Sum

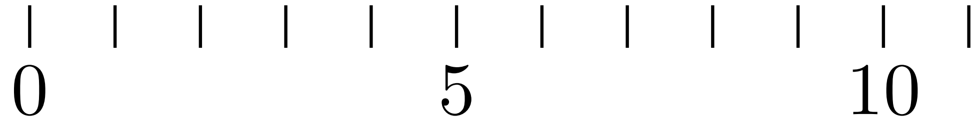
- Problem: Sum a given set of numbers
- Solution: Sum the numbers...

Gates

Problem: Given n intervals find the maximum number of overlapping intervals.

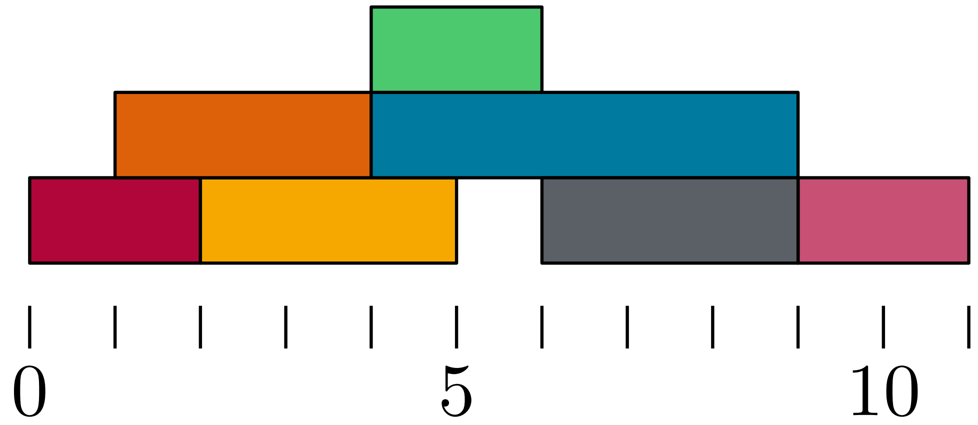
Gates

Problem: Given n intervals find the maximum number of overlapping intervals.



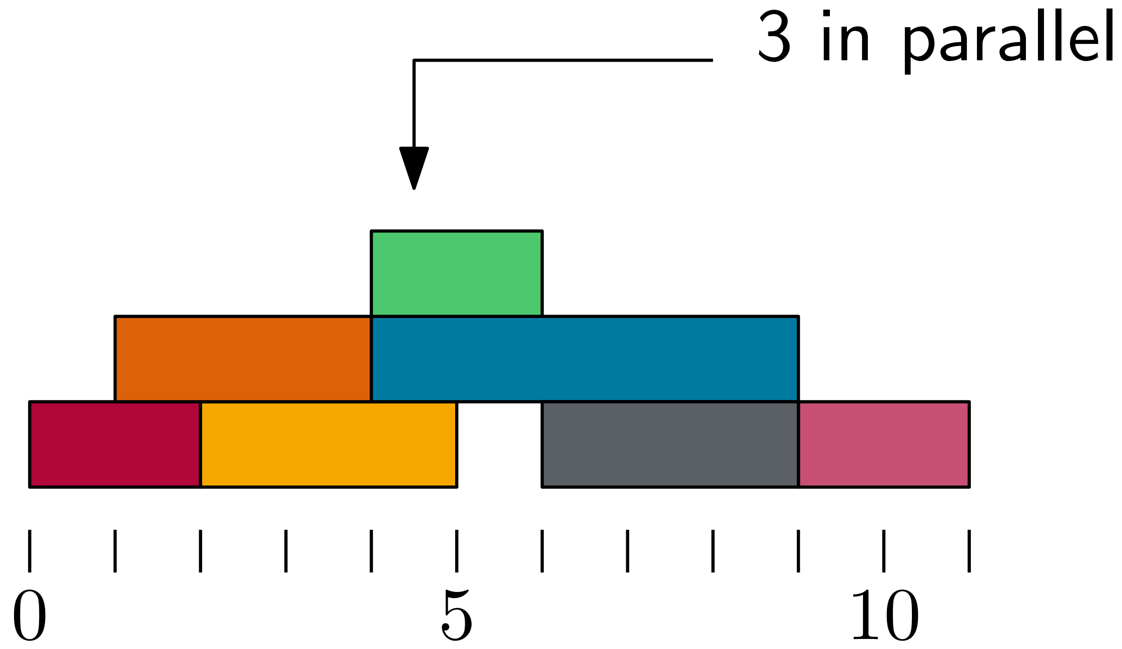
Gates

Problem: Given n intervals find the maximum number of overlapping intervals.



Gates

Problem: Given n intervals find the maximum number of overlapping intervals.



