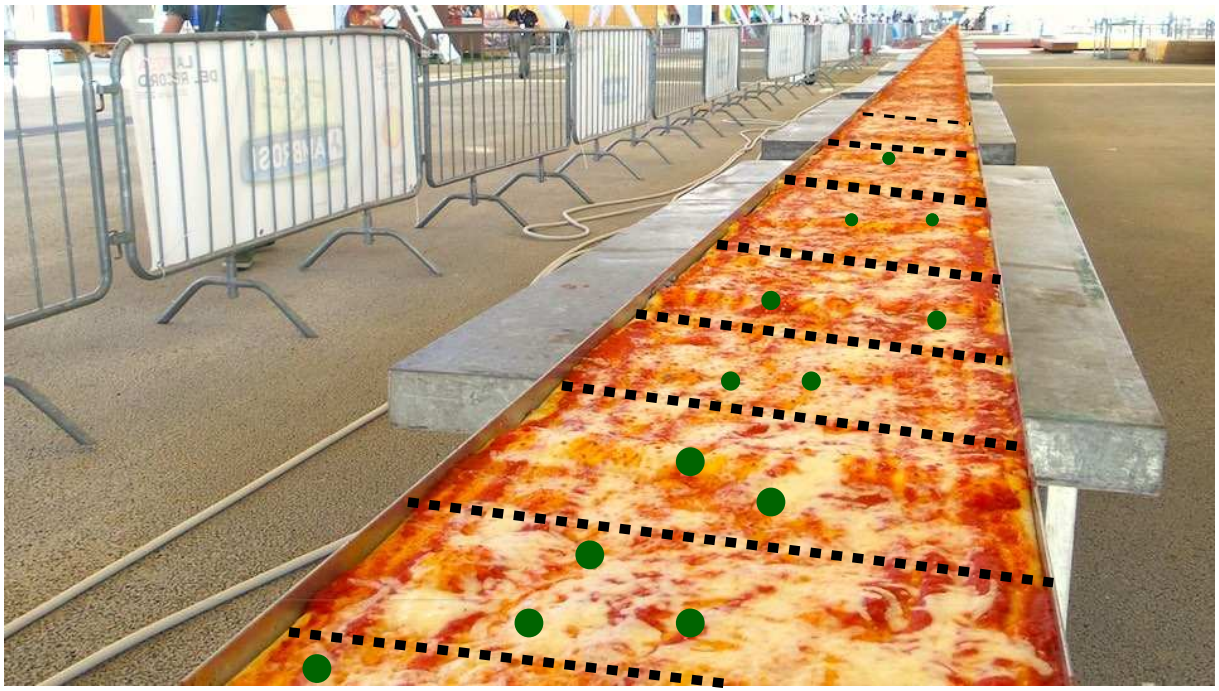
Gustavo's Pizza

Gustavo has very peculiar tastes when it comes to pizza al taglio: he wants his *slice* to have as many olives as possible, but never more than k.

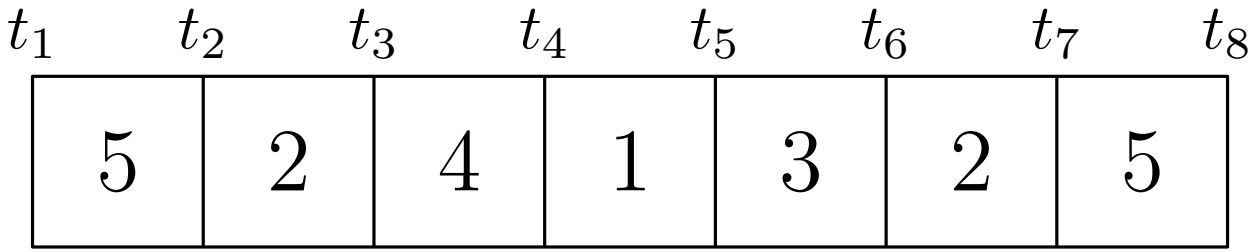
The pizza can be cut into discrete positions $t_1 < \cdots < t_n$. A *slice* (i, j) with $j > i$ represents the interval $[t_i, t_j]$.



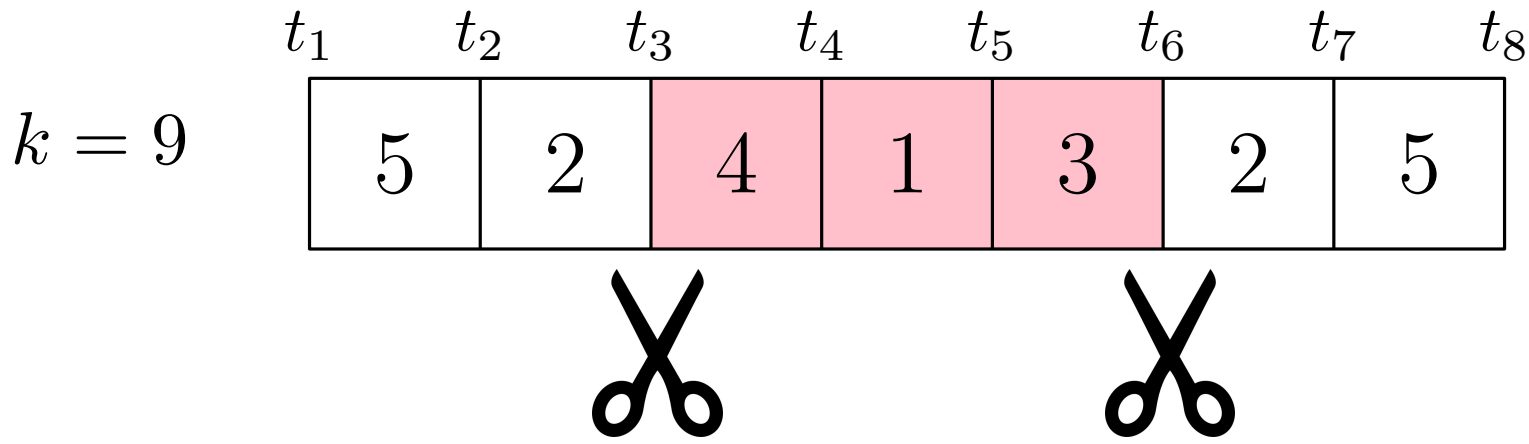
The interval $[t_i, t_{i+1}]$ contains η_i olives. Where to cut?

Example

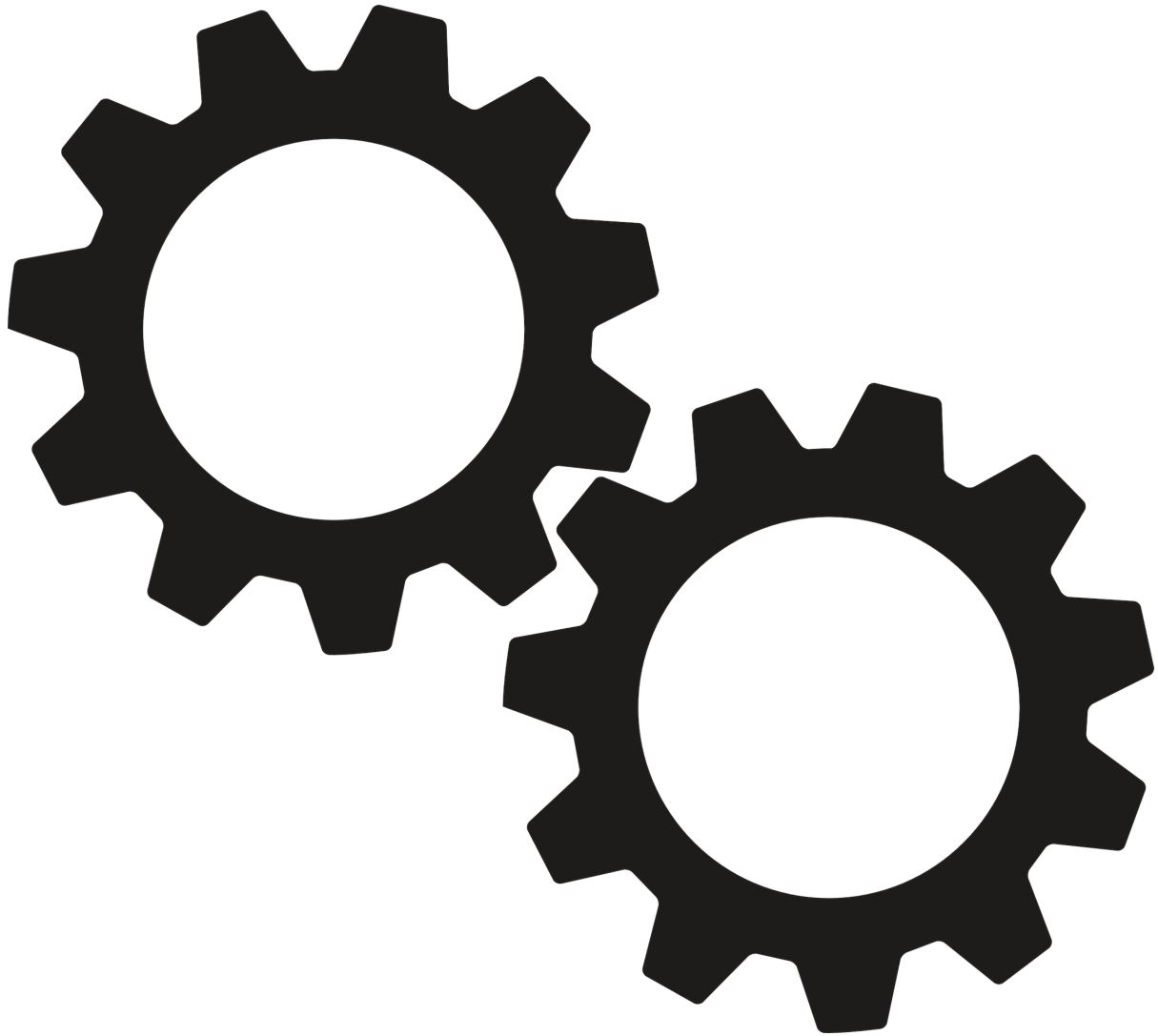
$k = 9$



Example



Solution: $(3, 6)$. Number of olives: $\sum_{i=3}^{6-1} \eta_i = 8$.



A Naive Solution

- For $i = 1, \ldots, n-1$:
  - For $j = i+1, \ldots, n$:
    - olives $\leftarrow 0$
    - For $k = i, \ldots, j-1$:
      - olives $\leftarrow$ olives $+ \eta_k$

. . .

A Naive Solution

- For $i = 1, \ldots, n-1$: $O(n)$
  - For $j = i+1, \ldots, n$: $O(n)$
    - olives $\leftarrow 0$
    - For $k = i, \ldots, j-1$: $O(n)$
      - olives $\leftarrow$ olives $+ \eta_k$

. . .

Total time: $O(n^3)$

A Naive Solution

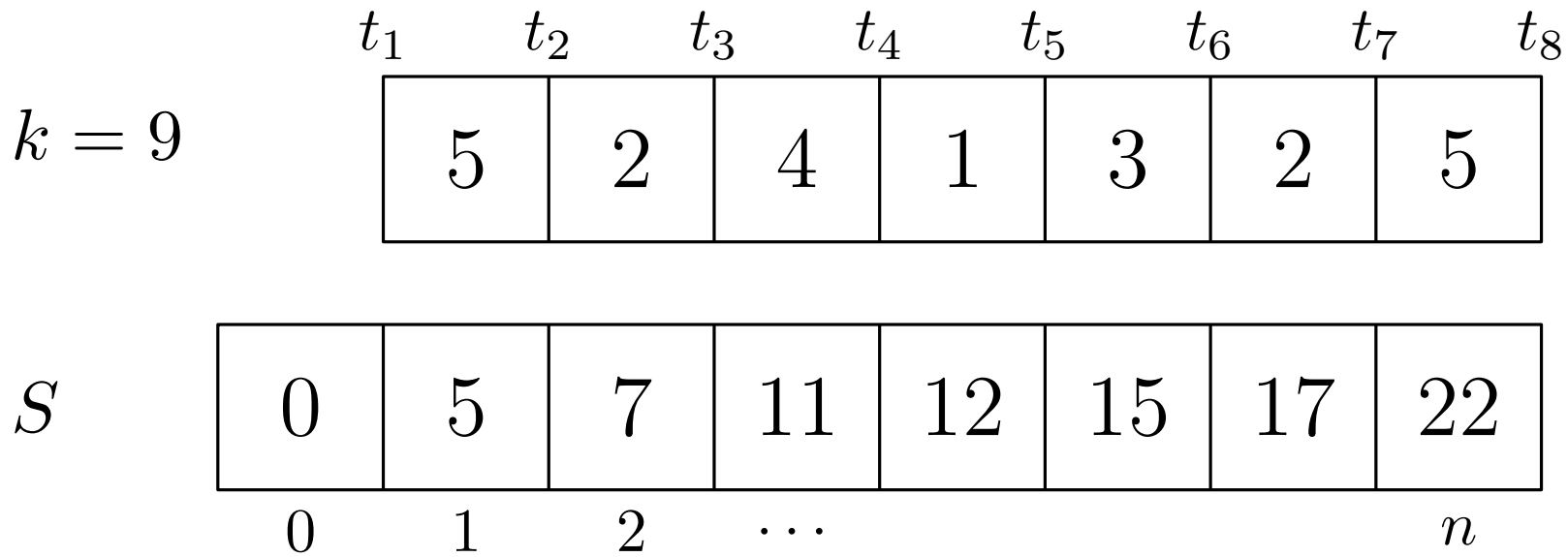
- For $i = 1, \ldots, n-1$: $O(n)$
  - For $j = i+1, \ldots, n$: $O(n)$
    - olives $\leftarrow 0$
    - For $k = i, \ldots, j-1$: $O(n)$
      - olives $\leftarrow$ olives $+ \eta_k$

. . .

