# Cumulative Scheduling with CP

# Scheduling problems

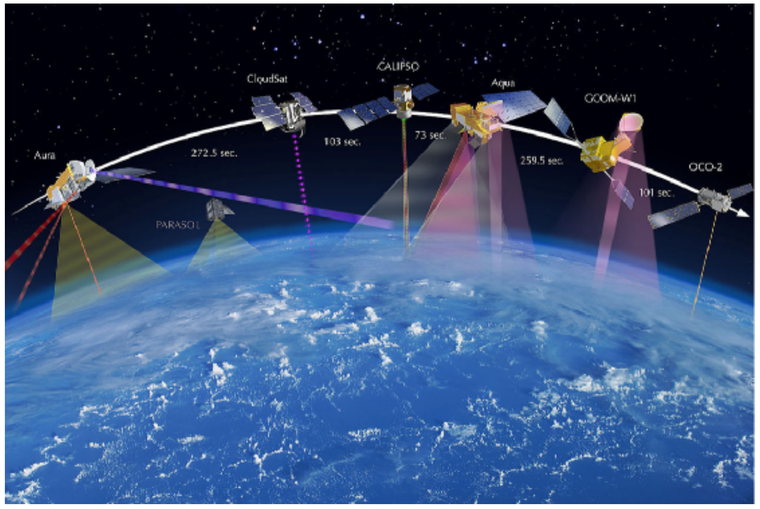
Scheduling = “Allocating scarce resources to activities over time” (Baker, 1974)

Assets of CP for solving scheduling problems:

- high-level flexible modeling abstractions for scheduling problems (activities, resource constraints, etc)
- strong filtering algorithms to prune the search tree
- good variable selection strategies during the search

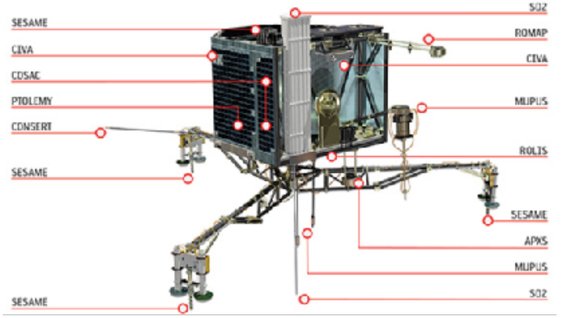
# Ex: File download from observation satellite

- Activities (aka jobs): file transfers
- Resources:
  - download channels: limited number of simultaneous downloads
  - memory banks: cannot simultaneously download files on the same memory bank
- Objective:
  - download as much data as possible within a given time window



# Ex: Planning Philae mission on comet 67P

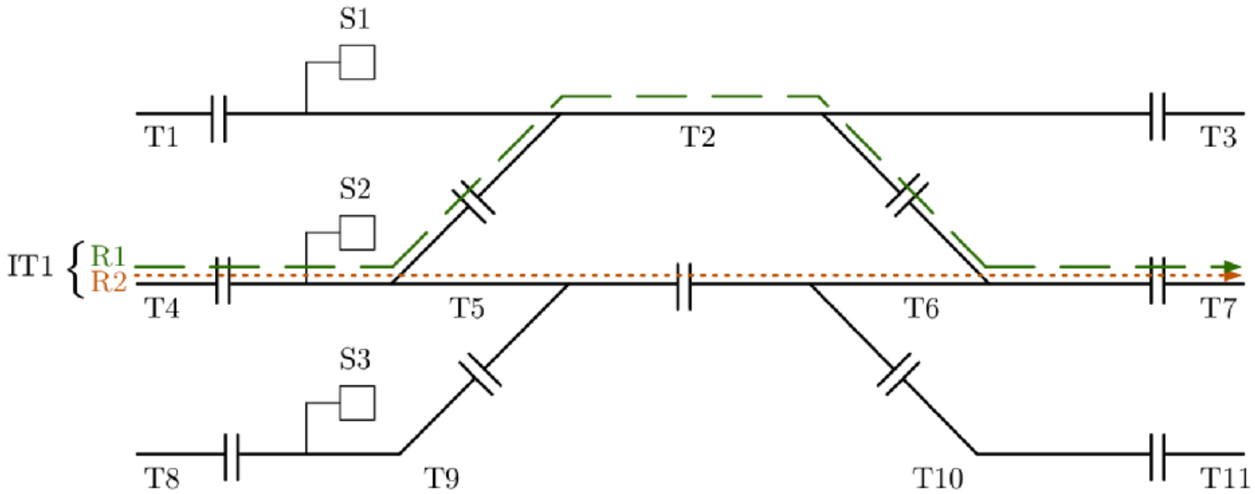
- Activities: scientific experiments
- Resources:
  - batteries: threshold on the instantaneous energy requirement
  - memory: experiments produce data; transfers are only possible when the spacecraft Rosetta (from which the lander module Philae is released) is visible
- Objective: maximize the lifespan of the batteries




[Hebrard, ACP Summer School 2017]