- Event scan!

- Event scan!
- For each interval $[a, b]$ ($a < b$) create two events $(a, 1)$ and $(b, -1)$.

- Event scan!
- For each interval $[a, b]$ ($a < b$) create two events $(a, 1)$ and $(b, -1)$.
- Sort them!

- Event scan!
- For each interval $[a, b]$ ($a < b$) create two events $(a, 1)$ and $(b, -1)$.
- Sort them!
- Initialize $s = s_{\mathsf{max}} = 0$.

- Event scan!
- For each interval $[a, b]$ ($a < b$) create two events $(a, 1)$ and $(b, -1)$.
- Sort them!
- Initialize $s = s_{\mathsf{max}} = 0$.
- Iterate over events (t, c):

- Event scan!
- For each interval $[a, b]$ ($a < b$) create two events $(a, 1)$ and $(b, -1)$.
- Sort them!
- Initialize $s = s_{\text{max}} = 0$.
- Iterate over events (t, c):
  - $s_{\text{max}} = \text{max}(s_{\text{max}}, s)$ if time passed since last event

- Event scan!
- For each interval $[a, b]$ ($a < b$) create two events $(a, 1)$ and $(b, -1)$.
- Sort them!
- Initialize $s = s_{\mathsf{max}} = 0$.
- Iterate over events (t, c):
  - $s_{\mathsf{max}} = \mathsf{max}(s_{\mathsf{max}}, s)$ if time passed since last event
  - $s := s + c$

- Event scan!
- For each interval $[a, b]$ ($a < b$) create two events $(a, 1)$ and $(b, -1)$.
- Sort them!
- Initialize $s = s_{\mathsf{max}} = 0$.
- Iterate over events (t, c):
  - $s_{\mathsf{max}} = \max(s_{\mathsf{max}}, s)$ if time passed since last event
  - $s := s + c$
- dominated by sorting: $\mathcal{O}(n \log n)$

Fairteam

Problem: Cut array into 3 pieces s.t. sums of the parts are as close together as possible. (minimize max - min)

Fairteam

Problem: Cut array into 3 pieces s.t. sums of the parts are as close together as possible. (minimize max - min)

- small n allows for n^2 or even n^3 brute-force

Homework

Problem: Given a global budget a, n friends with budget a_i and m papers with cost b_j, find maximum number of papers that can be finished. You can choose matching between friends and papers.

Homework

Problem: Given a global budget a, n friends with budget a_i and m papers with cost b_j, find maximum number of papers that can be finished. You can choose matching between friends and papers.

- to solve this for exactly k papers, take easiest papers, best friends and match greedily in sorted order

Homework

Problem: Given a global budget a, n friends with budget a_i and m papers with cost b_j, find maximum number of papers that can be finished. You can choose matching between friends and papers.

- to solve this for exactly k papers, take easiest papers, best friends and match greedily in sorted order
- thus, binary search over k to find maximum number of papers that still works

Homework

Problem: Given a global budget a, n friends with budget a_i and m papers with cost b_j, find maximum number of papers that can be finished. You can choose matching between friends and papers.

- to solve this for exactly k papers, take easiest papers, best friends and match greedily in sorted order
- thus, binary search over k to find maximum number of papers that still works
- the minimum time from your friends is the sum of the k easiest papers minus a

Never trust a greedy algorithm!

Order

- Why does sorting by l_i fail?

Order

- Why does sorting by l_i fail?
- Why does sorting by (l_i, r_i) fail?

Order

- Why does sorting by l_i fail?
- Why does sorting by (l_i, r_i) fail?
- Testcase `anti-greedy-3.in`:

```
3
1 3
1 3
2 2
```

Order

Problem: given a range $[l_i, r_i]$ of possible indices for every number $i \in [1, n]$, restore a permutation p matching these constraints ($l_{p_i} \leq i \leq r_{p_i}$ for all i).

Order

Problem: given a range $[l_i, r_i]$ of possible indices for every number $i \in [1, n]$, restore a permutation p matching these constraints ($l_{p_i} \le i \le r_{p_i}$ for all i).

- For index 1, only i's with $l_i \le 1$ should be considered

Order

Problem: given a range $[l_i, r_i]$ of possible indices for every number $i \in [1, n]$, restore a permutation p matching these constraints ($l_{p_i} \leq i \leq r_{p_i}$ for all i).

- For index 1, only i's with $l_i \leq 1$ should be considered
  - We can choose the one with the lowest r_i without blocking any options later

Order

Problem: given a range $[l_i, r_i]$ of possible indices for every number $i \in [1, n]$, restore a permutation p matching these constraints ($l_{p_i} \leq i \leq r_{p_i}$ for all i).