Total time: $O(n^3)$

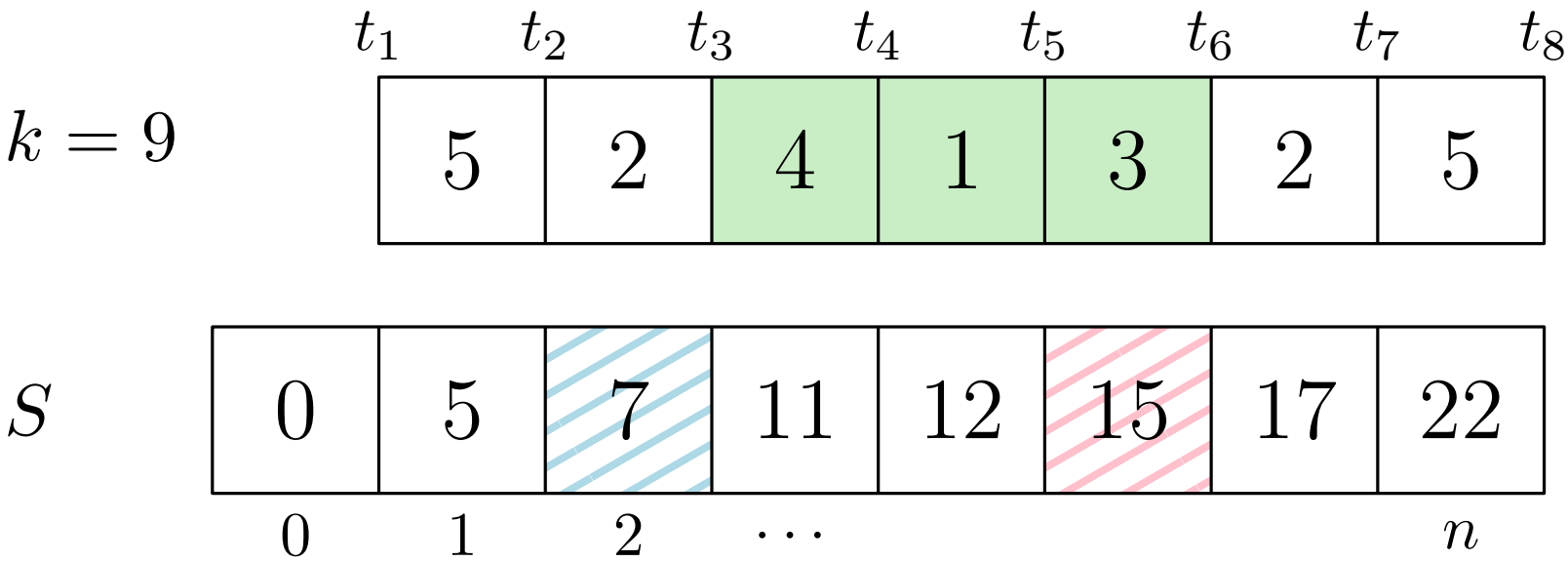
Can we do better?

Partial Sums



- Compute partial sums vector S $O(n)$

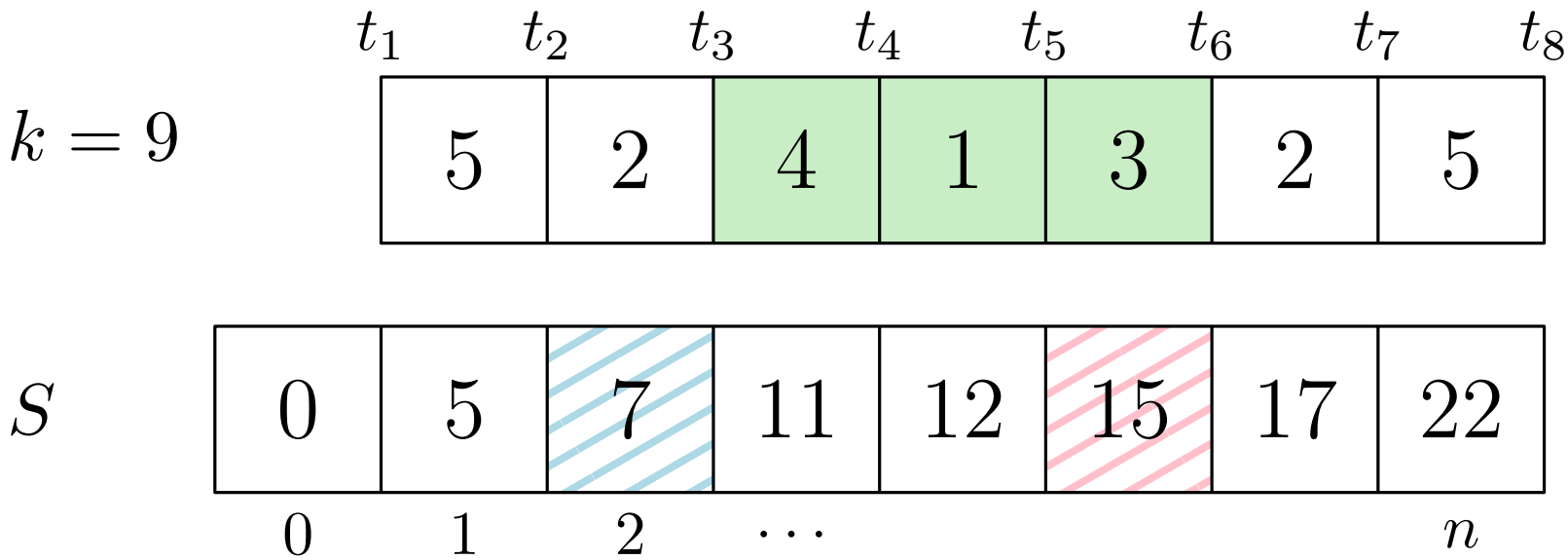
Partial Sums



- Compute partial sums vector S $O(n)$
- For $i = 1, \ldots, n-1$: $O(n)$
  - For $j = i+1, \ldots, n$: $O(n)$
    - olives $\leftarrow S[j-1] - S[i-1]$ $O(1)$

. . .

Partial Sums

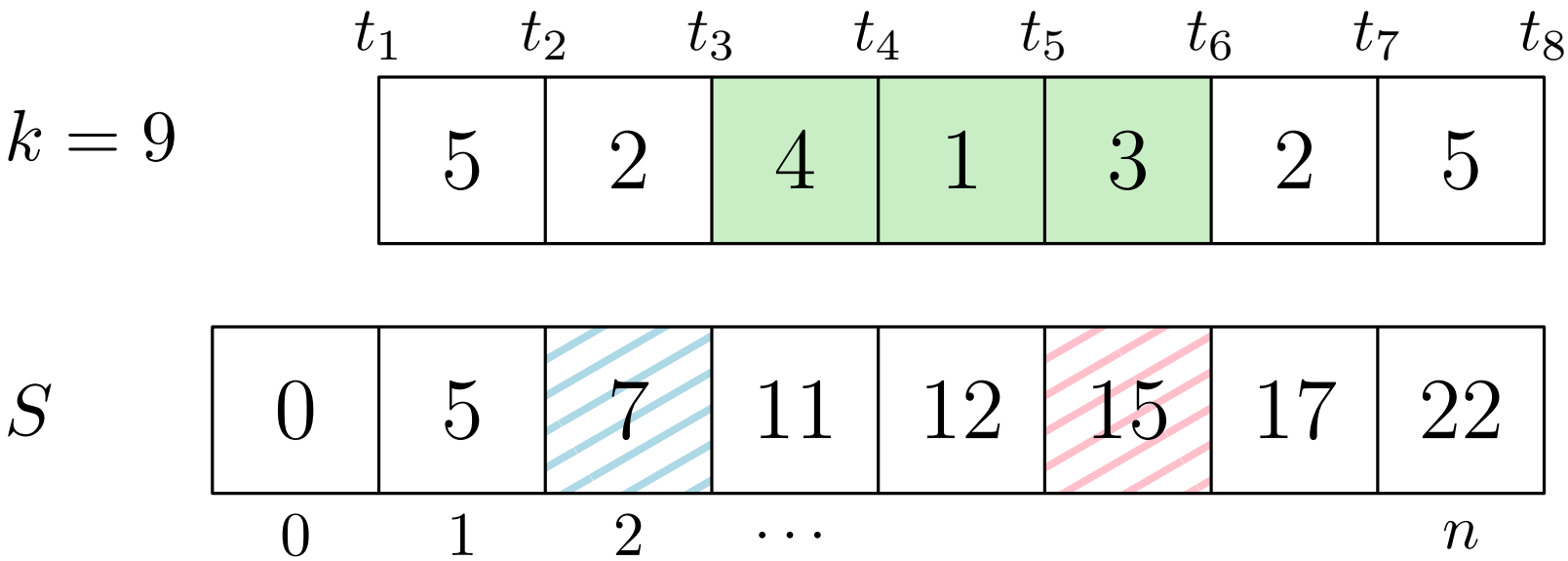


- Compute partial sums vector S $O(n)$
- For $i = 1, \ldots, n-1$: $O(n)$
  - For $j = i+1, \ldots, n$: $O(n)$
    - olives $\leftarrow S[j-1] - S[i-1]$ $O(1)$

. . .

Total time: $O(n^2)$

Partial Sums



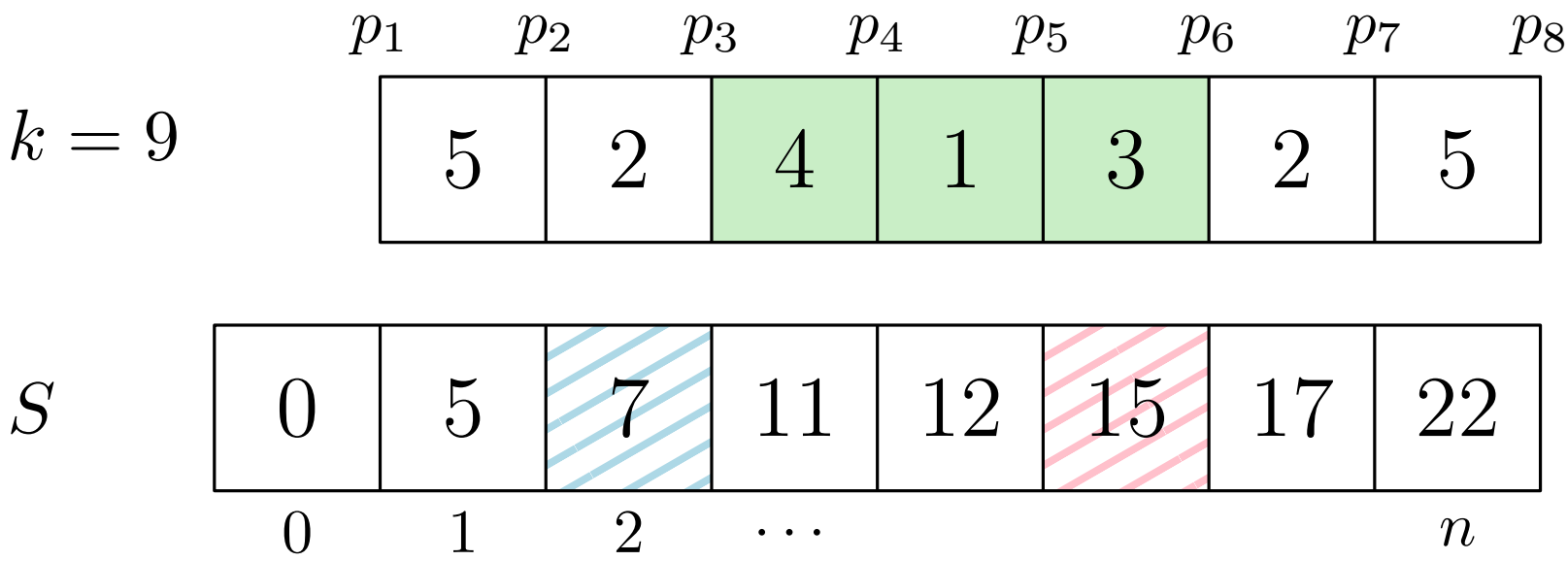
- Compute partial sums vector S $O(n)$
- For $i = 1, \ldots, n-1$: $O(n)$
  - For $j = i+1, \ldots, n$: $O(n)$
    - olives $\leftarrow S[j-1] - S[i-1]$ $O(1)$

. . .

Total time: $O(n^2)$

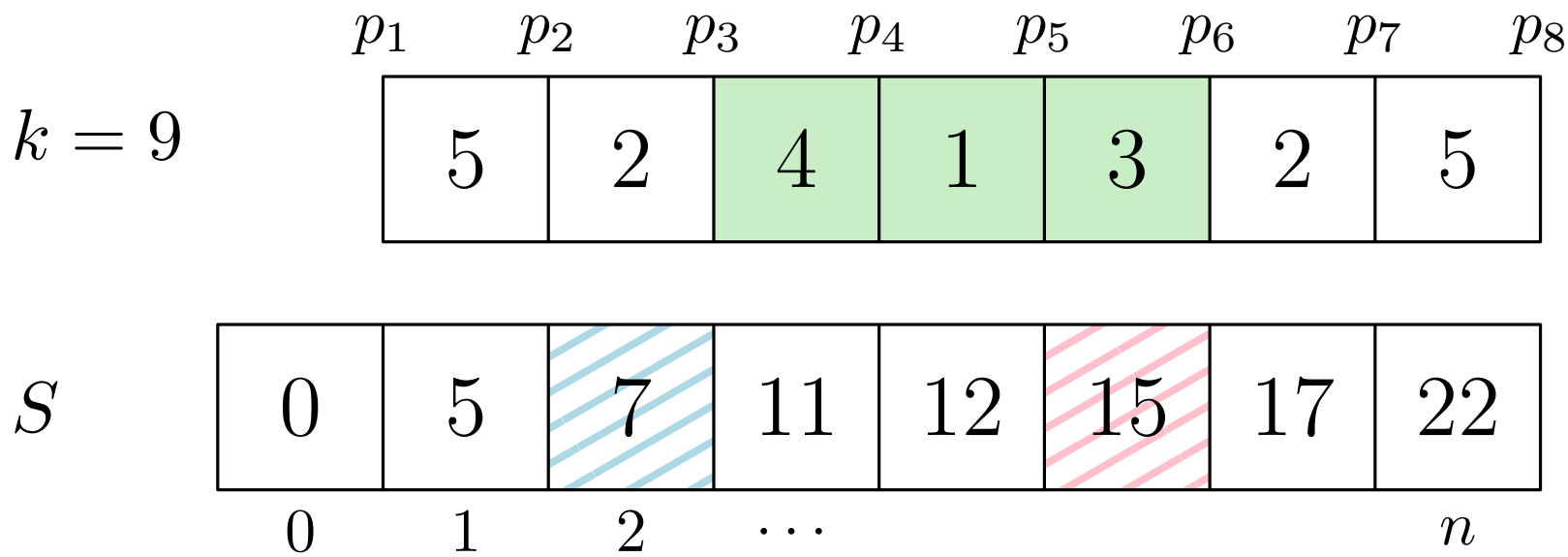
Can we do better?

Partial Sums + Binary Search



- Compute partial sums vector S — $O(n)$
- For $i = 1, \ldots, n-1$: — $O(n)$
  - Binary search S for the largest index $j \geq i$ such that $S[j] \leq S[i-1] + k$. — $O(\log n)$
  - olives $\leftarrow S[j] - S[i-1]$ — $O(1)$

Partial Sums + Binary Search



- Compute partial sums vector S — $O(n)$
- For $i = 1, \ldots, n-1$: — $O(n)$
  - Binary search S for the largest index $j \geq i$ such that $S[j] \leq S[i-1] + k$. — $O(\log n)$
  - olives $\leftarrow S[j] - S[i-1]$ — $O(1)$

Total time: $O(n \log n)$

Recap

- Naive

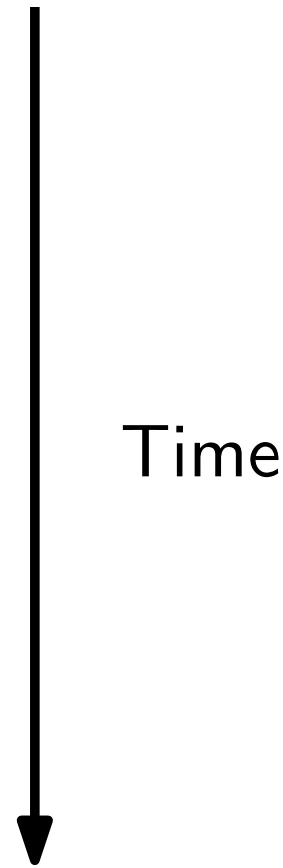
  $O(n^3)$

- Partial Sums

  $O(n^2)$

- Partial Sums + Binary Search

  $O(n \log n)$



Recap

- Naive

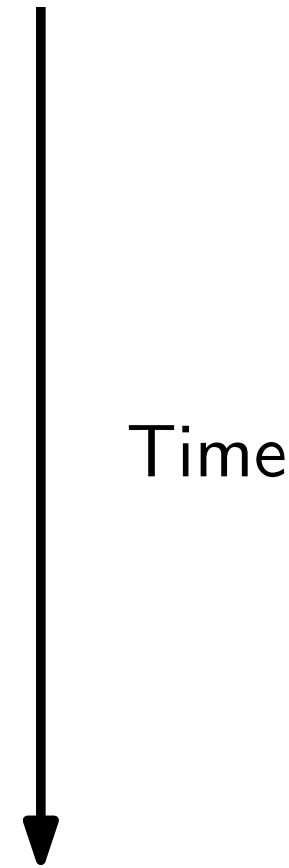
  $O(n^3)$

- Partial Sums

  $O(n^2)$

- Partial Sums + Binary Search

  $O(n \log n)$



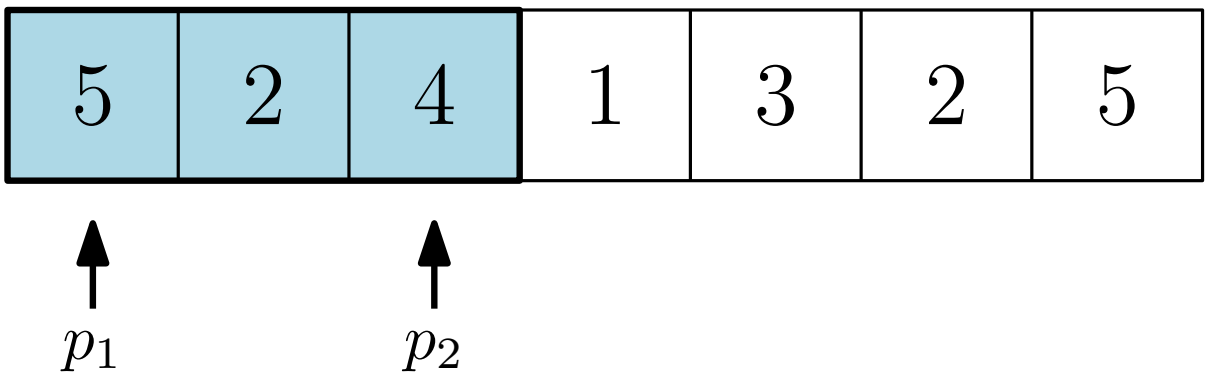
Can we do better?

Sliding Window

Sliding Window: Idea

- Keep two pointers p_1, p_2 to keep track of the current *window* W, i.e., the subsequence between p_1 and p_2.

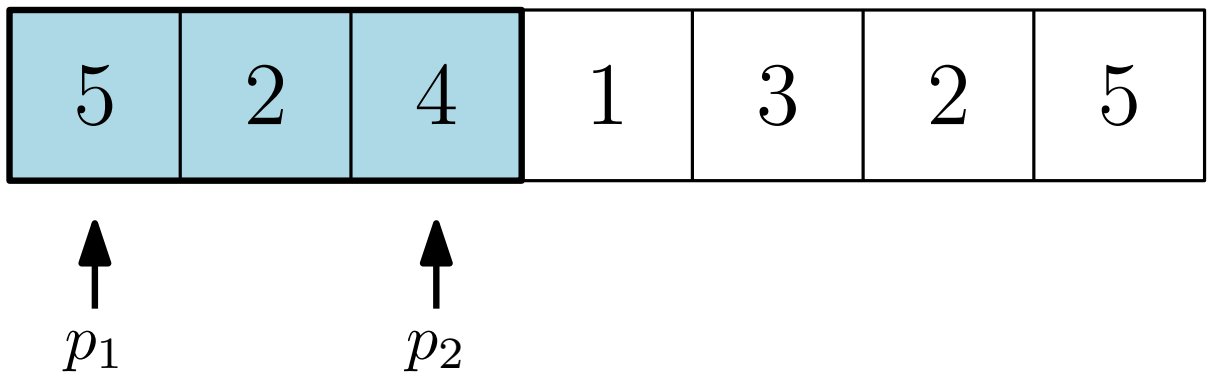
$k = 9$



Sliding Window: Idea

- Keep two pointers p_1, p_2 to keep track of the current *window* W, i.e., the subsequence between p_1 and p_2.

$k = 9$

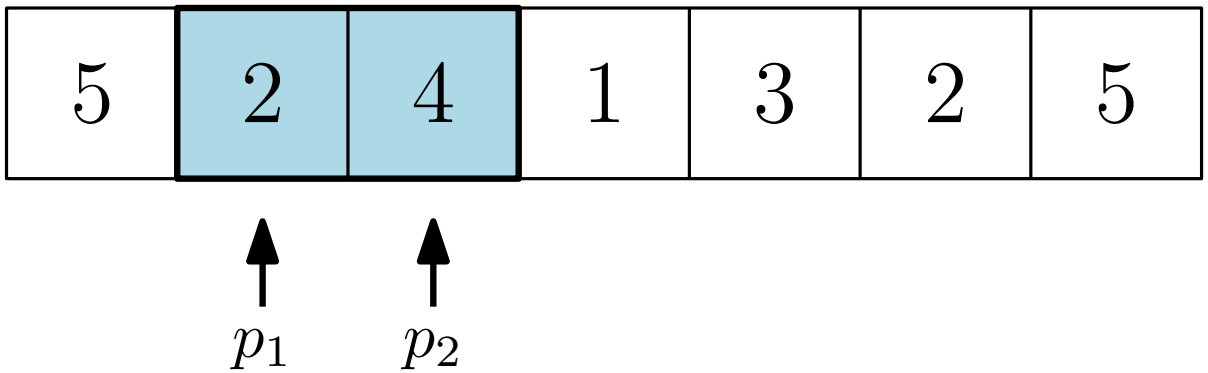


- Let $\sigma(p_1, p_2)$ be the sum of the elements in W.

Sliding Window: Idea

- Keep two pointers p_1, p_2 to keep track of the current *window* W, i.e., the subsequence between p_1 and p_2.

$k = 9$

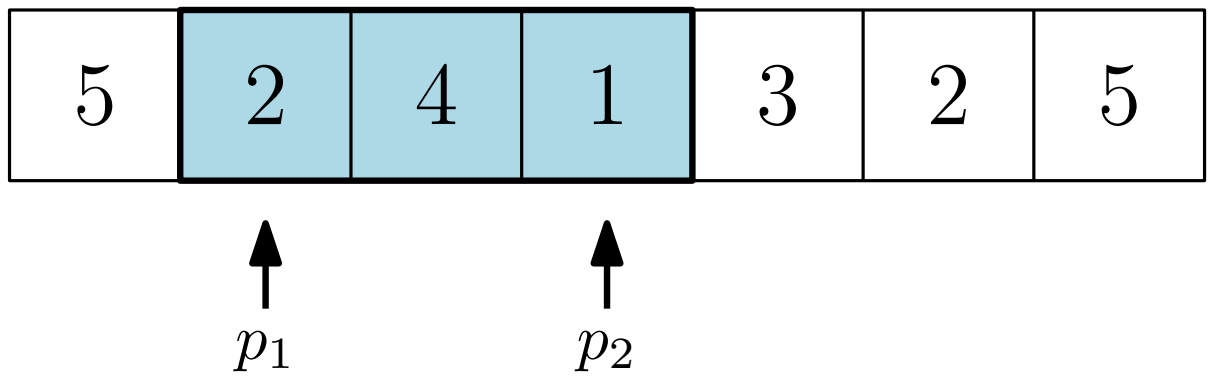


- Let $\sigma(p_1, p_2)$ be the sum of the elements in W.
- If $\sigma(p_1, p_2)$ is too large: increase p_1.

Sliding Window: Idea

- Keep two pointers p_1, p_2 to keep track of the current *window* W, i.e., the subsequence between p_1 and p_2.

$k = 9$

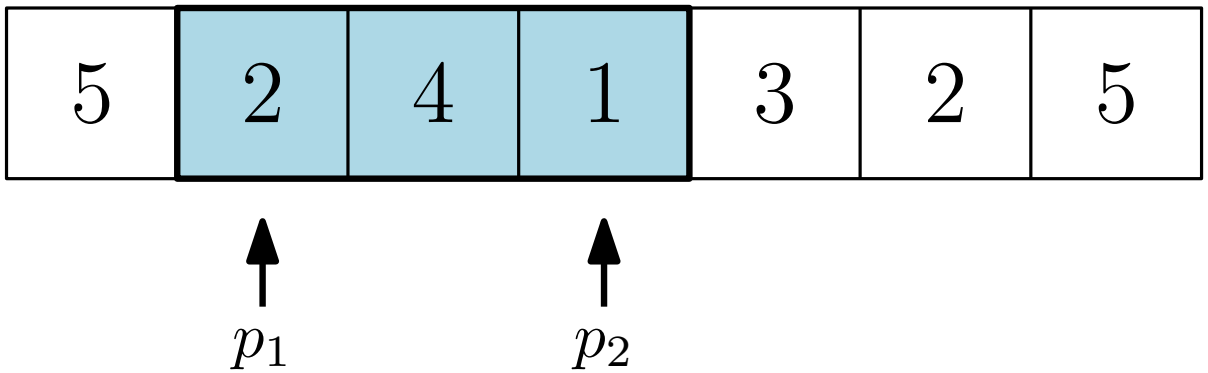


- Let $\sigma(p_1, p_2)$ be the sum of the elements in W.
- If $\sigma(p_1, p_2)$ is too large: increase p_1.
- If $\sigma(p_1, p_2)$ is too small: increase p_2.

Sliding Window: Idea

- Keep two pointers p_1, p_2 to keep track of the current *window* W, i.e., the subsequence between p_1 and p_2.

$k = 9$



- Let $\sigma(p_1, p_2)$ be the sum of the elements in W.
- If $\sigma(p_1, p_2)$ is too large: increase p_1.
- If $\sigma(p_1, p_2)$ is too small: increase p_2.
- Return "best" feasible window among those considered.

(plus suitable handling of edge cases)

A possible implementation

```
int left=0, right=-1, sum=0;
int best_left=-1, best_right=-1, best_sum=-1;

do
{
    if(sum<=k && right<n-1)
        sum += A[++right];
    else
        sum -= A[left++];

    if(sum<=k && sum>best_sum)
    {
        best_sum = sum; best_left = left; best_right = right;
    }
} while(left<n-1 || right<n-1);

std::cout << "Cut from position " << best_left+1
          << " to position " << best_right+2 << "\n";
```

A possible implementation

```
int left=0, right=-1, sum=0;
int best_left=-1, best_right=-1, best_sum=-1;

do
{
    if(sum<=k && right<n-1)
        sum += A[++right];
    else
        sum -= A[left++];

    if(sum<=k && sum>best_sum)
    {
        best_sum = sum; best_left = left; best_right = right;
    }
} while(left<n-1 || right<n-1);

std::cout << "Cut from position " << best_left+1
          << " to position " << best_right+2 << "\n";
```

Running time?

A possible implementation

```
int left=0, right=-1, sum=0;
int best_left=-1, best_right=-1, best_sum=-1;

do
{
    if(sum<=k && right<n-1)
        sum += A[++right];
    else
        sum -= A[left++];

    if(sum<=k && sum>best_sum)
    {
        best_sum = sum; best_left = left; best_right = right;
    }
} while(left<n-1 || right<n-1);

std::cout << "Cut from position " << best_left+1
          << " to position " << best_right+2 << "\n";
```

Running time?

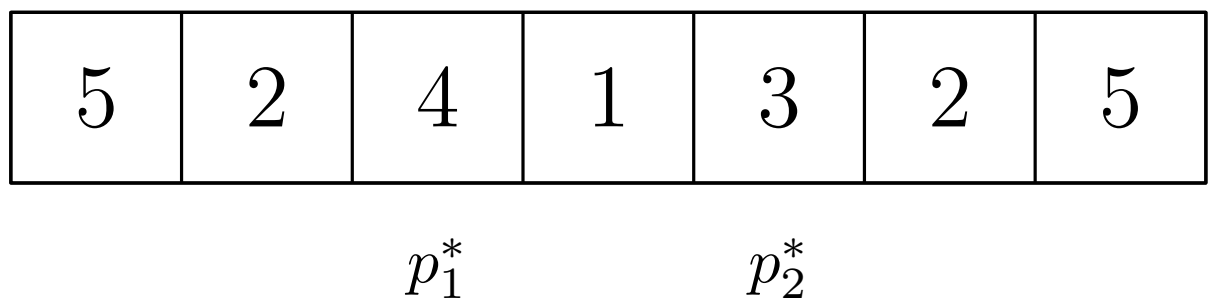
$O(n)$

Why does it work?

Observation: p_1 (and p_2) will get all values from 1 to n.

- Let $W^* = [p_1^*, p_2^*]$ be an optimal interval minimizing p_1^*.
- Consider the first instant where $p_1 = p_1^*$ or $p_2 = p_2^*$.

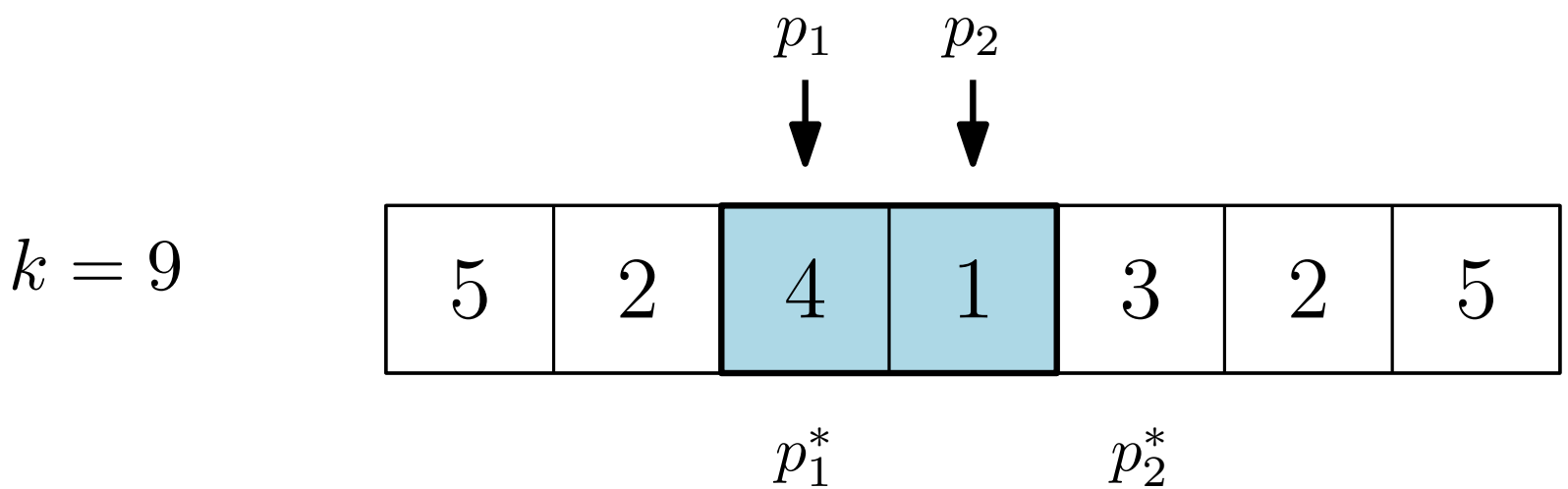
$k = 9$



Why does it work?

Observation: p_1 (and p_2) will get all values from 1 to n.

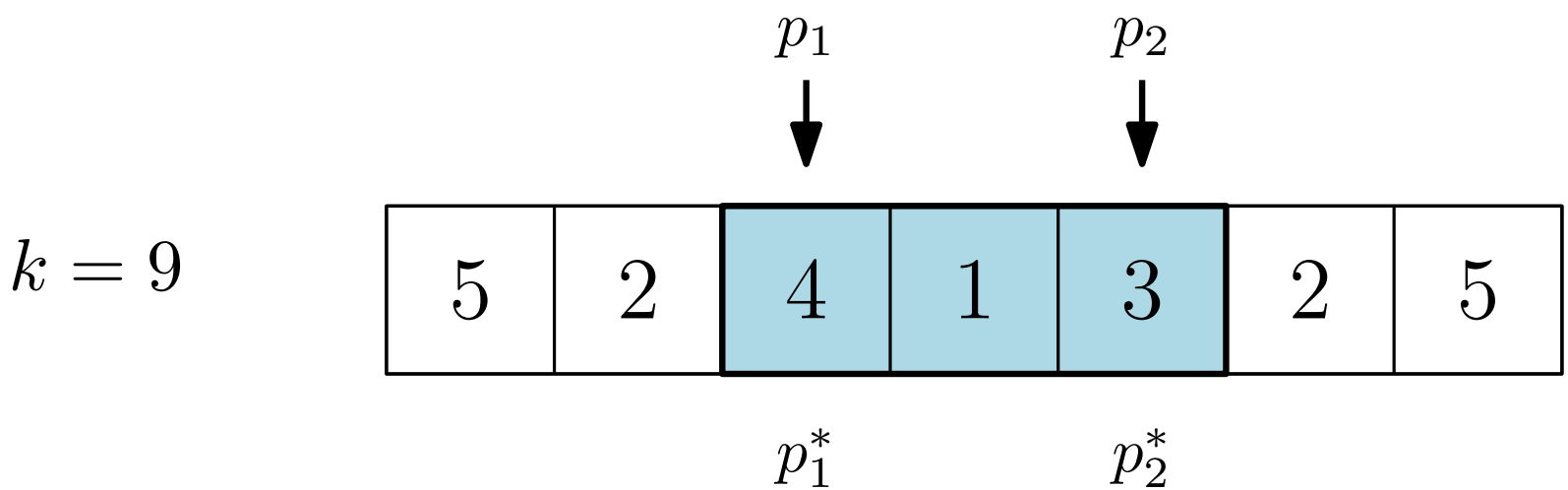
- Let $W^* = [p_1^*, p_2^*]$ be an optimal interval minimizing p_1^*.
- Consider the first instant where $p_1 = p_1^*$ or $p_2 = p_2^*$.
- If $p_1 = p_1^*$, then $p_2 < p_2^*$ and $\sigma(p_1, p) \leq \sigma(p_1^*, p_2^*) \leq k$, for every $p \in [p_2, p_2^* - 1]$.



Why does it work?

Observation: p_1 (and p_2) will get all values from 1 to n.

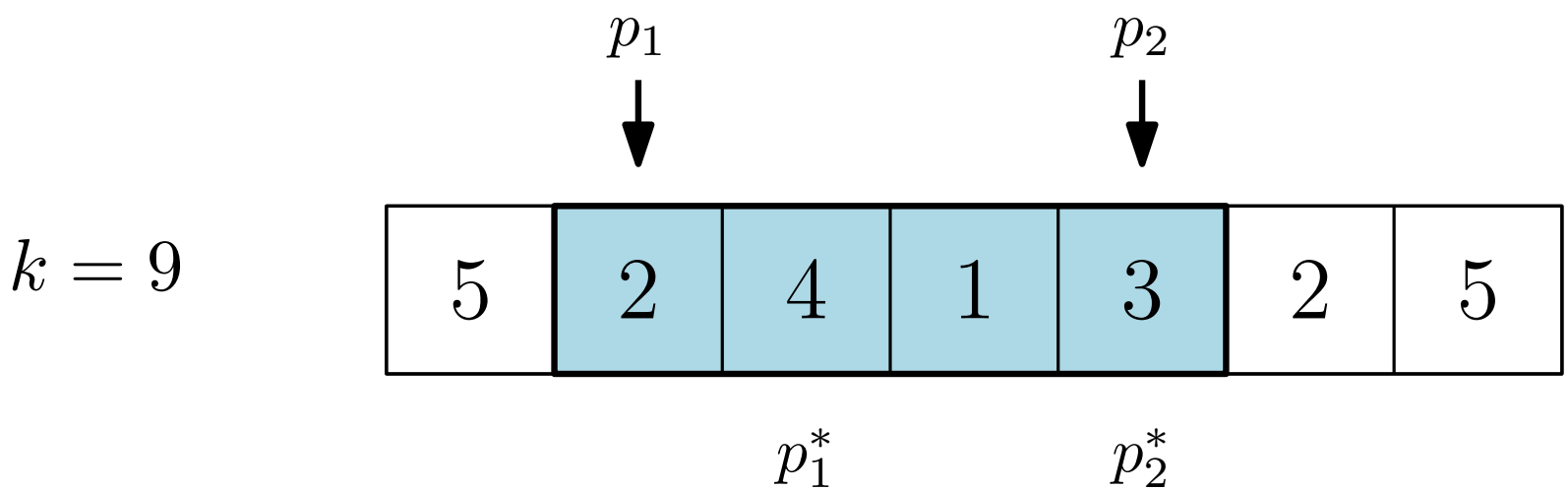
- Let $W^* = [p_1^*, p_2^*]$ be an optimal interval minimizing p_1^*.
- Consider the first instant where $p_1 = p_1^*$ or $p_2 = p_2^*$.
- If $p_1 = p_1^*$, then $p_2 < p_2^*$ and $\sigma(p_1, p) \leq \sigma(p_1^*, p_2^*) \leq k$, for every $p \in [p_2, p_2^* - 1]$.
- Therefore, p_2 will be incremented until it reaches p_2^* while $p_1 = p_1^*$ remains constant $\Longrightarrow$ the algorithm considers W^*.



Why does it work?

Observation: p_1 (and p_2) will get all values from 1 to n.

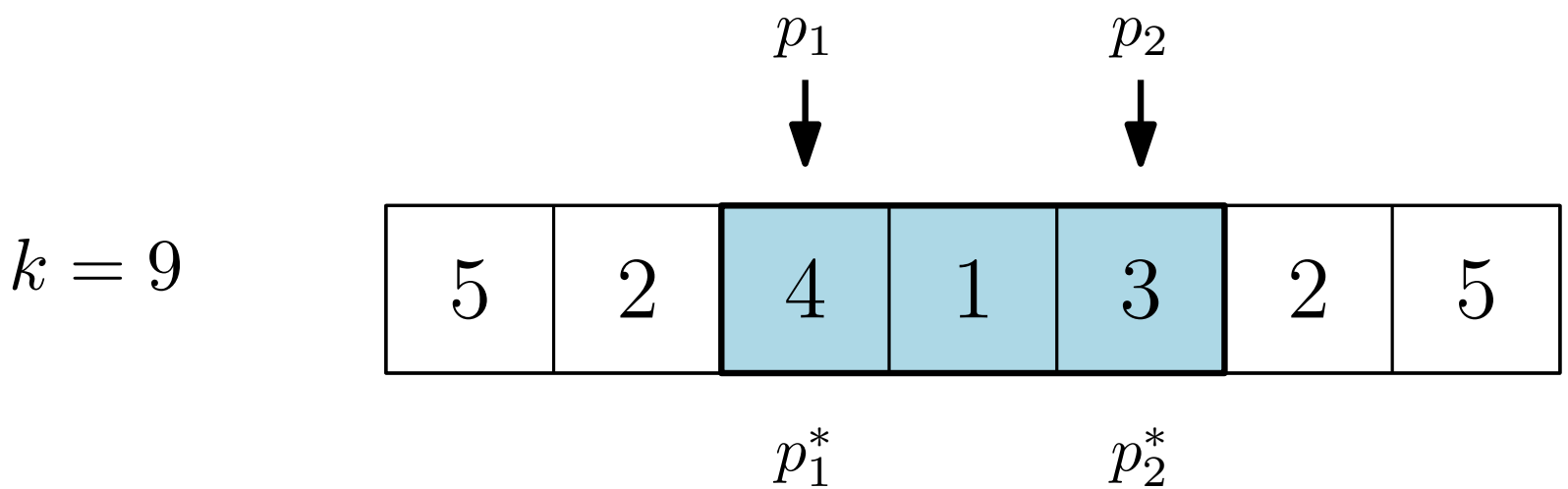
- Let $W^* = [p_1^*, p_2^*]$ be an optimal interval minimizing p_1^*.
- Consider the first instant where $p_1 = p_1^*$ or $p_2 = p_2^*$.
- If $p_2 = p_2^*$, then $p_1 < p_1^*$ and, for every $p \in [p_1, p_1^* - 1]$, $\sigma(p, p_2) > \sigma(p_1^*, p_2^*)$ and hence $\sigma(p, p_2) > k$.



Why does it work?

Observation: p_1 (and p_2) will get all values from 1 to n.

- Let $W^* = [p_1^*, p_2^*]$ be an optimal interval minimizing p_1^*.
- Consider the first instant where $p_1 = p_1^*$ or $p_2 = p_2^*$.
- If $p_2 = p_2^*$, then $p_1 < p_1^*$ and, for every $p \in [p_1, p_1^* - 1]$, $\sigma(p, p_2) > \sigma(p_1^*, p_2^*)$ and hence $\sigma(p, p_2) > k$.
- Therefore, p_1 will be incremented until it reaches p_1^* while $p_2 = p_2^*$ remains constant $\Longrightarrow$ the algorithm considers W^*.



Sliding Window

- We have proven that the algorithm always considers an optimal window.

Sliding Window

- We have proven that the algorithm always considers an optimal window.

Trick/Technique: Sliding Window

Some problems in which you need to find an interval can be solved in linear time using a sliding window approach, if you can ensure that an optimal interval will be considered.

Sushi Belt

Sushi Belt

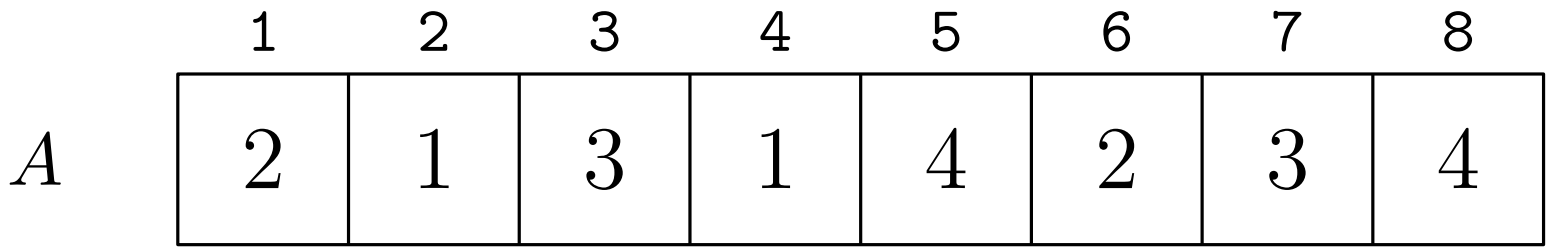
Gustavo is in a Sushi-belt restaurant: n small plates are lined up on a conveyor belt and will soon reach him

- Gustavo can eat an unlimited amount of food, as long as he never stops eating
- Gustavo does not want to eat any repeated dish
- Gustavo wants to eat as much as possible

What is the maximum number of dishes that Gustavo can eat?

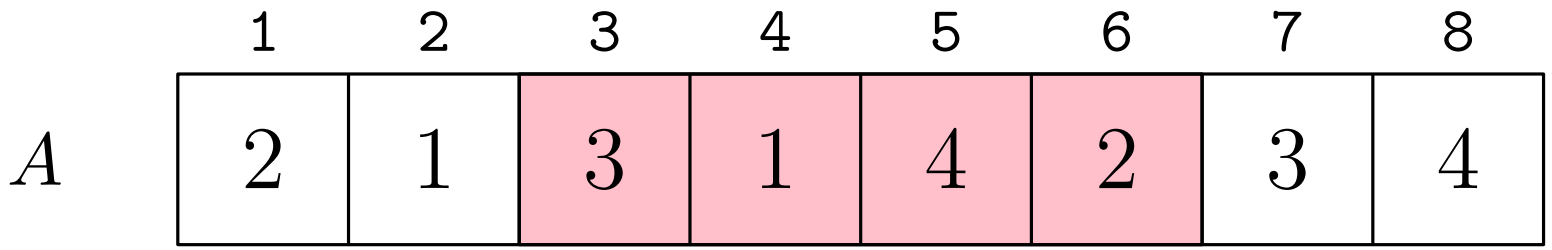
Sushi Belt

Given an array A of n integers in $\{1, \ldots, n\}$, find the longest contiguous subarray of A that contains only distinct elements.



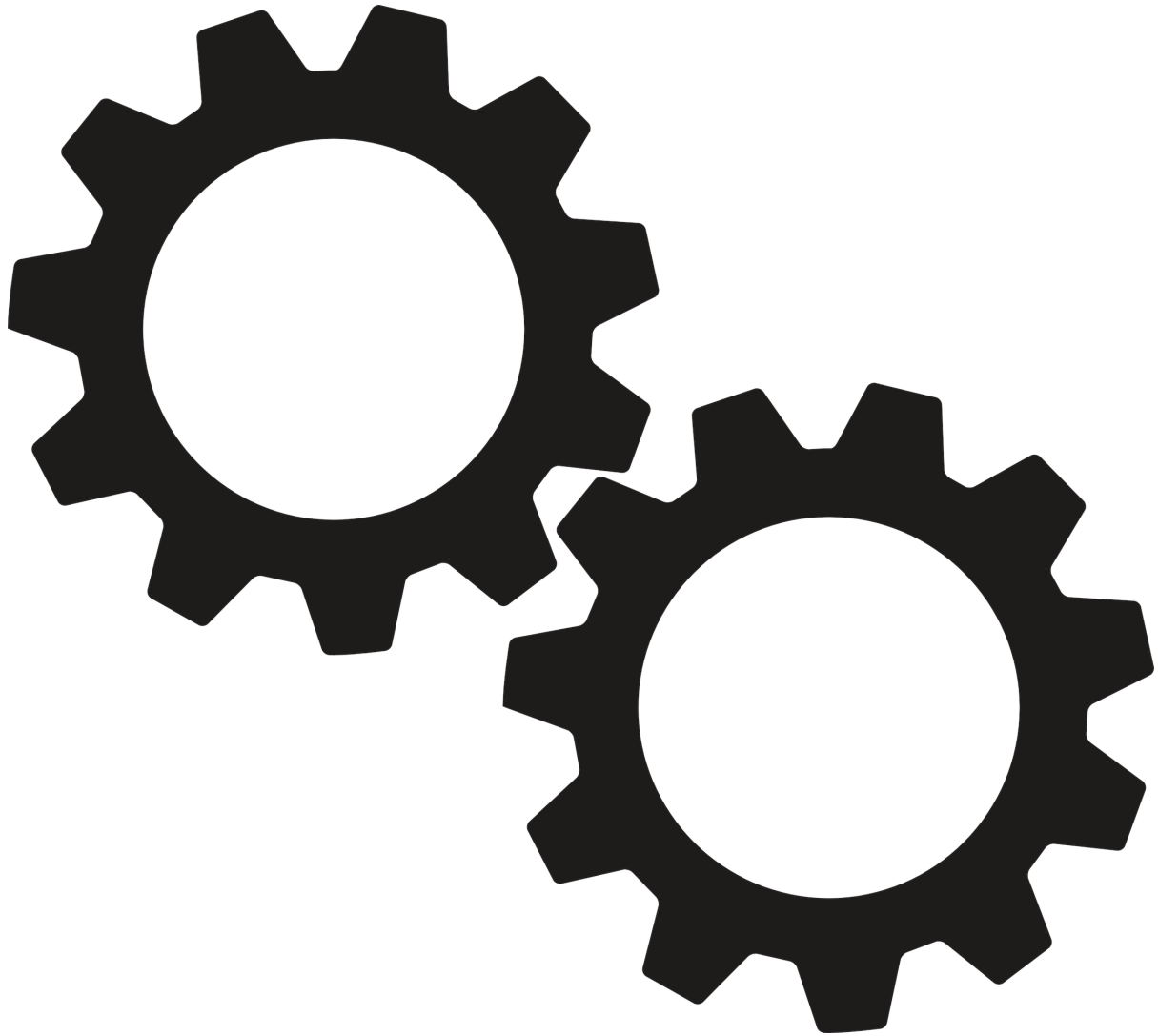
Sushi Belt

Given an array A of n integers in $\{1, \ldots, n\}$, find the longest contiguous subarray of A that contains only distinct elements.



Solution: $A[3 \ldots, 6]$, length: 4

Start eating from the 3rd plate, eat up to (and including) the 6-th plate



Sushi Belt: Naive Solution

- For $i = 1, \ldots, n$:
  - For $j = i, \ldots, n$:
    - If $A[i \ldots j]$ contains no duplicates:
      - $A[i \ldots j]$ is a candidate solution

    . . .
- Return longest candidate solution found

Sushi Belt: Naive Solution

- For $i = 1, \ldots, n$: $O(n)$
  - For $j = i, \ldots, n$: $O(n)$
    - If $A[i \ldots j]$ contains no duplicates: $O(n^2)$
      - $A[i \ldots j]$ is a candidate solution

    . . .
- Return longest candidate solution found

Total time: $O(n^4)$

Sushi Belt: Naive Solution

- For $i = 1, \ldots, n$: $O(n)$
  - For $j = i, \ldots, n$: $O(n)$
    - If $A[i \ldots j]$ contains no duplicates: $O(n^2)$
      - $A[i \ldots j]$ is a candidate solution

    . . .
- Return longest candidate solution found

Total time: $O(n^4)$

Sushi Belt: Naive Solution

- For $i = 1, \ldots, n$: $O(n)$
  - For $j = i, \ldots, n$: $O(n)$
    - If $A[i \ldots j]$ contains no duplicates: ~~$O(n^2)$~~ $O(n)$
      - $A[i \ldots j]$ is a candidate solution

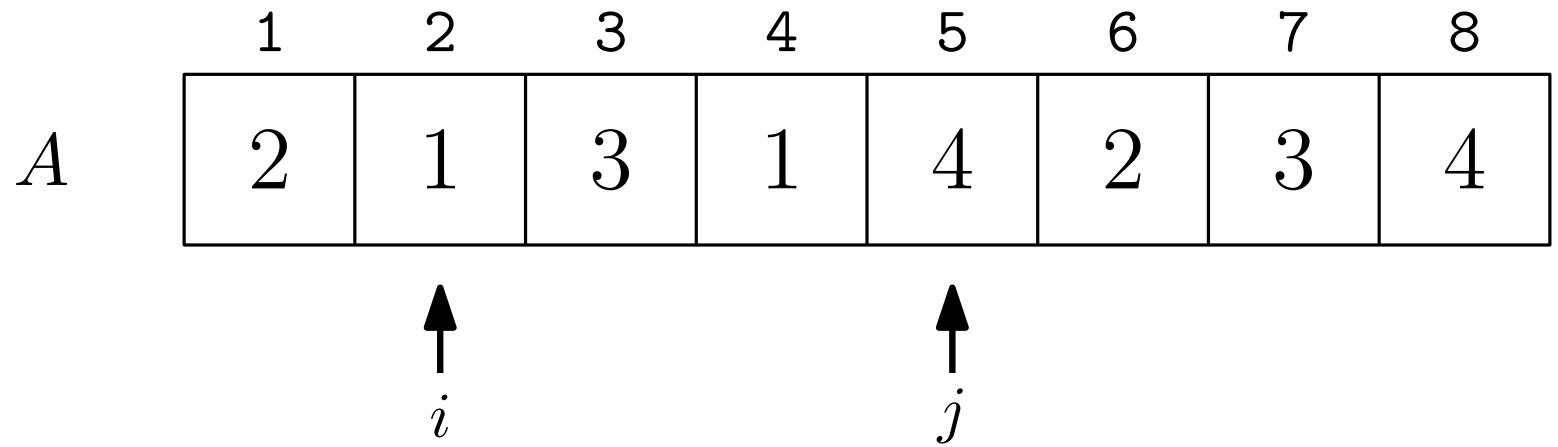
    . . .
- Return longest candidate solution found

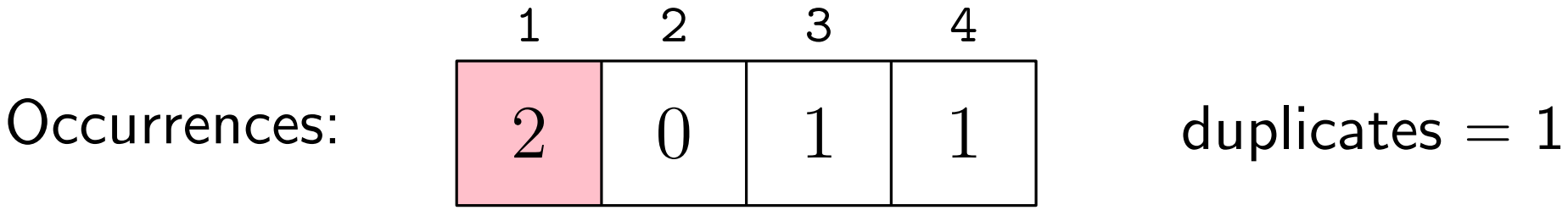
~~Total time: $O(n^4)$~~

Total time: $O(n^3)$ via counting sort

Sushi Belt: Checking for Duplicates

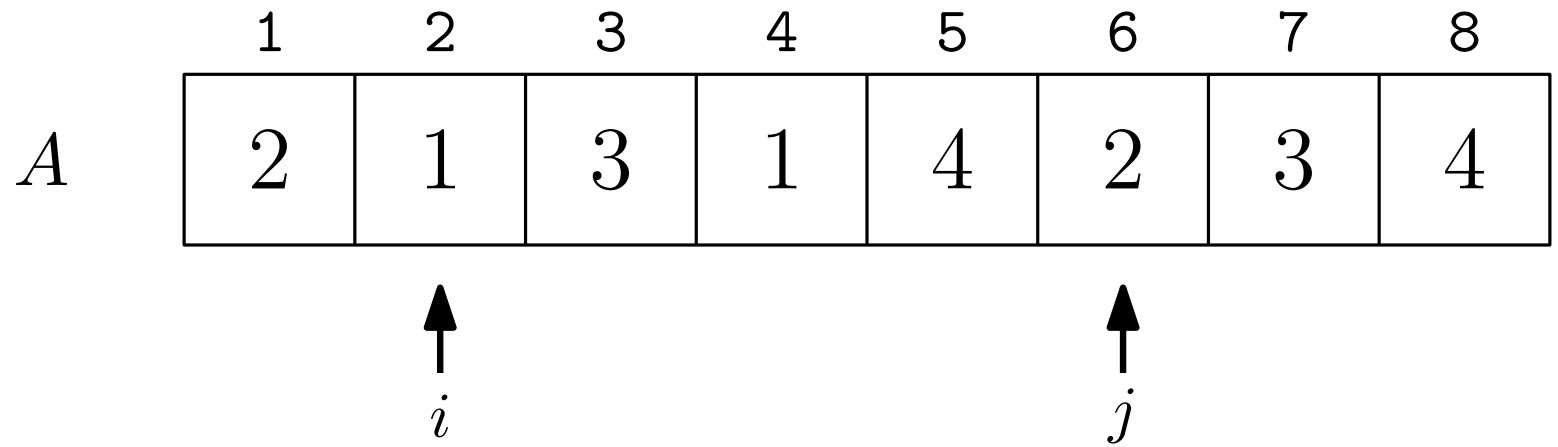
- Do not run counting sort each time we need to check for duplicates
- Keep the number of occurrences of each type updated
- Keep track of the number of duplicates, i.e., counts ≥ 2

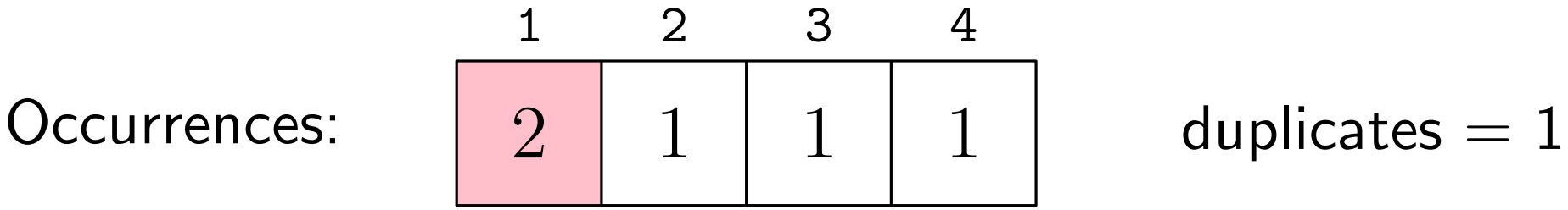




Sushi Belt: Checking for Duplicates

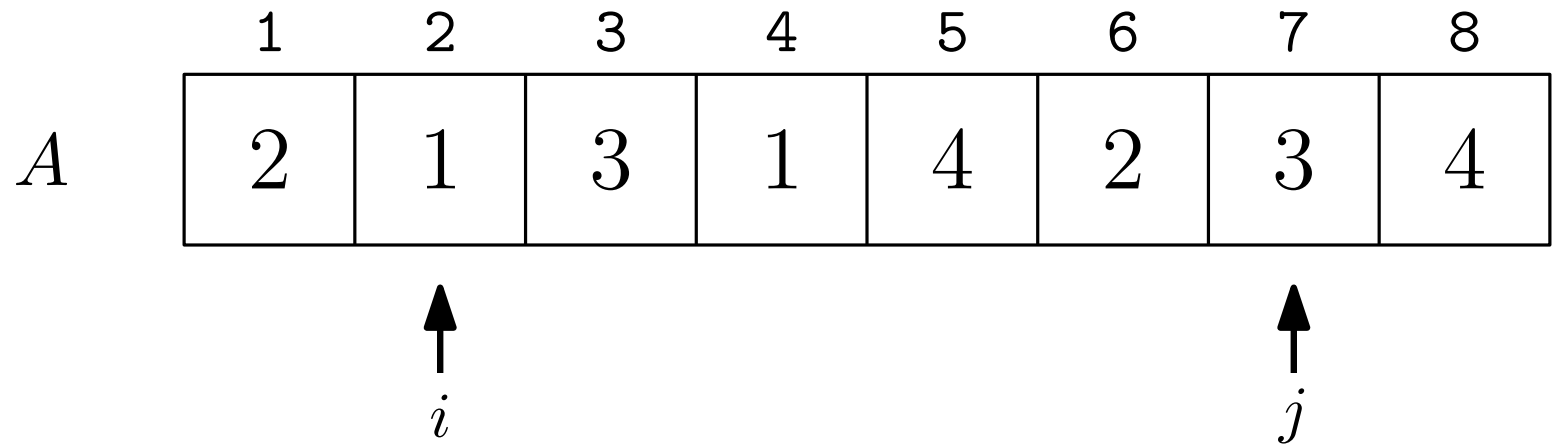
- Do not run counting sort each time we need to check for duplicates
- Keep the number of occurrences of each type updated
- Keep track of the number of duplicates, i.e., counts ≥ 2

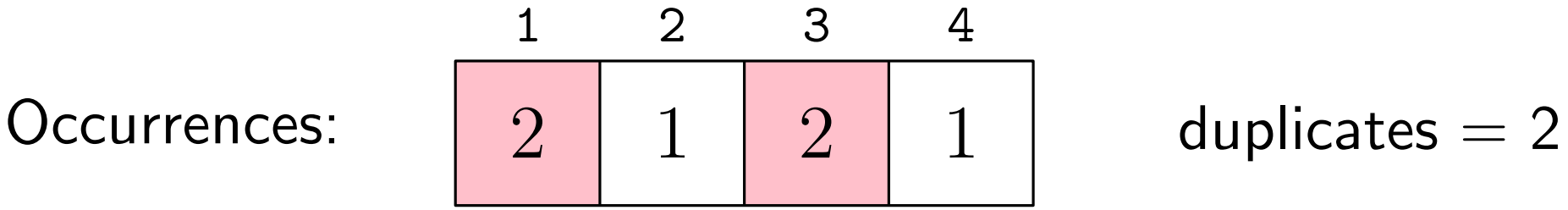




Sushi Belt: Checking for Duplicates

- Do not run counting sort each time we need to check for duplicates
- Keep the number of occurrences of each type updated
- Keep track of the number of duplicates, i.e., counts ≥ 2





Sushi Belt: (A Less) Naive Solution

- For $i = 1, \ldots, n$: $O(n)$
  - For $j = i, \ldots, n$: $O(n)$
    - If $A[i \ldots j]$ contains no duplicates: ~~$O(n^2)$~~ $O(n)$
      - $A[i \ldots j]$ is a candidate solution

    . . .
- Return longest candidate solution found

~~Total time: $O(n^4)$~~

Total time: $O(n^3)$ via counting sort

Sushi Belt: (A Less) Naive Solution

- For $i = 1, \ldots, n$: $O(n)$
  - For $j = i, \ldots, n$: $O(n)$
    - If $A[i \ldots j]$ contains no duplicates: ~~$O(n^2)$~~ ~~$O(n)$~~ $O(1)$
      - $A[i \ldots j]$ is a candidate solution

    . . .
- Return longest candidate solution found

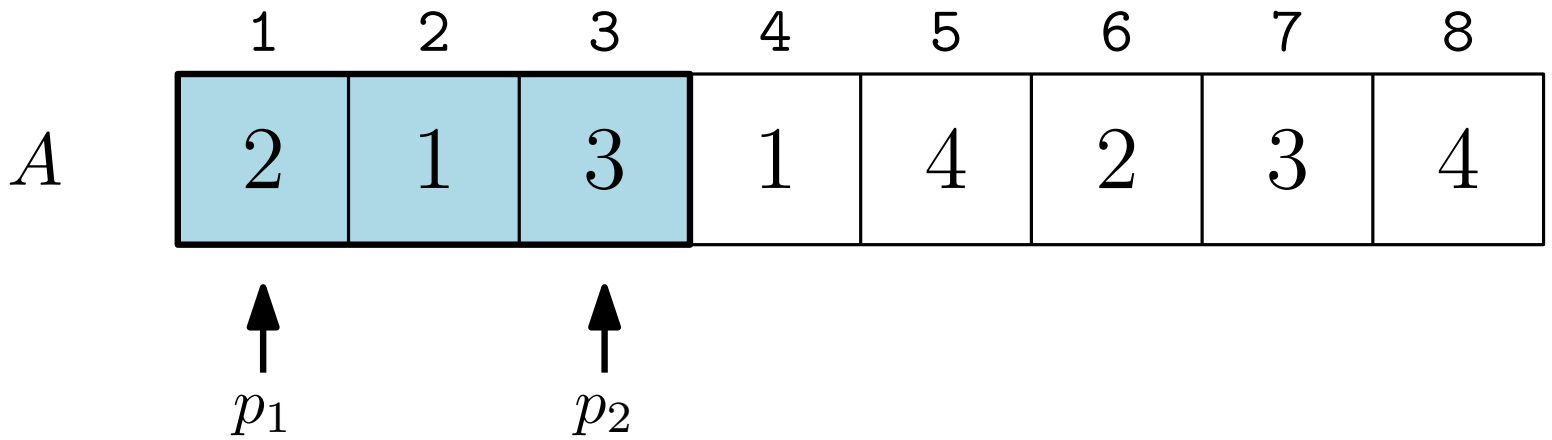
~~Total time: $O(n^4)$~~

~~Total time: $O(n^3)$ via counting sort~~

Total time: $O(n^2)$ by updating counts in $O(1)$ time

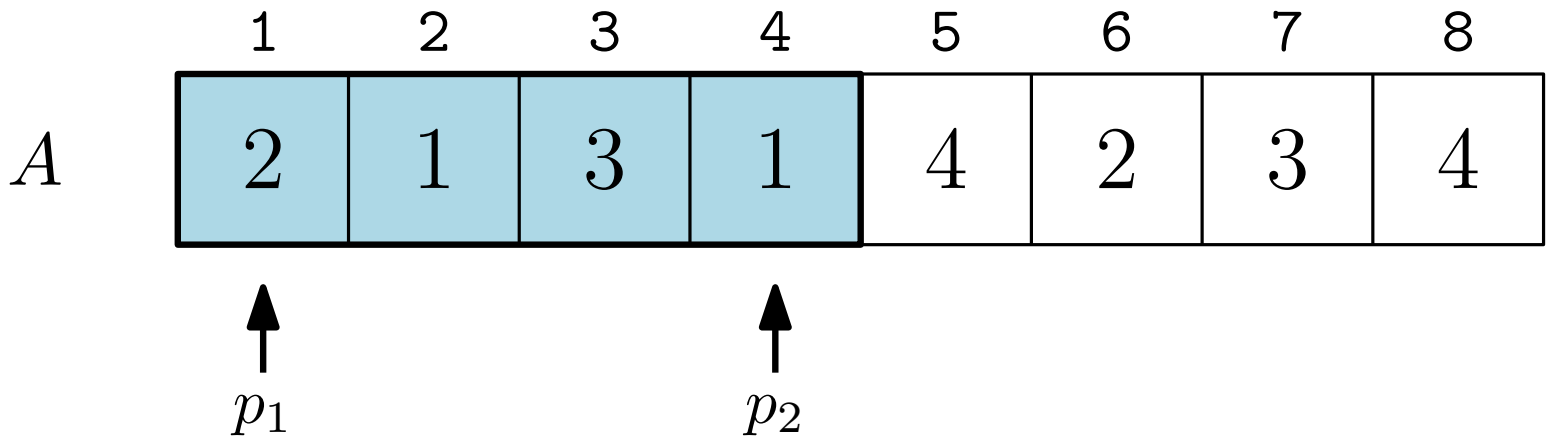
Sushi Belt: Sliding Window

- Keep two pointers p_1, p_2 to keep track of the current *window* $W = [p_1, p_2]$
- Initially $p_1 = 1$, $p_2 = 0$



Sushi Belt: Sliding Window

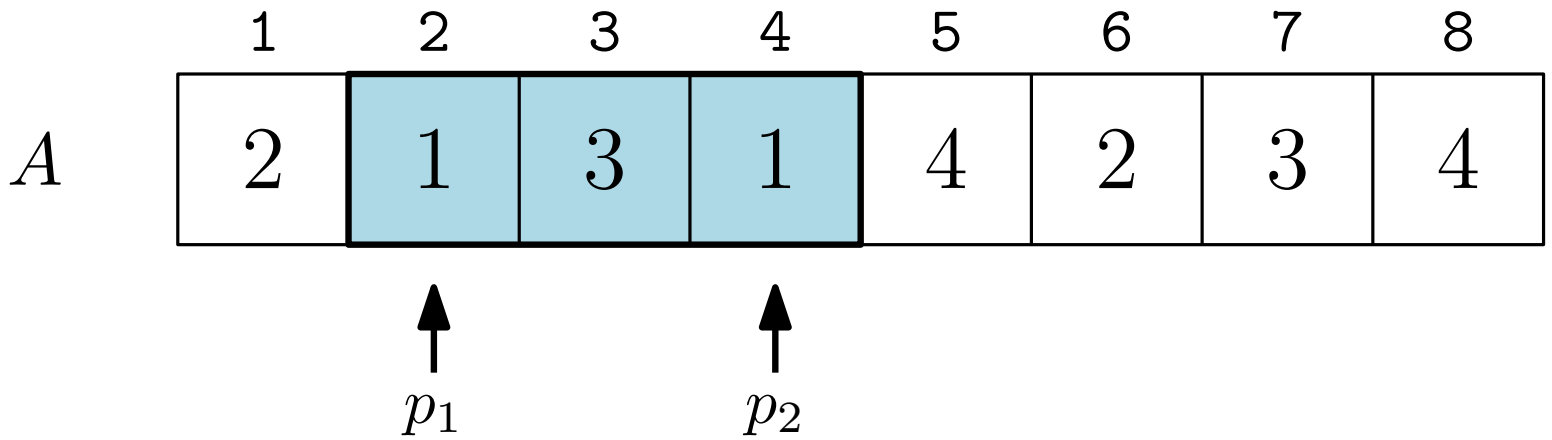
- Keep two pointers p_1, p_2 to keep track of the current *window* $W = [p_1, p_2]$
- Initially $p_1 = 1$, $p_2 = 0$



- If $A[p_1 \dots p_2]$ contains no duplicates and $p_2 < n$: increase p_2

Sushi Belt: Sliding Window

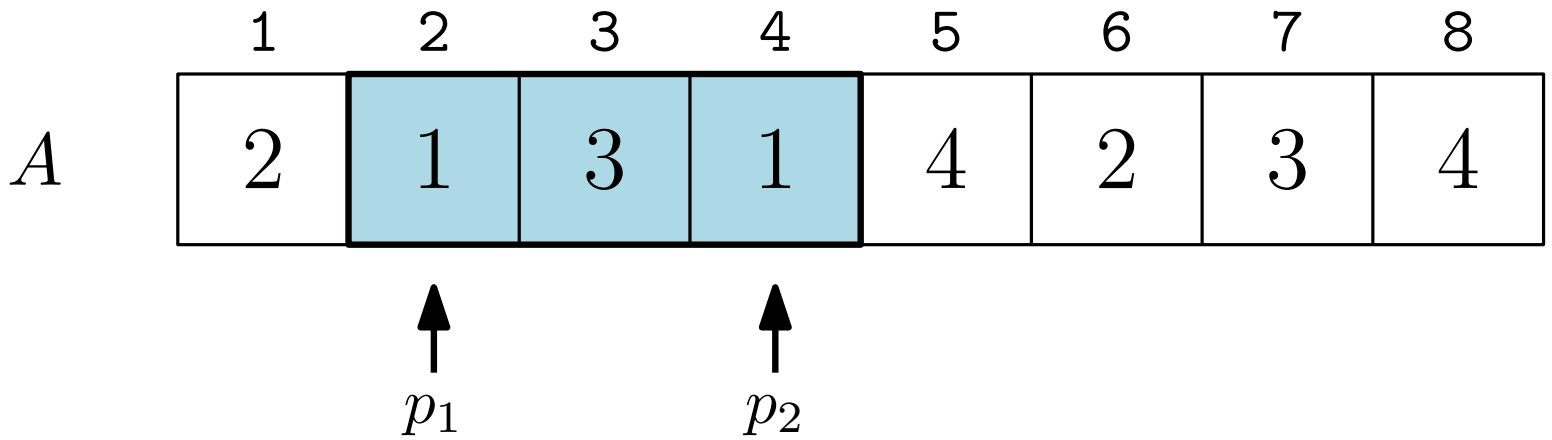
- Keep two pointers p_1, p_2 to keep track of the current *window* $W = [p_1, p_2]$
- Initially $p_1 = 1$, $p_2 = 0$



- If $A[p_1 \ldots p_2]$ contains no duplicates and $p_2 < n$: increase p_2
- If $A[p_1 \ldots p_2]$ contains duplicates or $p_2 = n$: increase p_1

Sushi Belt: Sliding Window

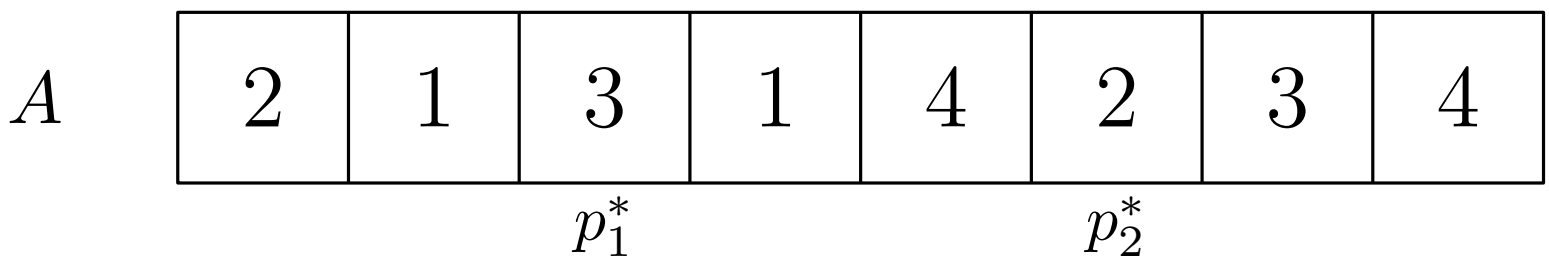
- Keep two pointers p_1, p_2 to keep track of the current *window* $W = [p_1, p_2]$
- Initially $p_1 = 1$, $p_2 = 0$



- If $A[p_1 \ldots p_2]$ contains no duplicates and $p_2 < n$: increase p_2
- If $A[p_1 \ldots p_2]$ contains duplicates or $p_2 = n$: increase p_1
- Return "best" feasible window among those considered.

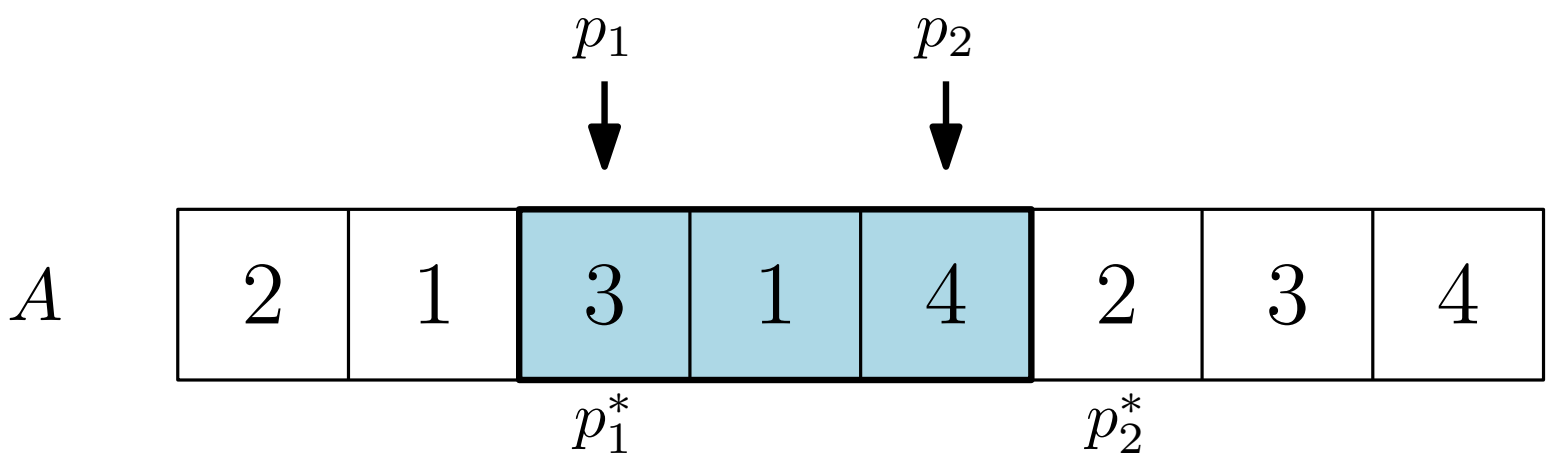
Sliding Window: Correctness

- Let $W^* = [p_1^*, p_2^*]$ be an optimal interval
- Consider the first instant where $p_1 = p_1^*$ or $p_2 = p_2^*$.



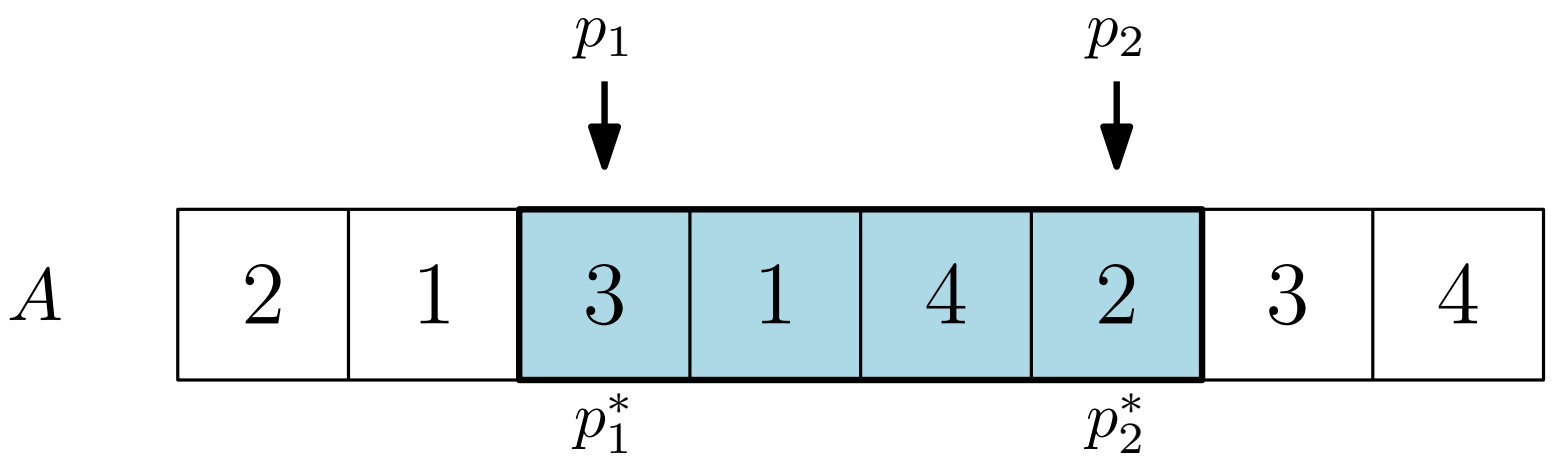
Sliding Window: Correctness

- Let $W^* = [p_1^*, p_2^*]$ be an optimal interval
- Consider the first instant where $p_1 = p_1^*$ or $p_2 = p_2^*$.
- If $p_1 = p_1^*$ then $p_2 < p_2^*$ and $A[p_1 \ldots p]$ contains no duplicates for all $p = p_2, \ldots, p_2^* - 1$



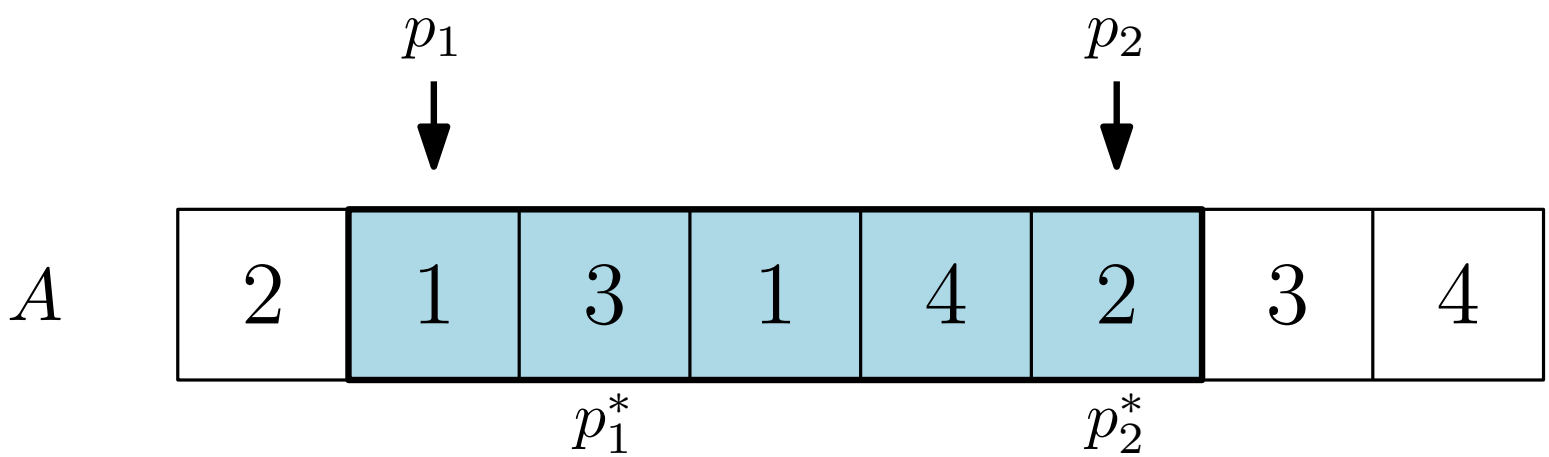
Sliding Window: Correctness

- Let $W^* = [p_1^*, p_2^*]$ be an optimal interval
- Consider the first instant where $p_1 = p_1^*$ or $p_2 = p_2^*$.
- If $p_1 = p_1^*$ then $p_2 < p_2^*$ and $A[p_1 \ldots p]$ contains no duplicates for all $p = p_2, \ldots, p_2^* - 1$
- Therefore, p_2 will be incremented until it reaches p_2^* while $p_1 = p_1^*$ remains constant $\Longrightarrow$ the algorithm considers W^*.



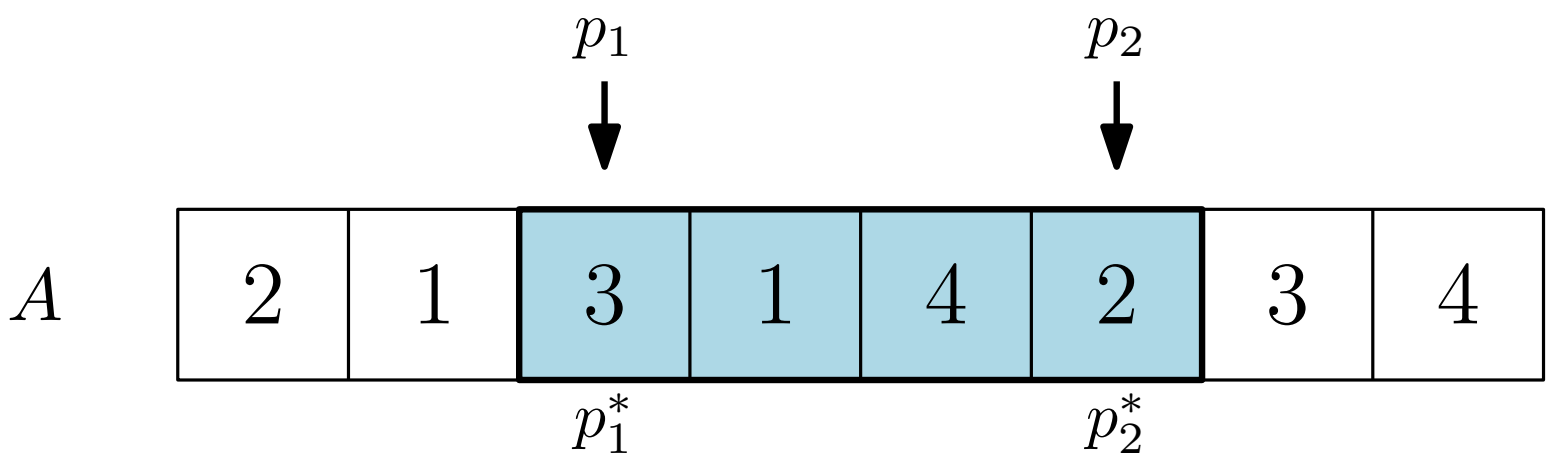
Sliding Window: Correctness

- Let $W^* = [p_1^*, p_2^*]$ be an optimal interval
- Consider the first instant where $p_1 = p_1^*$ or $p_2 = p_2^*$.
- If $p_2 = p_2^*$ then $p_1 < p_1^*$ and $A[p \dots p_2]$ contains duplicates for all $p = p_1, \dots, p_1^* - 1$



Sliding Window: Correctness

- Let $W^* = [p_1^*, p_2^*]$ be an optimal interval
- Consider the first instant where $p_1 = p_1^*$ or $p_2 = p_2^*$.
- If $p_2 = p_2^*$ then $p_1 < p_1^*$ and $A[p \dots p_2]$ contains duplicates for all $p = p_1, \dots, p_1^* - 1$
- Therefore, p_1 will be incremented until it reaches p_1^* while $p_2 = p_2^*$ remains constant $\Longrightarrow$ the algorithm considers W^*.



Sliding Window: Correctness

- We have proven that the algorithm always considers an optimal window.
- In fact we did not need to make any assumption about a specific optimal window in our proof...

Sliding Window: Correctness

- We have proven that the algorithm always considers an optimal window.
- In fact we did not need to make any assumption about a specific optimal window in our proof...
- Our algorithm discovers **all** optimal solutions!

Sliding Window: Correctness

- We have proven that the algorithm always considers an optimal window.
- In fact we did not need to make any assumption about a specific optimal window in our proof...
- Our algorithm discovers **all** optimal solutions!

Trick/Technique: Sliding Window

Some problems in which you need to find an interval can be solved in linear time using a sliding window approach, if you can ensure that an optimal interval will be considered.

A possible implementation

```
int left=0, right=-1, duplicates=0;
int best_len = -1;
std::vector<int> counts(n);

do
{
    if(duplicates==0 && right<n-1)
        duplicates += ( ++counts[A[++right]] >= 2 );
    else
        duplicates -= ( counts[A[left++]]-- >= 2 );

    if(duplicates==0)
        best_len = std::max(best_len, right-left+1);

} while(left<n-1 || right<n-1);

std::cout << "Gustavo can eat " << best_len << " dishes\n";
```

A possible implementation

```
int left=0, right=-1, duplicates=0;
int best_len = -1;
std::vector<int> counts(n);

do
{
    if(duplicates==0 && right<n-1)
        duplicates += ( ++counts[A[++right]] >= 2 );
    else
        duplicates -= ( counts[A[left++]]-- >= 2 );

    if(duplicates==0)
        best_len = std::max(best_len, right-left+1);

} while(left<n-1 || right<n-1);

std::cout << "Gustavo can eat " << best_len << " dishes\n";
```

Time: $O(n)$