# Ex: Train scheduling with interlocking

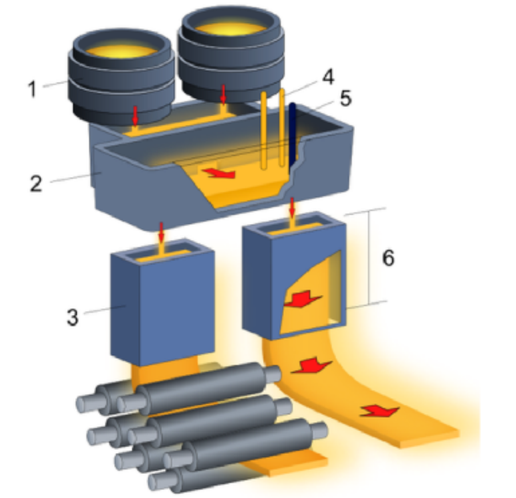
- Activities: trains going through a station, each with a set of possible routes
- Resources: track segments (only one train at a time, to avoid accidents)
- Objective: maximize the train throughput during a time window

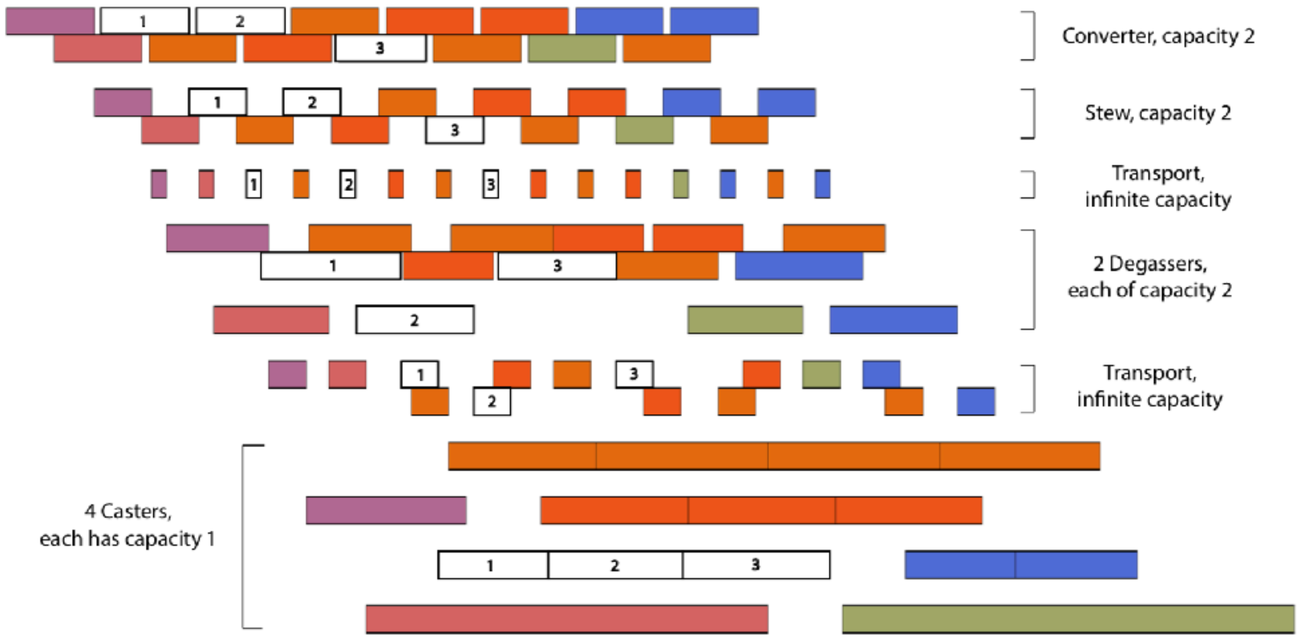



[Cappart & Schaus, CPAIOR 2017]

# Ex: Caster scheduling for steel production

- Activities: process pockets of molten metal (heat)
- Resources: converter, degasser, caster
- Objective: minimize the makespan

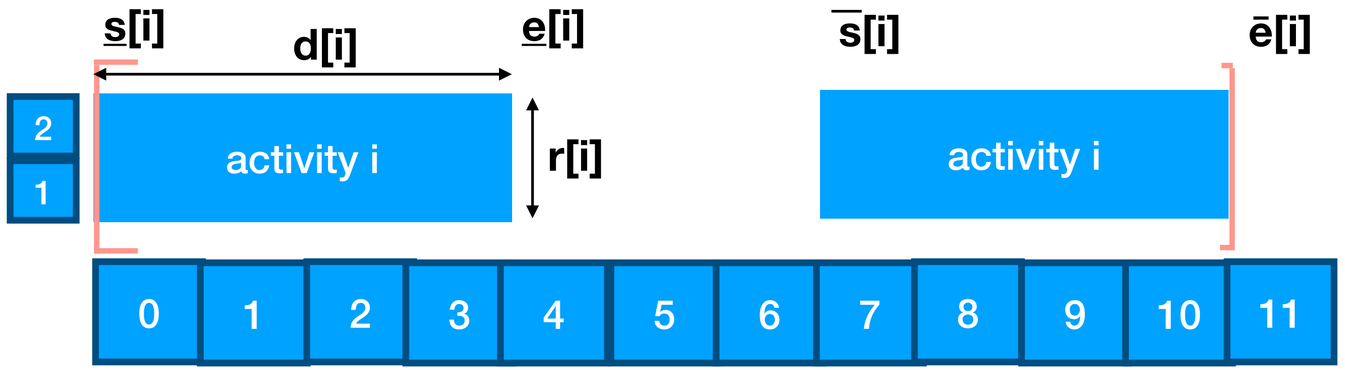



[Gay & Schaus, CP 2014]