- For index 1, only i's with $l_i \leq 1$ should be considered
  - We can choose the one with the lowest r_i without blocking any options later

Why?

Order

Problem: given a range $[l_i, r_i]$ of possible indices for every number $i \in [1, n]$, restore a permutation p matching these constraints ($l_{p_i} \leq i \leq r_{p_i}$ for all i).

- For index 1, only i's with $l_i \leq 1$ should be considered
  - We can choose the one with the lowest r_i without blocking any options later

$\boxed{\text{Why?}}$

- Repeat . . .

Order

Problem: given a range $[l_i, r_i]$ of possible indices for every number $i \in [1, n]$, restore a permutation p matching these constraints ($l_{p_i} \leq i \leq r_{p_i}$ for all i).

- For index 1, only i's with $l_i \leq 1$ should be considered
  - We can choose the one with the lowest r_i without blocking any options later $\boxed{\text{Why?}}$
- Repeat ...
- Use `priority_queue`/`set` for available numbers

Order

Problem: given a range $[l_i, r_i]$ of possible indices for every number $i \in [1, n]$, restore a permutation p matching these constraints ($l_{p_i} \leq i \leq r_{p_i}$ for all i).

- For index 1, only i's with $l_i \leq 1$ should be considered
  - We can choose the one with the lowest r_i without blocking any options later

    Why?
- Repeat ...
- Use `priority_queue`/`set` for available numbers
- This is actually earliest-deadline-first scheduling!

Order

Problem: given a range $[l_i, r_i]$ of possible indices for every number $i \in [1, n]$, restore a permutation p matching these constraints ($l_{p_i} \leq i \leq r_{p_i}$ for all i).

- For index 1, only i's with $l_i \leq 1$ should be considered
  - We can choose the one with the lowest r_i without blocking any options later

    Why?
- Repeat ...
- Use `priority_queue`/`set` for available numbers
- This is actually earliest-deadline-first scheduling!
- Runtime: $\mathcal{O}(n \log n)$

Darkness

Problem: Given objects with positions and velocities on a line, what's the minimum time to make them meet in one point? You can place up to k portals.

Darkness

Problem: Given objects with positions and velocities on a line, what's the minimum time to make them meet in one point? You can place up to k portals.

- This is a typical application of *binary search the answer*.

Darkness

Problem: Given objects with positions and velocities on a line, what's the minimum time to make them meet in one point? You can place up to k portals.

- This is a typical application of *binary search the answer*.
- How to check if a given time is possible? We first consider the case $k = 0$.

Darkness

Problem: Given objects with positions and velocities on a line, what's the minimum time to make them meet in one point? You can place up to k portals.

- This is a typical application of *binary search the answer*.
- How to check if a given time is possible? We first consider the case $k = 0$.
- For every object keep the interval where it can be at time t and intersect all intervals. If the resulting interval is nonempty, we can choose a meeting point.

Darkness

Problem: Given objects with positions and velocities on a line, what's the minimum time to make them meet in one point? You can place up to k portals.

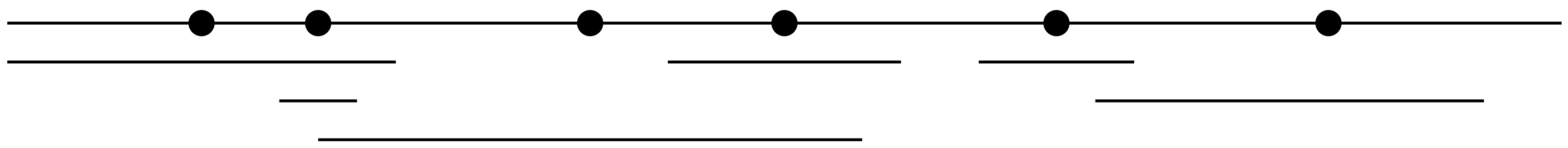
- This is a typical application of *binary search the answer*.
- How to check if a given time is possible? We first consider the case $k = 0$.
- For every object keep the interval where it can be at time t and intersect all intervals. If the resulting interval is nonempty, we can choose a meeting point.
- For general k, we have to partition the objects into $k + 1$ groups and place a portal in k of them.

darkness

- Use a scan-line approach and place portals to the goal whenever an interval closes:

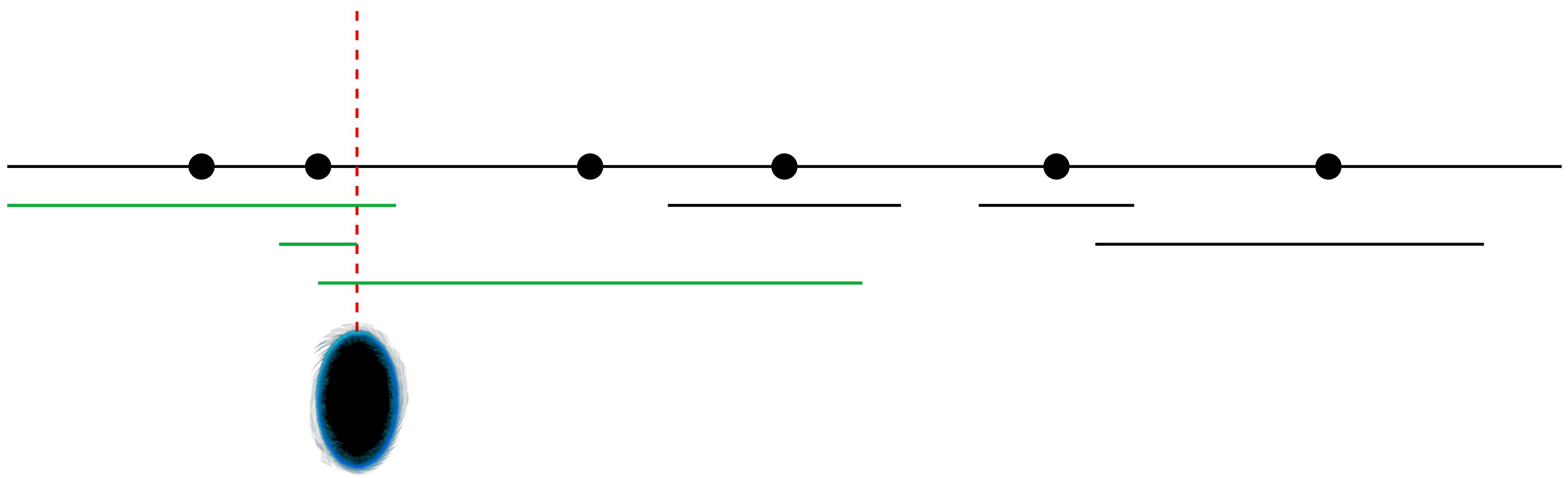
darkness

- Use a scan-line approach and place portals to the goal whenever an interval closes:



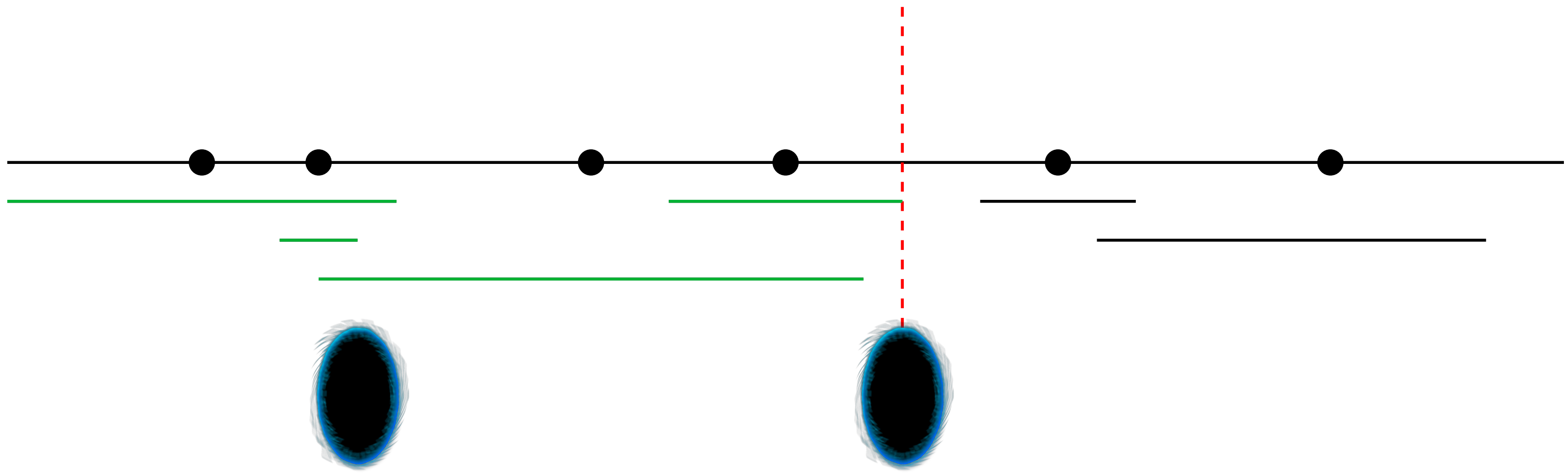
darkness

- Use a scan-line approach and place portals to the goal whenever an interval closes:



darkness

- Use a scan-line approach and place portals to the goal whenever an interval closes:



darkness

- Use a scan-line approach and place portals to the goal whenever an interval closes: