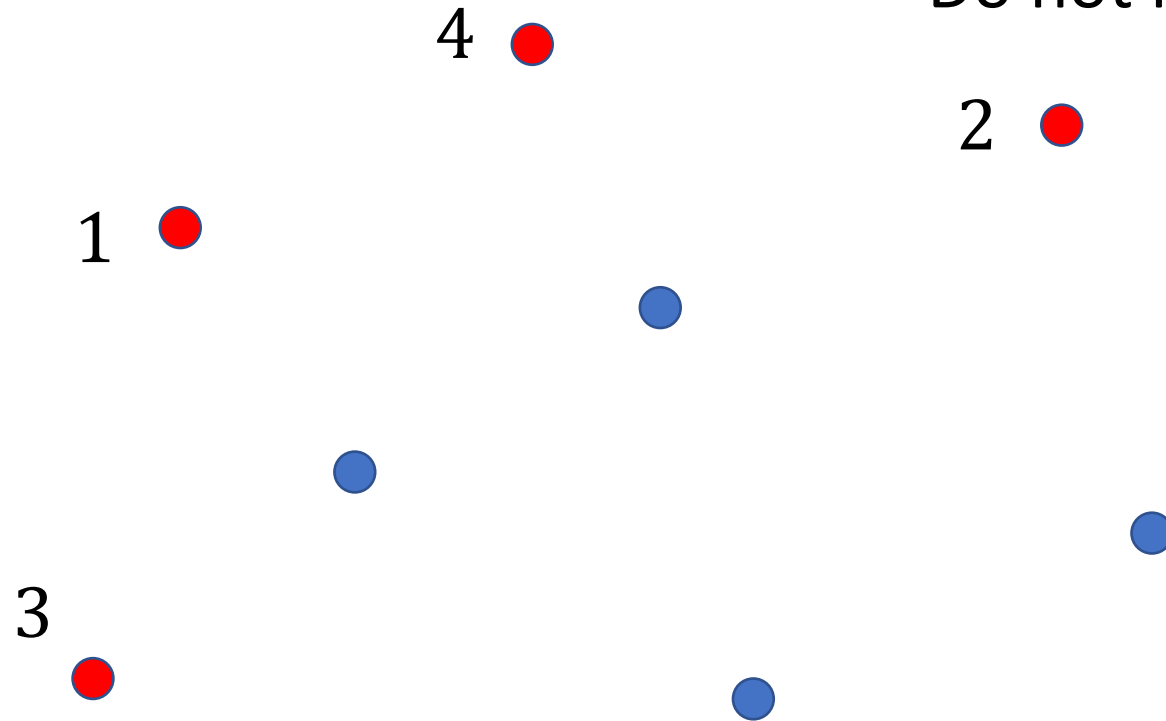


# *CHASE* example



# *CHASE* example

Do not move back!

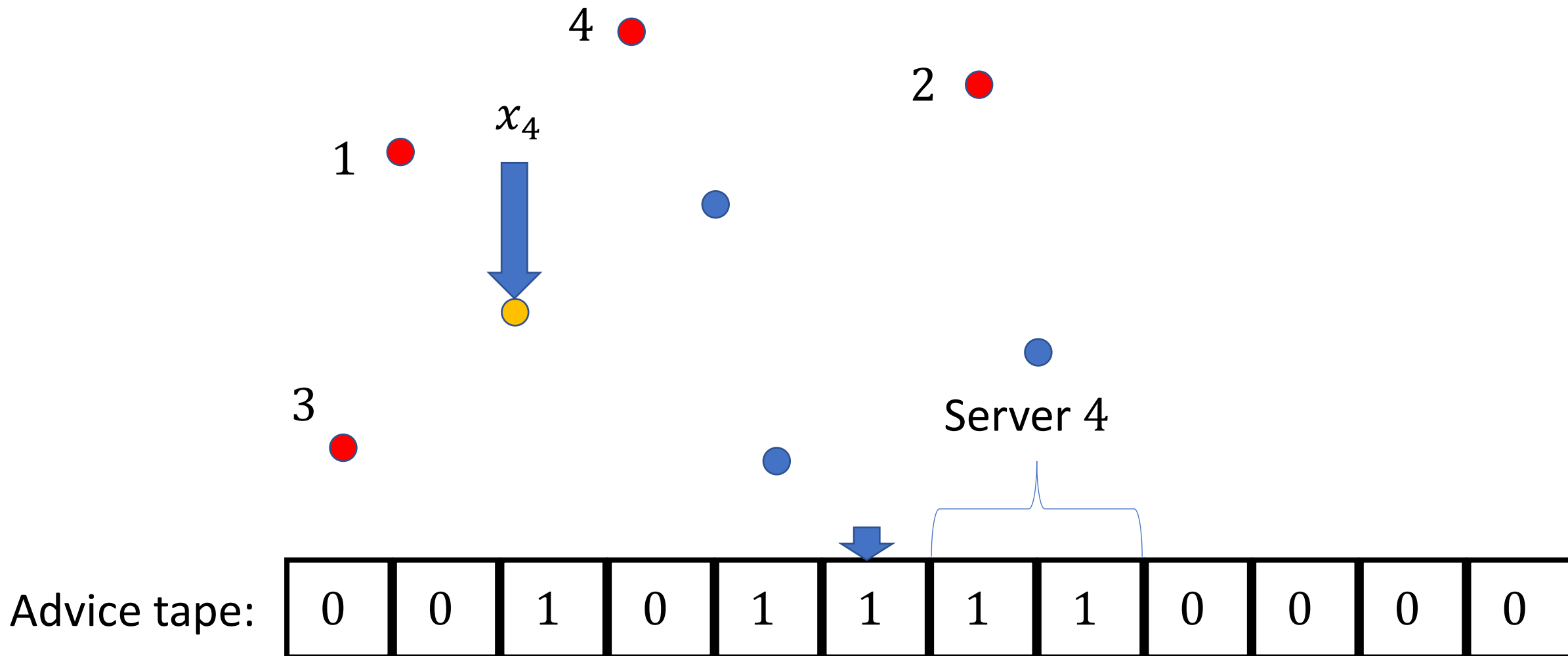


Advice tape:

0	0	1	0	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---



# *CHASE* example



# Analysis of *CHASE*

We analyze the behavior of *CHASE* on a cycle-by-cycle basis

Overall behavior is simply the sum over cycles

Let  $C = y_1, y_2, \dots, y_\alpha$  be a cycle in  $x^{(s)}$

Advice corresponding to  $y_i$  for  $i = 1, 2, \dots, \alpha - 1$  is a single bit 0

Advice corresponding to  $y_\alpha$  is  $1s$ , which takes  $1 + \log k = 1 + 2\alpha$  bits

Overall, length of advice corresponding to cycle  $C$  is

$$1 \cdot (\alpha - 1) + (1 + 2\alpha) \cdot 1 = 3\alpha$$

# Analysis of *CHASE*

To process cycle  $C$  with  $\alpha$  requests *CHASE* reads  $3\alpha$  bits of advice

Therefore, to process the entire input sequence

$$x_1, x_2, \dots, x_n$$

*CHASE* reads  $3n$  bits of advice

What about the competitive ratio?

# Analysis of *CHASE*

*OPT* uses only server  $s$  to process all requests in  $C$  (since  $C \subseteq x^{(s)}$ )

Also *OPT* is lazy and  $C$  is contiguous

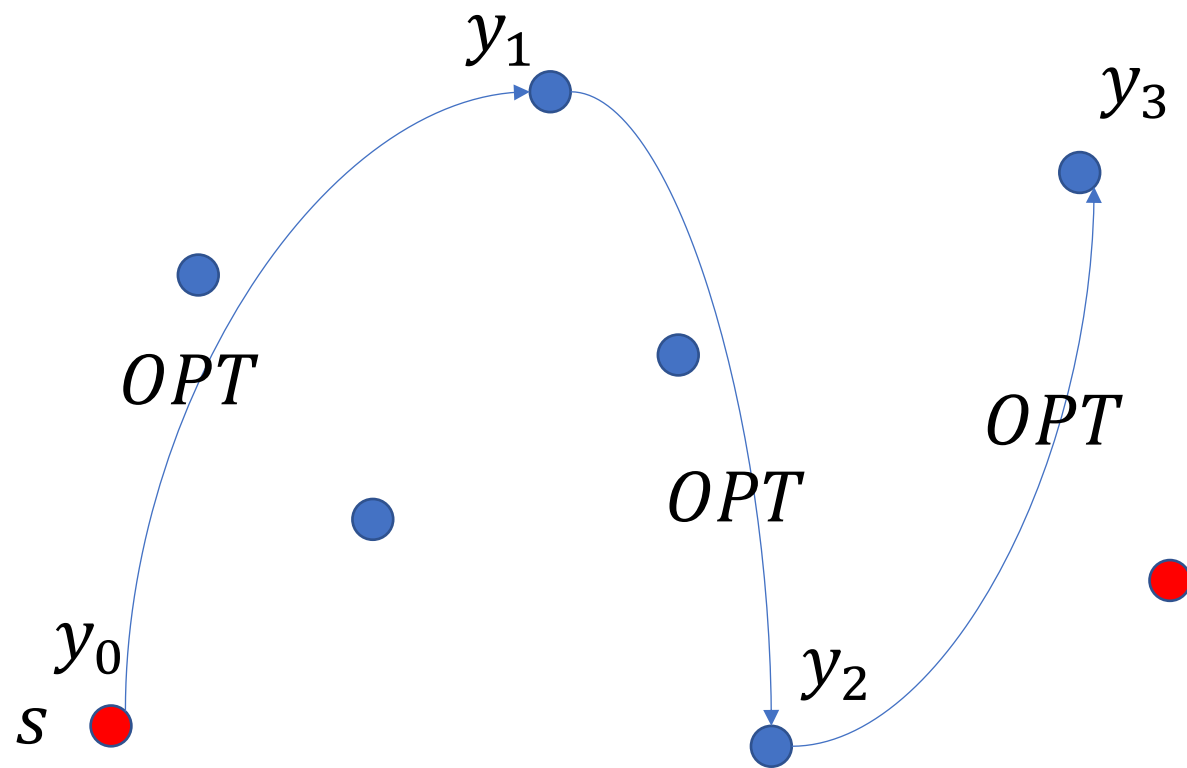
$$OPT(C) = \sum_{i=1}^{\alpha} d(y_{i-1}, y_i)$$

*OPT* and *CHASE* have server  $s$  located at  $y_0$  prior to servicing  $C$

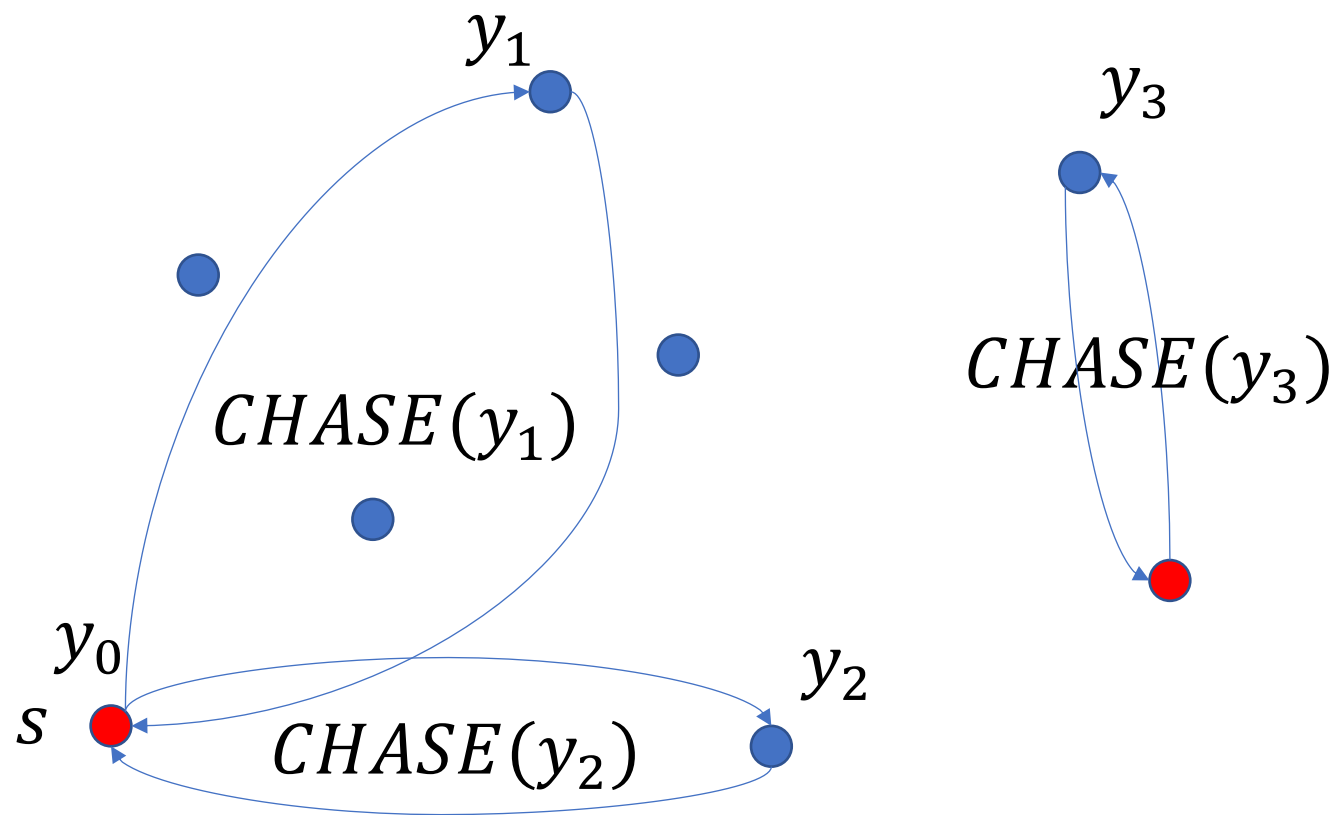
Then for  $i \leq \alpha - 1$

$$CHASE(y_i) \leq 2d(y_0, y_i)$$

# Analysis of *CHASE*



# Analysis of *CHASE*



# Analysis of *CHASE*

$$OPT(C) = \sum_{i=1}^{\alpha} d(y_{i-1}, y_i)$$

$$CHASE(y_i) \leq 2d(y_0, y_i)$$

$$CHASE(y_i) \leq 2(d(y_0, y_1) + d(y_1, y_i)) \text{ (triangle inequality)}$$

$$CHASE(y_i) \leq 2(d(y_0, y_1) + d(y_1, y_2) + d(y_2, y_i))$$

$$CHASE(y_i) \leq 2 \sum_{j=1}^i d(y_{j-1}, y_j) \leq 2 OPT(C)$$

# Analysis of *CHASE*

Lastly note that  $CHASE(y_\alpha) = d(y_0, y_\alpha) \leq \sum_{i=1}^{\alpha} d(y_{i-1}, y_i) = OPT(C)$

Therefore, we get

$$CHASE(C) = \sum_{i=1}^{\alpha} CHASE(y_i) \leq 2\alpha OPT(C) = (\log k) OPT(C)$$

QED



Upper bound  
technique: Adapting  
Randomized Algorithms

# Simple observation

*ALG*

- Uses  $b$  random bits
- Achieves competitive ratio  $\rho$  against oblivious adversary

implies there exists deterministic *ALG'* such that

- Uses  $b$  bits of advice
- Achieves competitive ratio  $\rho$

# Simple observation

This follows since oracle can always fix  $b$  random bits to particular values so as to achieve at least the average performance

This works best when  $b$  is small, as in barely random algorithms

What if a randomized algorithm uses an unspecified (and potentially very large) number of random bits?

We can still turn it into an algorithm with relatively short advice under some mild conditions – we will see this later!

Upper bound  
technique: Adapting  
Offline Algorithms

# Adapting offline algorithms

This technique amounts to examining various **offline algorithms** for the given problem and seeing how much advice is needed to convert them to **online algorithms**

For concreteness, we will study Makespan, which has an offline PTAS

An offline problem has a polynomial time approximation scheme (PTAS) if for every  $\epsilon > 0$  it can be approximated to within  $(1 + \epsilon)$  in polynomial time.

Polynomial can (and usually does) depend on  $\epsilon$ , e.g.,  $n^{\frac{1}{\epsilon}}$

# Makespan problem setting

You control several *identical* machines

Jobs arrive one by one

You are only concerned with a *processing* time of each job

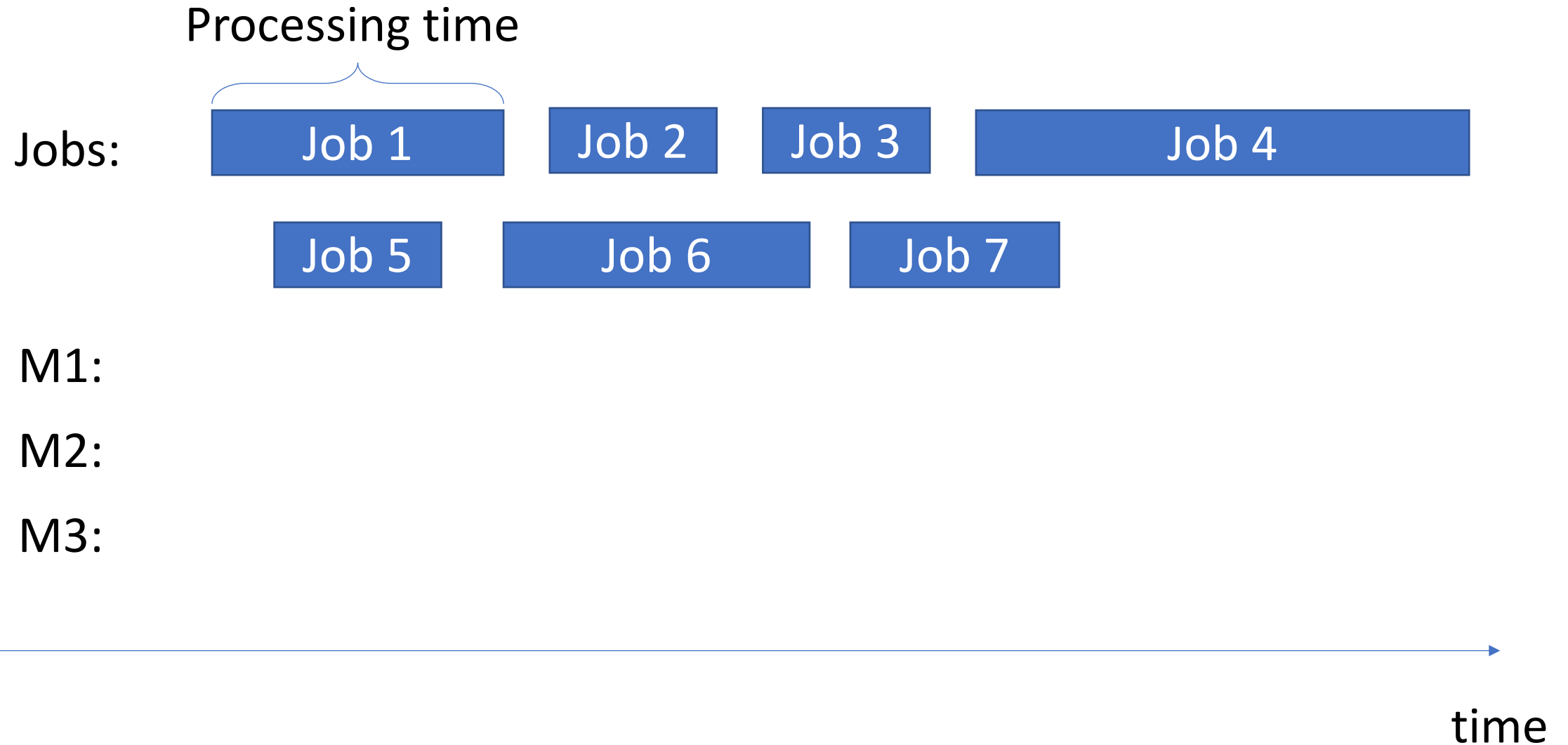
Schedule each job to be executed on one of the machines

Jobs are scheduled back to back on each machine

Goal: minimize longest *completion time* of a machine

Real-life examples: scheduling a project consisting of subtasks with multiple workers; running a cloud computing service on several servers; running a supercomputing server; etc.

# Makespan problem example:

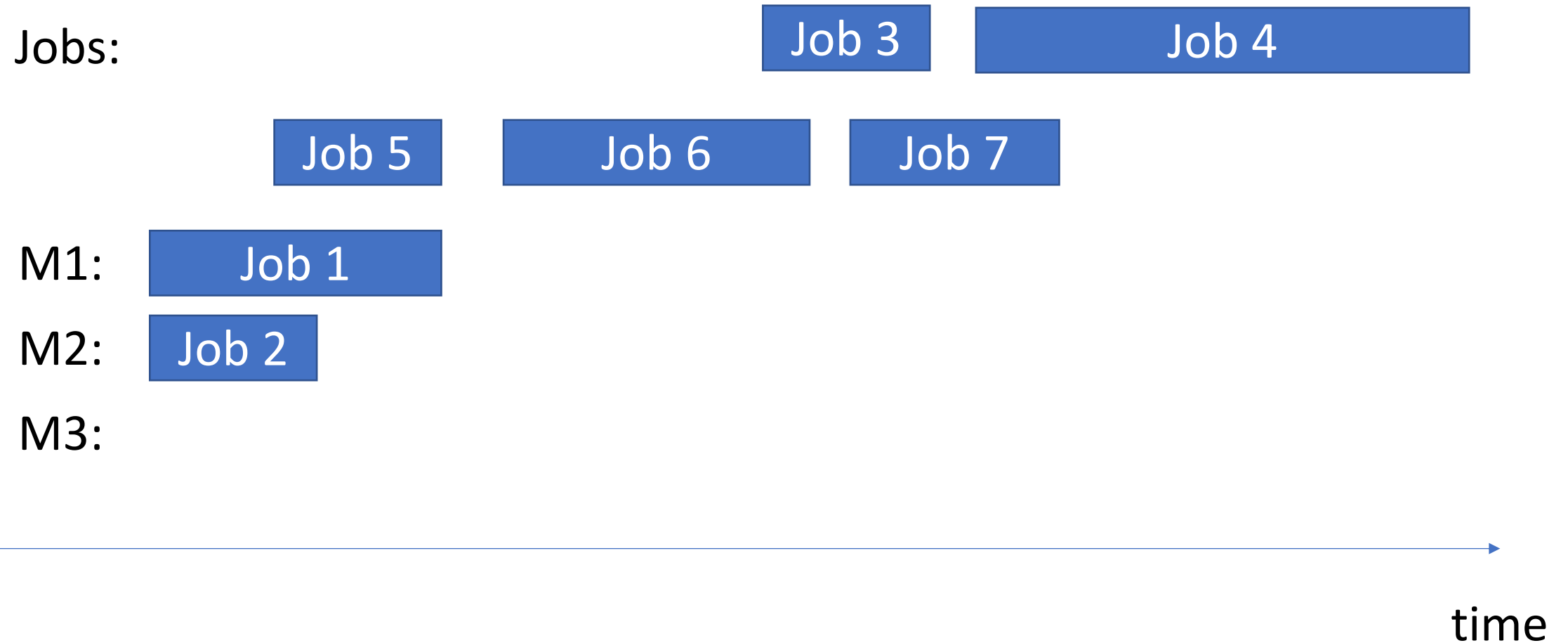


# Makespan problem example:

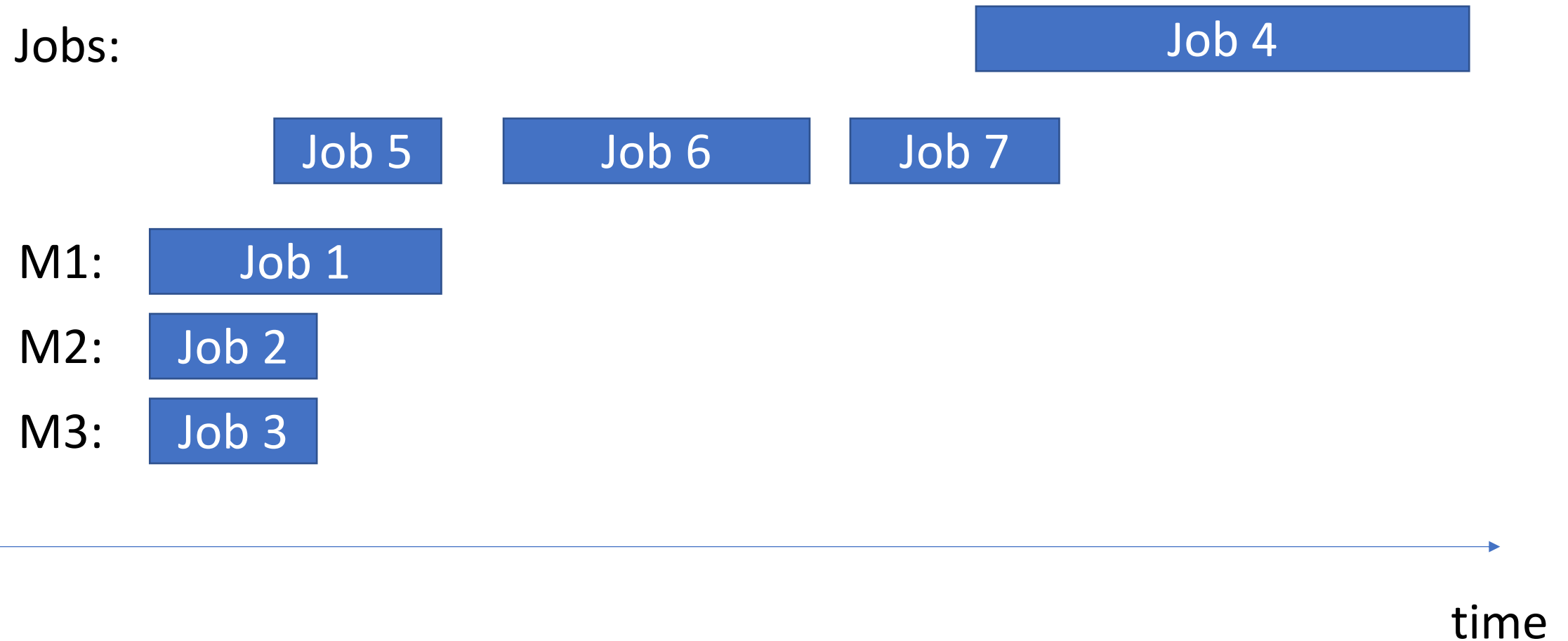




# Makespan problem example:

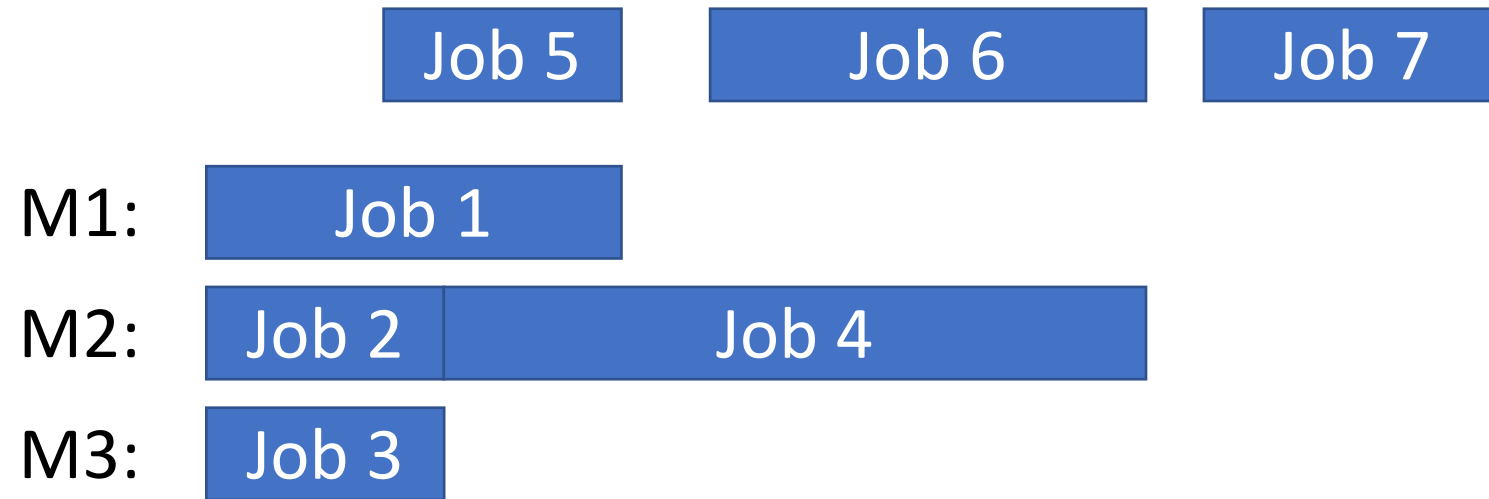


# Makespan problem example:



# Makespan problem example:

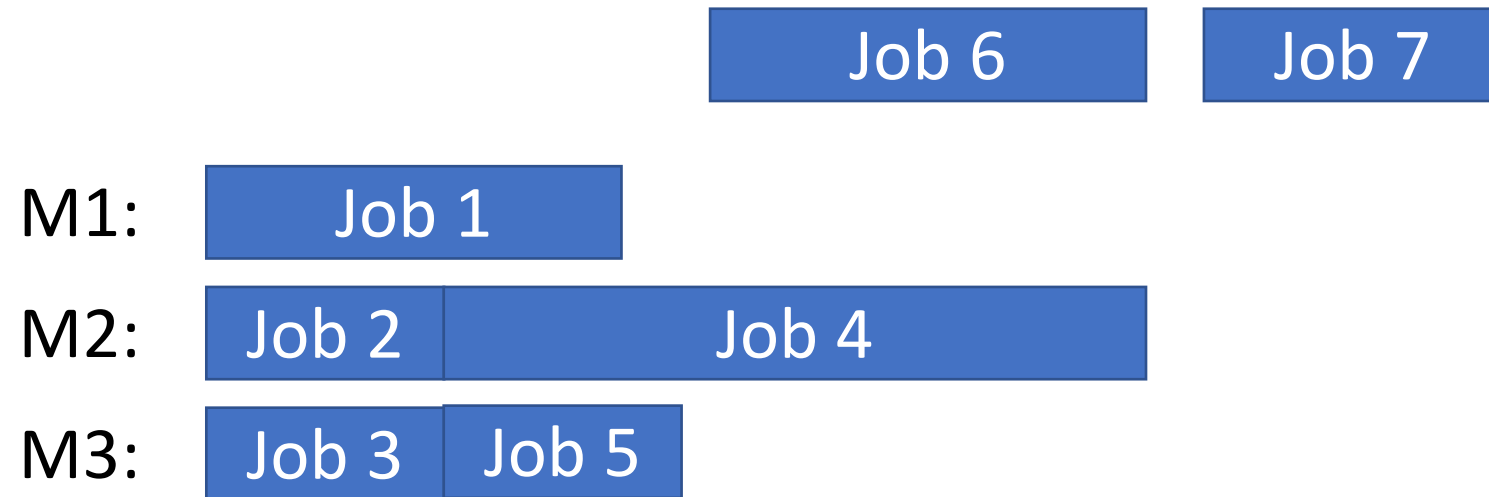
Jobs:



time

# Makespan problem example:

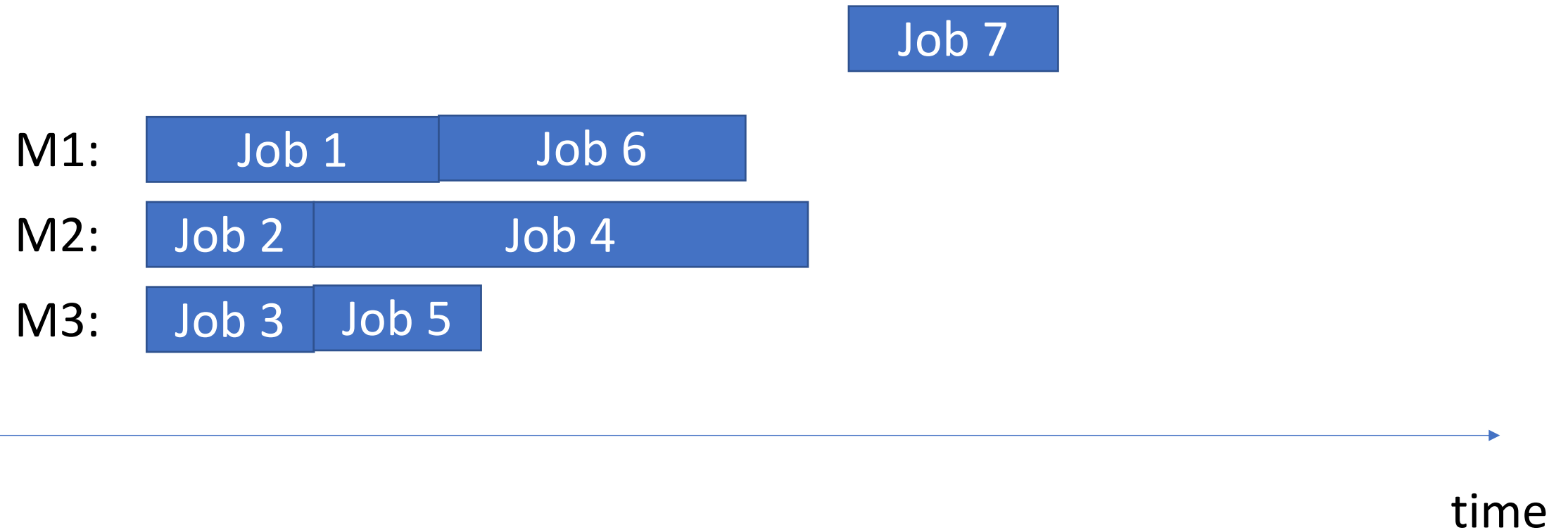
Jobs:



time

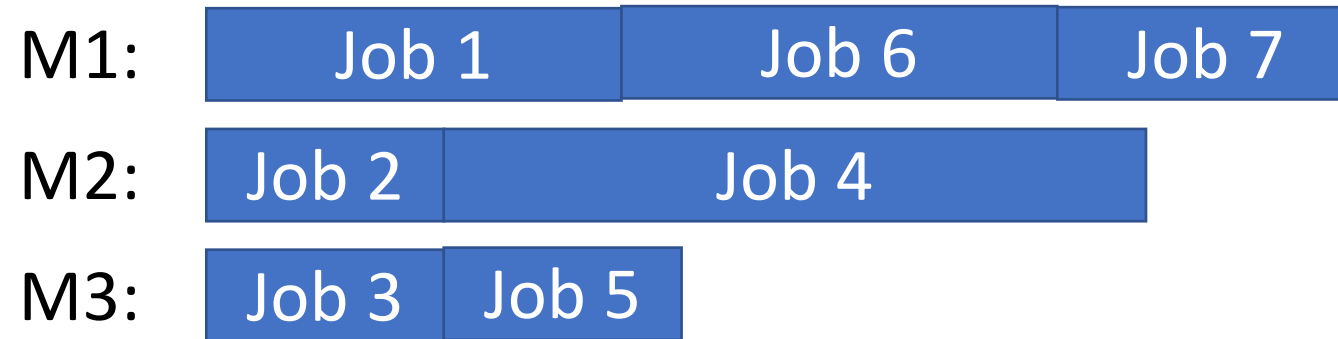
# Makespan problem example:

Jobs:



# Makespan problem example:

Jobs:



time

# Makespan problem example: sample schedule

Jobs:

Makespan

Longest completion time

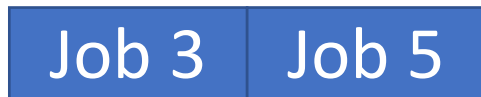
M1:



M2:



M3:



time

# Makespan

input sequence:  $p_1, p_2, \dots, p_n$ ,

where  $p_i \in \mathbb{N}$  is the processing time of job  $i$

$m$  – number of machines

Two optimal advice strategies:

(1) encode the entire input sequence on advice tape, which requires

$$\sum_i \lceil \log p_i \rceil \leq \lceil \log OPT \rceil n \text{ bits of advice}$$

(2) Encode which machine  $OPT$  schedules each job on, which requires

$$n \log m \text{ bits of advice}$$



# Makespan

Goal: design an online algorithm with advice that achieves “near” optimality and uses significantly less advice than  $\min(n \log OPT, n \log m)$

In the regime where  $\log OPT$  and  $\log m$  are *small* compared to  $n$

We shall design an algorithm that for every  $\epsilon > 0$

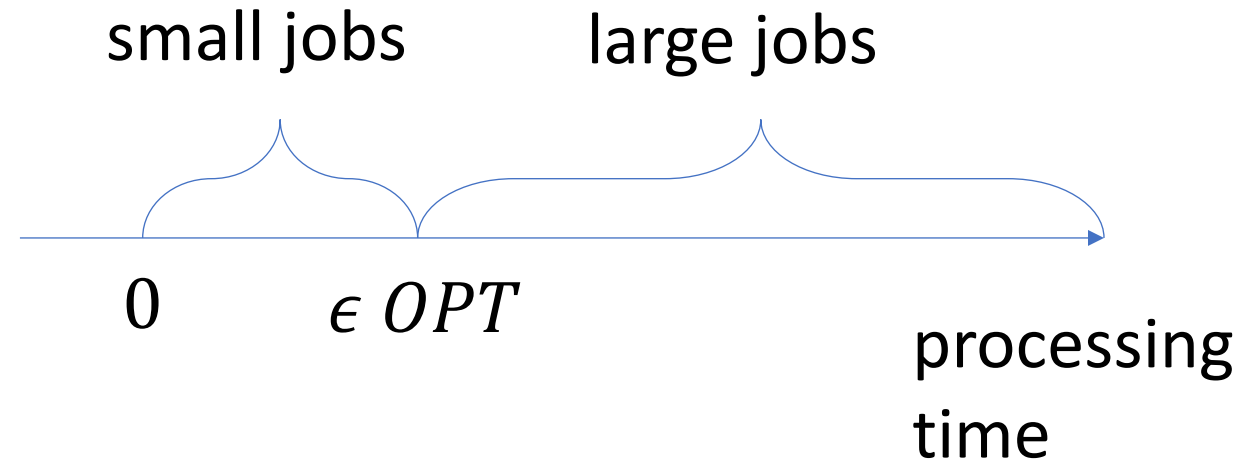
- It achieves competitive ratio  $(1 + \epsilon)$
- It uses advice of length  $\log OPT + O_\epsilon(\log n)$

# Classical **OFFLINE** PTAS for Makespan

Fix  $\epsilon > 0$

Call a job  $p_j$  **small** if  $p_j < \epsilon OPT$

Call a job  $p_j$  **large** if  $p_j \geq \epsilon OPT$



PTAS works in 2 phases:

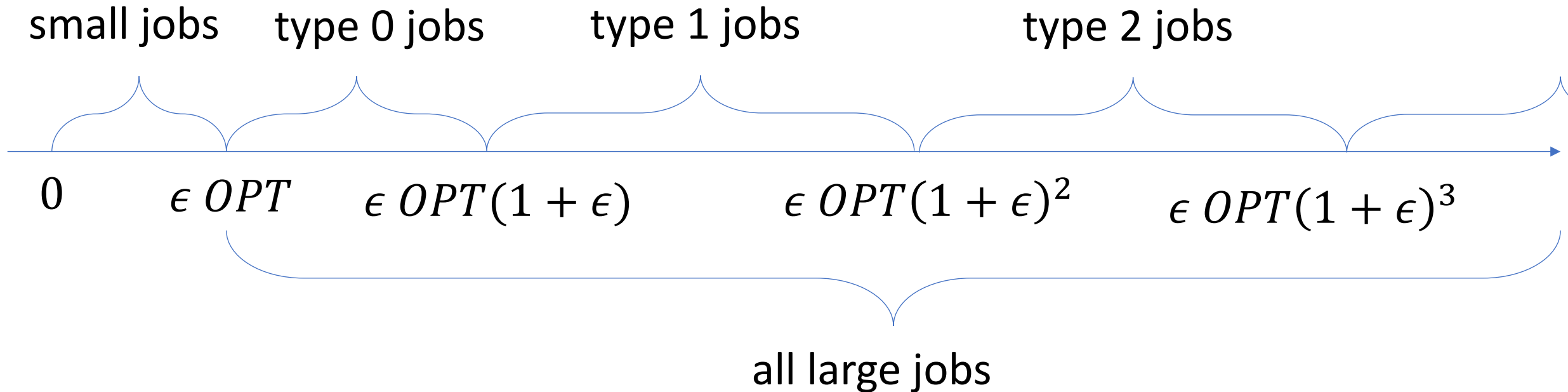
Phase 1: handle all large jobs

Phase 2: handle all small jobs

# High level overview

Phase 1:

subdivide all large jobs into further types depending on size



# High level overview

Phase 1:

subdivide all large jobs into further types depending on size

Job  $j$  has type  $i$  if

$$p_j \in [\epsilon OPT(1 + \epsilon)^i, \epsilon OPT(1 + \epsilon)^{i+1})$$

Smallest type is  $i = 0$  corresponding to interval  $[\epsilon OPT, \epsilon OPT(1 + \epsilon))$

Largest type is  $i = \log_{1+\epsilon} \frac{1}{\epsilon}$  since

$$\epsilon OPT (1 + \epsilon)^{\log_{1+\epsilon} \frac{1}{\epsilon}} = \epsilon OPT \frac{1}{\epsilon} = OPT$$