# Decomposition of Cumulative

# Attributes of a cumulative activity




For this activity i:
s[i] ∈ [0,7] : start time
d[i] = 4 : duration
r[i] = 2 : resource requirement
e[i] = s[i] + d[i]  (a view):
activity i ends *just before* e[i]

# Cumulative constraint [Aggoun & Beldiceanu, 1993]

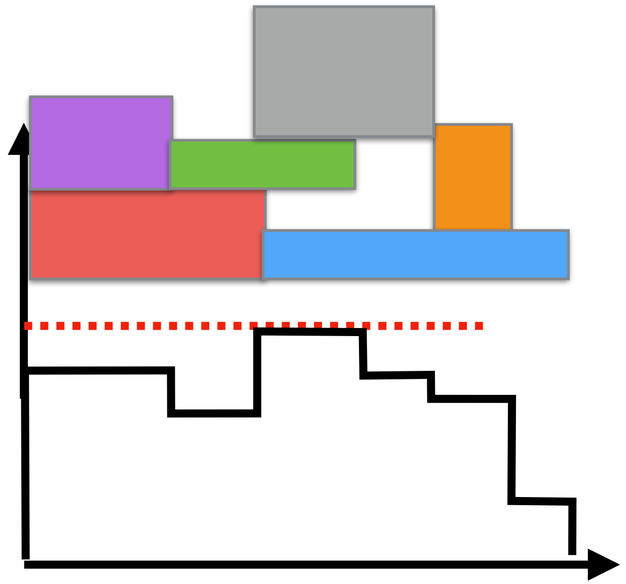
**public** Cumulative(IntVar[] s, int[] d, int[] r, int C)

start

duration

resource requirement

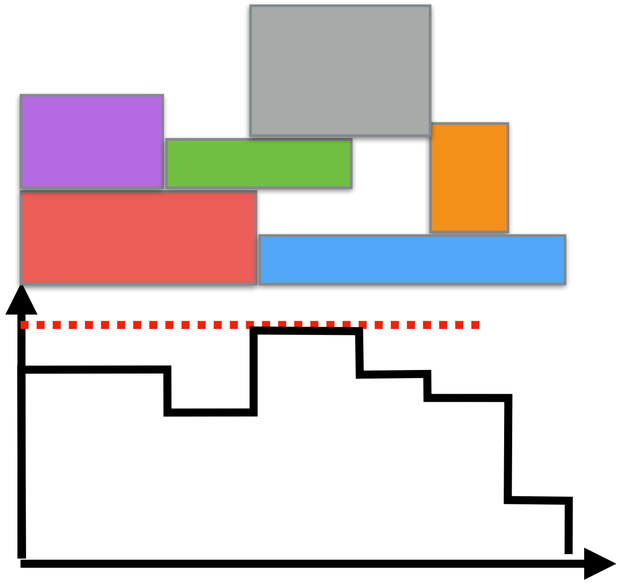
resource capacity



# Cumulative constraint

At any time *t*, the total resource requirement by the activities running at *t* does not exceed the resource capacity *C*:

$$\forall t : \left( \sum_{i : s_i \leq t < e_i} r_i \right) \leq C$$



# Reminder: Reified constraints

- $b \equiv x \leq y$ means that b is true iff $x \leq y$
- D(x) = {1,2,3} D(y)={0,1,5} D(b)={false, true}

# Reminder: Reified constraints

- $b \equiv x \leq y$ means that b is true iff $x \leq y$
- D(x) = {1,2,3} D(y)={0,~~1,5~~} D(b)={false, ~~true~~}

# Reminder: Reified constraints

- $b \equiv x \leq y$ means that b is true iff $x \leq y$
- D(x) = {1,2,3} D(y)={~~0~~,1,5} D(b)={~~false~~, true}

# Decomposition of Cumulative

Let *A* be the set of activities and *H* the time horizon of the project:

$$\forall t \in [0,H) : \left( \sum_{i \in A : s_i \leq t < e_i} r_i \right) \leq C$$

- Reify whether an activity runs at time *t*, encoding *false* as 0 and *true* as 1:

$$\forall i \in A : \forall t \in [0,H) : o_{i,t} \in \{0,1\} \wedge o_{i,t} \equiv s_i \leq t < e_i$$

- Enforce that the cumulated requirement over all activities is always at most the capacity *C*:

$$\forall t \in [0,H) : \sum_{i \in A} o_{i,t} \cdot r_i \leq C$$

# Decomposition of Cumulative

Consider the following example, where s[i] = {4,5,6}:


**Reified constraints for *i* at *t* = 6:**

$$s[i] \leq \bar{s}[i] = 6 < \underline{e}[i] \leq e[i] \equiv o_{i6} = 1$$

reified constraints

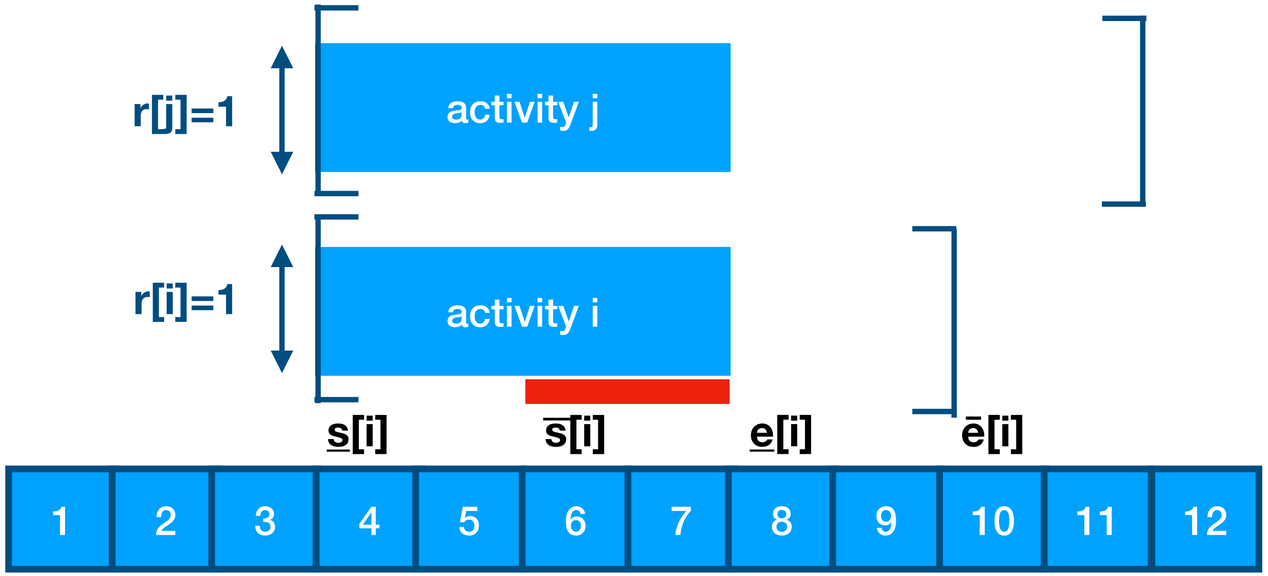
**Reified constraints for *i* at *t* = 7:**

$$s[i] \leq \bar{s}[i] \leq 7 < 8 = \underline{e}[i] \leq e[i] \equiv o_{i7} = 1$$

reified constraints

- If activity *i* starts late (at 6), then it overlaps the interval [6,7]
- If activity *i* ends early (at 8), then it overlaps the interval [6,7]

# How does the decomposition work?



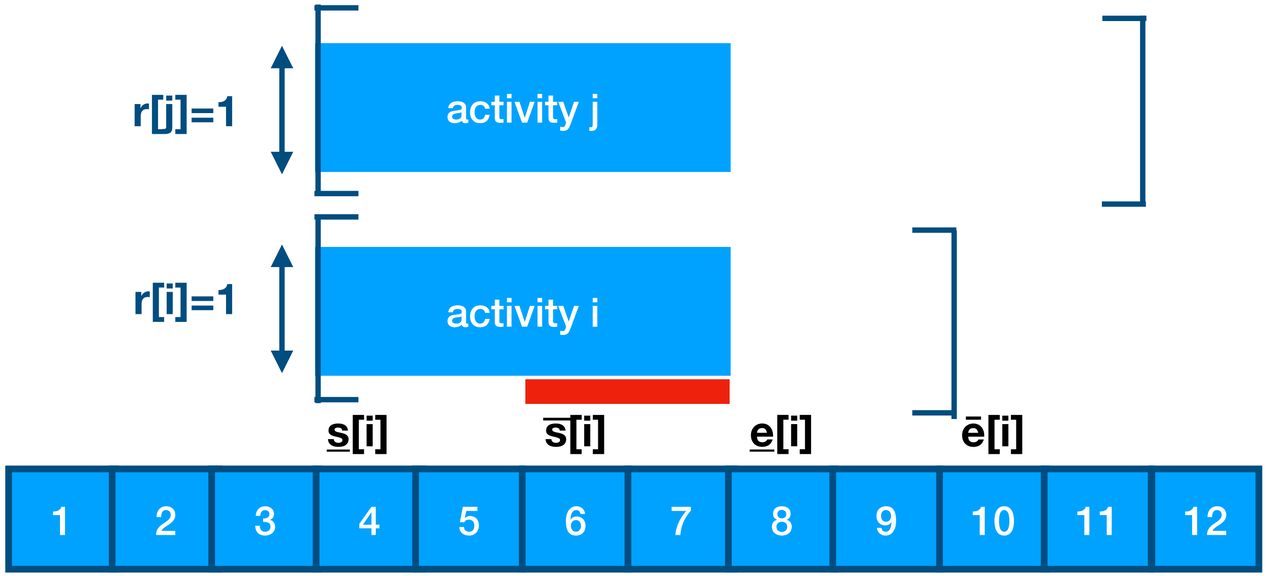

C=1 (i.e., the two activities cannot overlap)

$$s[i] \in [4,6],\ e[i] \in [8,10],\ d[i] = 4$$
$$s[j] \in [4,8],\ e[j] \in [8,12],\ d[j] = 4$$

**Reified constraints for *i* at *t* = 6 and *t* = 7:**

$$s[i] \leq \bar{s}[i] = 6 \leq 6 < 8 = \underline{e}[i] \leq e[i] \equiv o_{i6} = 1$$
$$s[i] \leq \bar{s}[i] = 6 \leq 7 < 8 = \underline{e}[i] \leq e[i] \equiv o_{i7} = 1$$

# How does the decomposition work?




C=1 (i.e., the two activities cannot overlap)

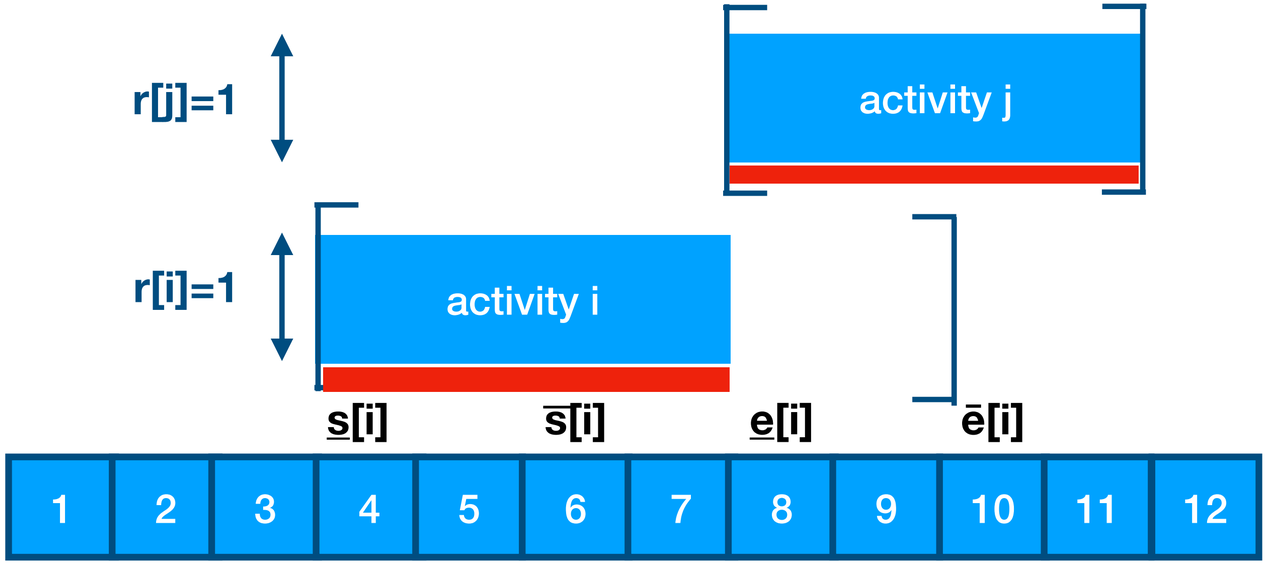
**Sum constraint at *t* = 6:**

$$o_{i6} \cdot r[i] + o_{j6} \cdot r[j] \leq 1 \Leftrightarrow 1 + o_{j6} \leq 1 \Leftrightarrow o_{j6} = 0 \Leftrightarrow s[j] > 6, \text{ as } e[j] \in [8,12]$$

**Sum constraint at *t* = 7:**

$$o_{i7} \cdot r[i] + o_{j7} \cdot r[j] \leq 1 \Leftrightarrow 1 + o_{j7} \leq 1 \Leftrightarrow o_{j7} = 0 \Leftrightarrow s[j] > 7, \text{ as } e[j] \in [8,12]$$

# How does the decomposition work?




C=1 (i.e., the two activities cannot overlap)

$$s[j] \in \{8\} \textbf{ and } e[j] \in \{12\}$$

**Reified constraints for *j* at *t* = 8, 9, 10, and 11:**

$$o_{j8} = o_{j9} = o_{j10} = o_{j11} = 1$$

**Sum constraint at t = 8:**

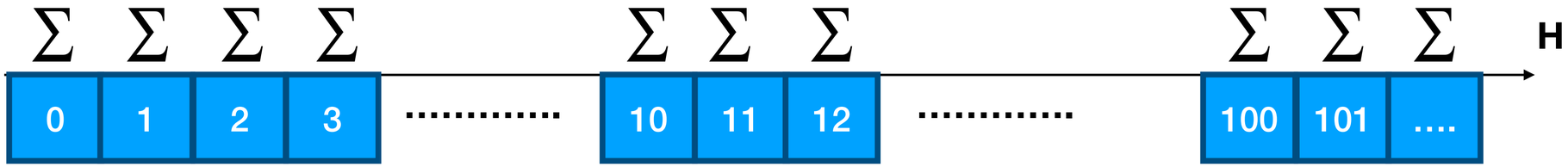
$$o_{i8} \cdot r[i] + o_{j8} \cdot r[j] \leq 1 \Leftrightarrow o_{i8} = 0 \Leftrightarrow e[i] \leq 8$$

**Sum constraint at t = 9:**

$$o_{i9} \cdot r[i] + o_{j9} \cdot r[j] \leq 1 \Leftrightarrow o_{i9} = 0 \Leftrightarrow e[i] \leq 9$$

# Drawback of the decomposition

- Discretization of time + reified constraint at each time
- Heavy fixpoint computation: runtime proportional to H




- H = 10 hours, n = 10 activities
- Time unit = 1 minute: 6,000 variables and 600 sum constraints 😢
- Time unit = 1 second: 360,000 variables and 36,000 sum constraints 😭

# TimeTable filtering

We would prefer, for $n$ activities:

- To limit the number of variables and make sure it depends only on $n$
- To compute the fixpoint with a time complexity that depends only on $n$

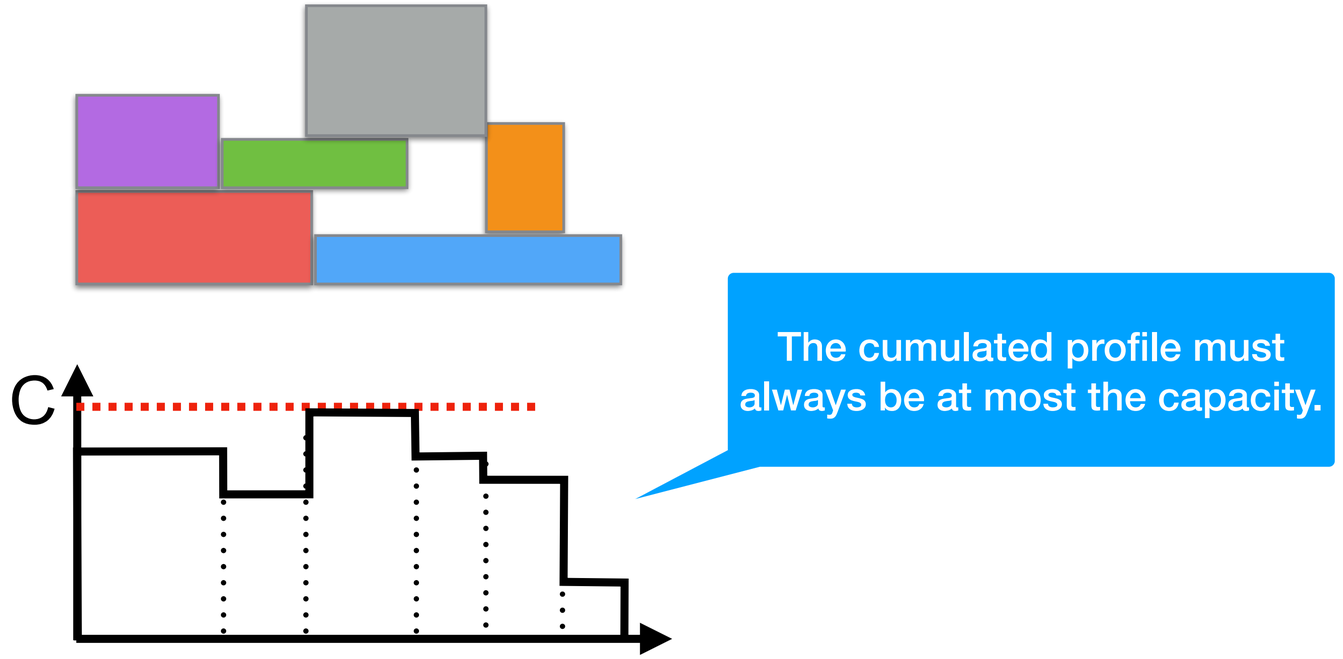
TimeTable filtering is used to do the same filtering but faster: $O(n^2)$ time.

# TimeTable Filtering for Cumulative: Checking Feasibility with a Profile

# Checker

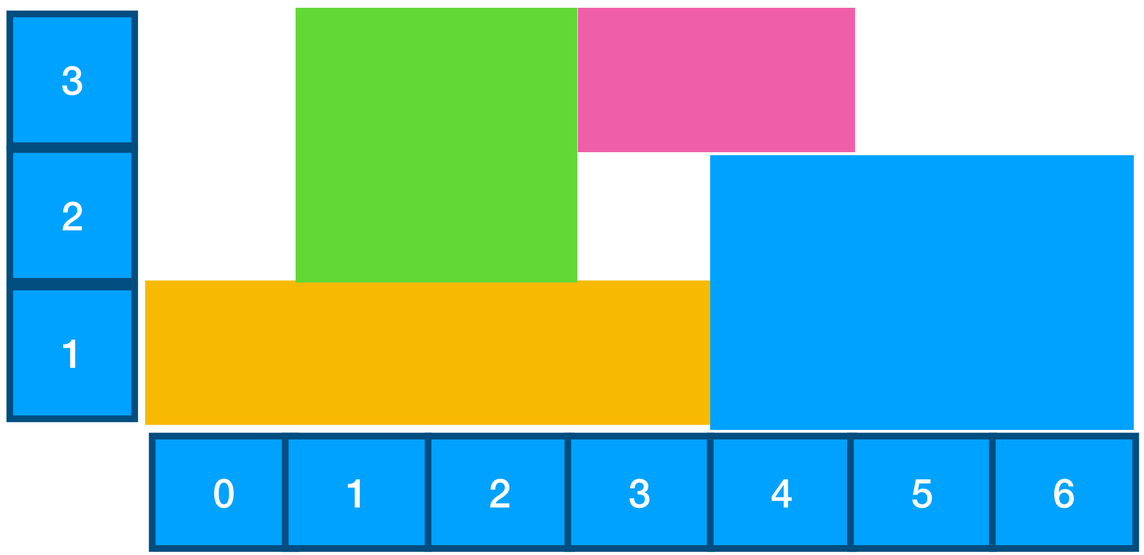
Assume for now that all the start variables are fixed:

- How do we check if a Cumulative constraint is satisfied?
- What is the time complexity?



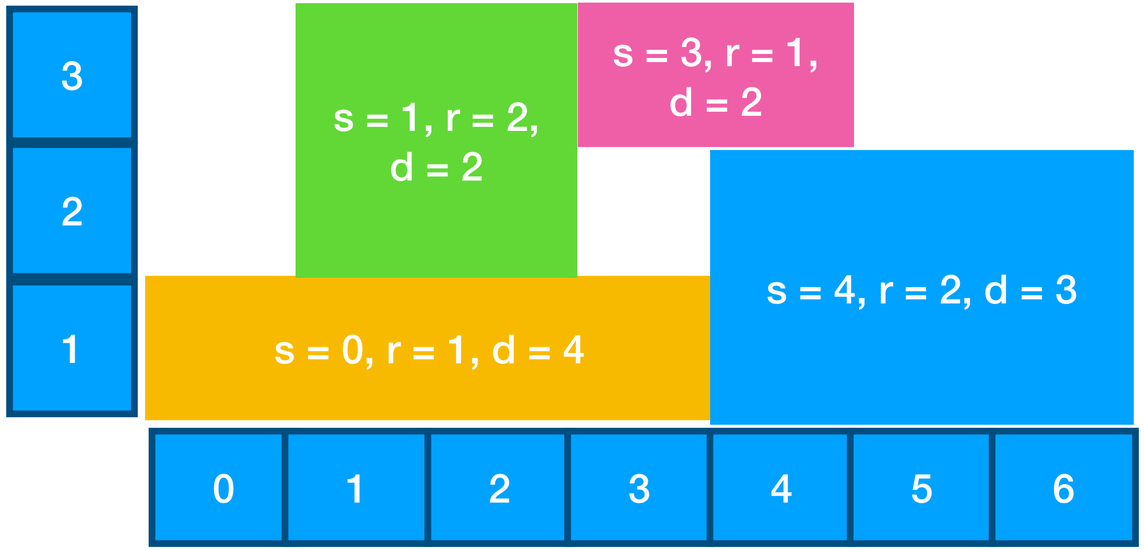
# Checker: Approach

- For *n* activities, the cumulated profile has at most 2*n* rectangles.
- Build the cumulated profile
  and check that none of its rectangles exceeds the capacity.




# Checker: Activity Events

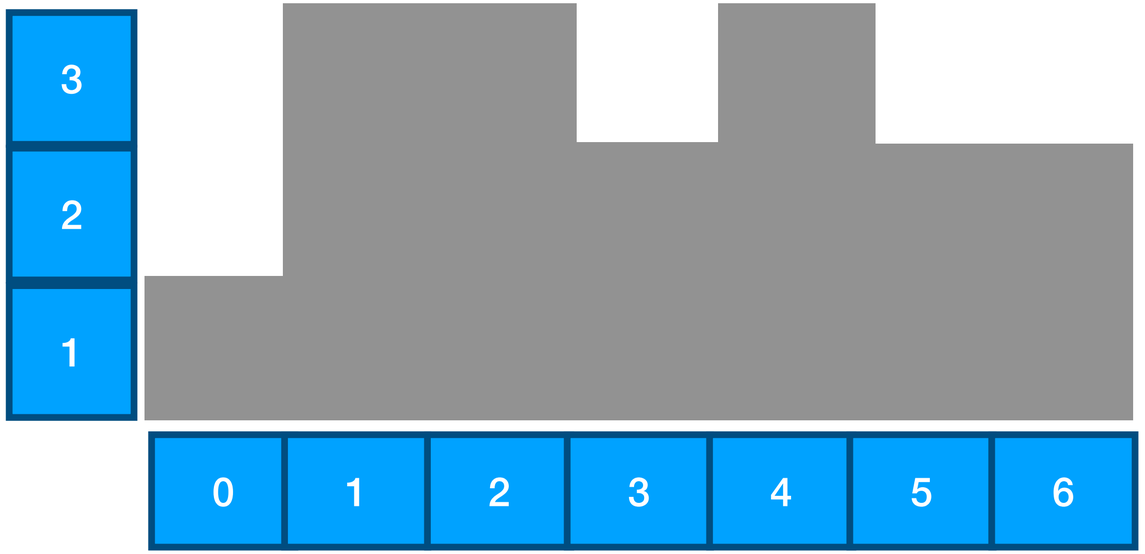
| t | 0 | 4 | 1 | 3 | 3 | 5 | 4 | 7 |
|---|---|---|---|---|---|---|---|---|
| h | 1 | -1 | 2 | -2 | 1 | -1 | 2 | -2 |




- Public class Event(int t, int h)
- For start event: t = start, h = requirement; for end event: t = end, h = -requirement
- Create events: Event(0,1), Event(4,-1), Event(1, 2), Event(3,-2), Event(3,1), Event(5,-1), Event(4,2), Event(7,-2)
- Sort events following time: Event(0,1), Event(1, 2), Event(3,-2), Event(3,1), Event(4,-1), Event(4,2), Event(5,-1), Event(7,-2)

# Checker: Activity Events

| t | 0 | 1 | 3 | 3 | 4 | 4 | 5 | 7 |
|---|---|---|---|---|---|---|---|---|
| h | 1 | 2 | -2 | 1 | -1 | 2 | -1 | -2 |
| Height | 1 | 3 | | 2 | | 3 | 2 | 0 |




- We iterate over events and compute the cumulated height at all times t
- It takes O(*n*) time to process all events
- Between two adjacent times with different cumulated heights a new rectangle of the profile is created.
- It takes O(n log n) time to sort all events
- So the overall time complexity of the checker is O(n log n)

# Let’s code this algorithm

```
public class Profile {
     static class Event {
        private final int t;
        private final int h;
    }
    static class Rectangle {
        private final int start;
        private final long dur;
        private final int height;
        private final int end;
    }

    private final Rectangle[] profileRectangles;

    public Profile(Rectangle... rectangles) {
        // compute the profile rectangles in two steps
        // step1: create timeline
        // step2: sweep
        this.profileRectangles = profile.toArray(new Rectangle[0]);
    }
}
```