# High level overview

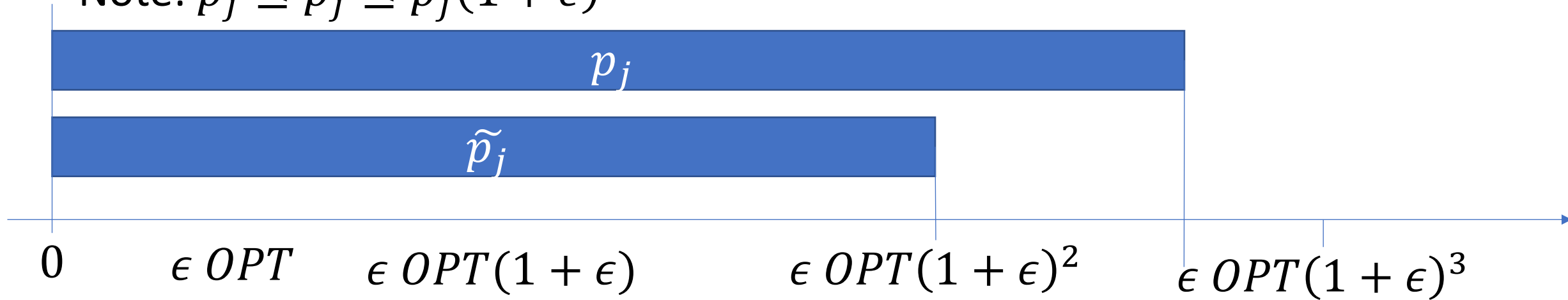
Phase 1:

subdivide all large jobs into further types depending on size

Round each job  $p_j$  down to the lower end of the interval corresponding to the type of job  $j$

Let  $\tilde{p}_j$  denote the rounded processing times

Note:  $\tilde{p}_j \leq p_j \leq \tilde{p}_j(1 + \epsilon)$



# High level overview

## Phase 1:

- subdivide all large jobs into further types depending on size
- round each job  $p_j$  down to the lower end  $\tilde{p}_j$  of the interval corresponding to the type of job  $j$
- solve Makespan optimally with respect to rounded large times  $\tilde{p}_j$ 
  - This can be done by dynamic programming: if the number of **distinct** processing times is  $k$  then Makespan can be solved optimally in time  $n^{O(k)}$
  - We have  $k = \log_{1+\epsilon} \frac{1}{\epsilon}$  **distinct** rounded weights
- Substitute the rounded time  $\tilde{p}_j$  with the actual time  $p_j$  in the above schedule

# High level overview

Phase 2:

- assign all small jobs greedily (use least loaded machine each time)

# What happens with makespan?

$OPT$  – the minimum optimal makespan with respect to all jobs (small and large)

Phase 1:

Makespan with respect to rounded job times  $\tilde{p}_j \leq OPT$  (why?)

Substituting  $p_j$  instead of  $\tilde{p}_j$  extends makespan to at most  $(1 + \epsilon)OPT$  (why?)



# What happens with makespan?

After phase 1 all large jobs have been scheduled with makespan  
 $\leq (1 + \epsilon)OPT$

Suppose we finish phase 2, let  $i$  denote the machine index that defines the makespan

If this machine has no small jobs then makespan  $\leq (1 + \epsilon)OPT$

Otherwise let  $j$  denote the last small job

By greediness, makespan is at most

$$p_j + \frac{1}{k} \sum_{\ell} p_{\ell} \leq \epsilon OPT + OPT = (1 + \epsilon)OPT$$

# PTAS for Makespan

We can get approximation  $(1 + \epsilon) OPT$  in time  $n^{O(\log_{1+\epsilon} \frac{1}{\epsilon})}$  offline

How to turn it into an online algorithm with advice?

Oracle writes down on advice tape:

- $OPT$
- The number of small jobs
- For each type  $i \in \{0, 1, \dots, \log_{1+\epsilon} \frac{1}{\epsilon}\}$  the number of large items of type  $i$

# Advice algorithm for Makespan

Oracle writes down on advice tape:

- $OPT$   $\log OPT$  bits
- The number of small jobs  $\log n$  bits
- For each  $i \in \{0, 1, \dots, \log_{1+\epsilon} \frac{1}{\epsilon}\}$  the number of large items of type  $i$

$$\left(\log_{1+\epsilon} \frac{1}{\epsilon} + 1\right) \log n \text{ bits}$$

Overall advice length:  $\log OPT + \left(\log_{1+\epsilon} \frac{1}{\epsilon} + 2\right) \log n =$   
 $\log OPT + O_{\epsilon}(\log n)$

# Advice algorithm for Makespan

What does online algorithm do with this advice?

Algorithm knows  $\tilde{p}_j$  for every large job  $j$  since

$$\tilde{p}_j = \epsilon OPT (1 + \epsilon)^i$$

if job  $j$  has type  $i$

Thus, the online algorithm can create an offline optimal schedule with respect to  $\tilde{p}_j$  before it even sees any input

# Advice algorithm for Makespan

## **Preprocessing:**

uses advice to create an optimal offline schedule  $S$  for  $\tilde{p}_j$

## **Processing online item $p_j$ :**

if  $p_j$  is large

    schedule job  $j$  according to  $S$  instead of  $\tilde{p}_j$

else

    schedule job  $j$  greedily assuming all large jobs have been scheduled according to  $S$  and with processing times  $\tilde{p}_j$

# Advice algorithm for Makespan

This online algorithm is similar to the offline PTAS for Makespan

What is the difference?

In the offline PTAS we schedule small jobs only after all large jobs have been scheduled – Phase 2

In the online advice algorithm small jobs can arrive before all large jobs have arrived. So we pretend that all large jobs have rounded weights and then we pretend they all have arrived. We schedule small jobs on top of that. – Modified Phase 2

# Advice algorithm for Makespan

Thus, analysis of competitive ratio of advice algorithm is similar

Using real weights instead of rounded weights leads to makespan  $\leq (1 + \epsilon)OPT$  – Phase 1

For Modified Phase 2, let  $i$  denote the machine defining makespan

If there are no small jobs on that machine then makespan  $\leq (1 + \epsilon)OPT$

Otherwise, let  $p_j$  be such a job. Then by greediness, the makespan is  $\leq$

$$\begin{aligned} p_j + \frac{1}{k} \left( \sum_{\ell: \text{small}} p_\ell + \sum_{\ell: \text{large}} \widetilde{p}_\ell \right) &\leq p_j + \frac{1 + \epsilon}{k} \sum_{\ell} p_\ell \\ &\leq \epsilon OPT + (1 + \epsilon)OPT \leq (1 + 2\epsilon)OPT \end{aligned}$$

# Summary

We showed that for every  $\epsilon > 0$  there is an online algorithm that achieves competitive ratio

$$(1 + 2\epsilon)OPT$$

and uses

$$\log OPT + O_\epsilon(\log n)$$

bits of advice in the tape advice model.