# Step 1: Create timeline

```
public class Profile {

    private final Rectangle[] profileRectangles;

    public Profile(Rectangle... rectangles) {
        // step1: create timeline (slide before)
        ArrayList<Rectangle> profile = new ArrayList<Rectangle>();
        Event[] events = new Event[2 * rectangles.length + 2];
        for (int i = 0; i < rectangles.length; i++) {
            Rectangle r = rectangles[i];
            events[i] = new Event(r.start, r.height);// start events
            events[rectangles.length + i] = new Entry(r.end, -r.height); // end events
        }
        points[2 * rectangles.length] = new Event(Integer.MIN_VALUE, 0); //dummy start
        points[2 * rectangles.length + 1] = new Event(Integer.MAX_VALUE, 0); //dummy end

        Arrays.sort(events);
        // step2: sweep (next slides)
        this.profileRectangles = profile.toArray(new Rectangle[0]);
    }
}
```

Two dummy entries

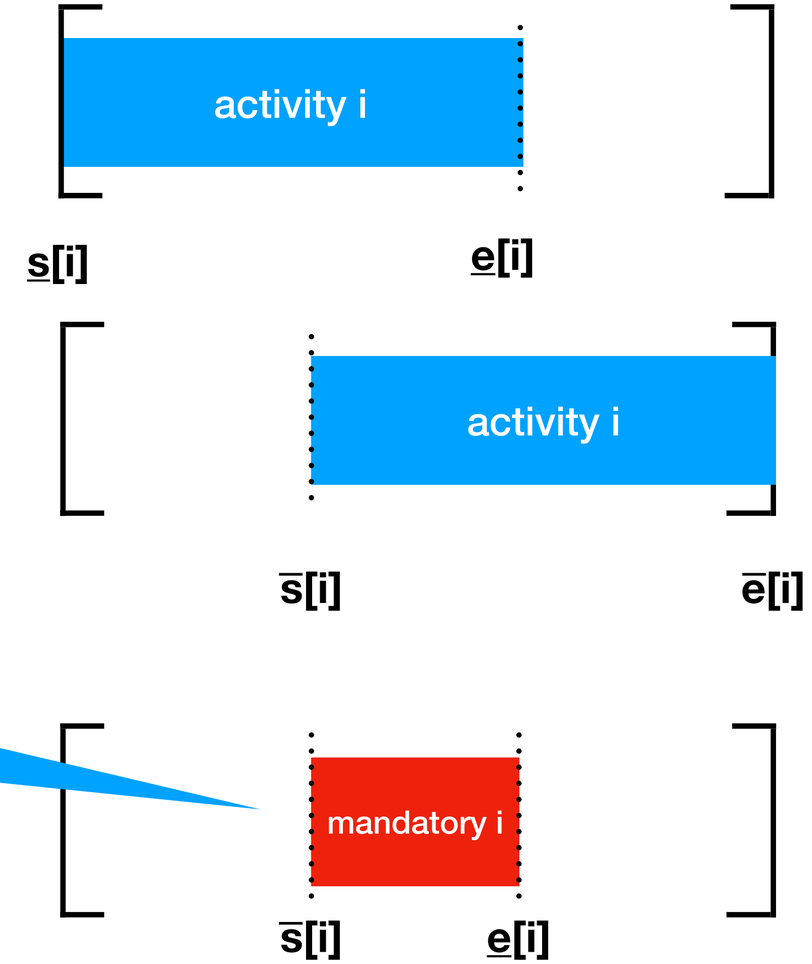
# Step 2: Sweep

```
public class Profile {

    private final Rectangle[] profileRectangles;

    public Profile(Rectangle... rectangles) {
        // step1: create timeline (previous slide …)
        // step2: sweep
        int sweepHeight = 0;
        int sweepTime = points[0].key;
        for (Event e : events) {
            int t = e.key;
            int h = e.value;
            if (t != sweepTime) {
               profile.add(new Rectangle(sweepTime, t - sweepTime, sweepHeight, t));
               sweepTime = t;
            }
            sweepHeight += h;
        }
        this.profileRectangles = profile.toArray(new Rectangle[0]);

    }
}
```

# TimeTable Filtering for Cumulative: Filtering Time Bounds

# Activity: Definitions

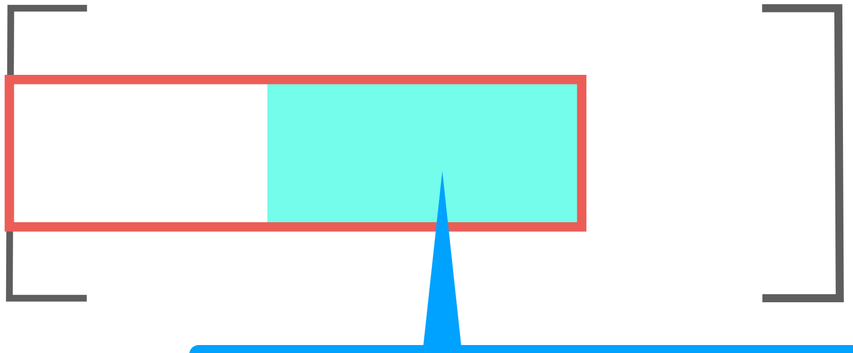
At any time *t*, the total requirement by the activities running at *t* does not exceed the capacity *C*.




**The mandatory part of activity i only exists if e[i] > s̄[i].**

# Mandatory profile

In practice, during search, not all the start variables are fixed yet,
so there is some flexibility when activities can be scheduled:

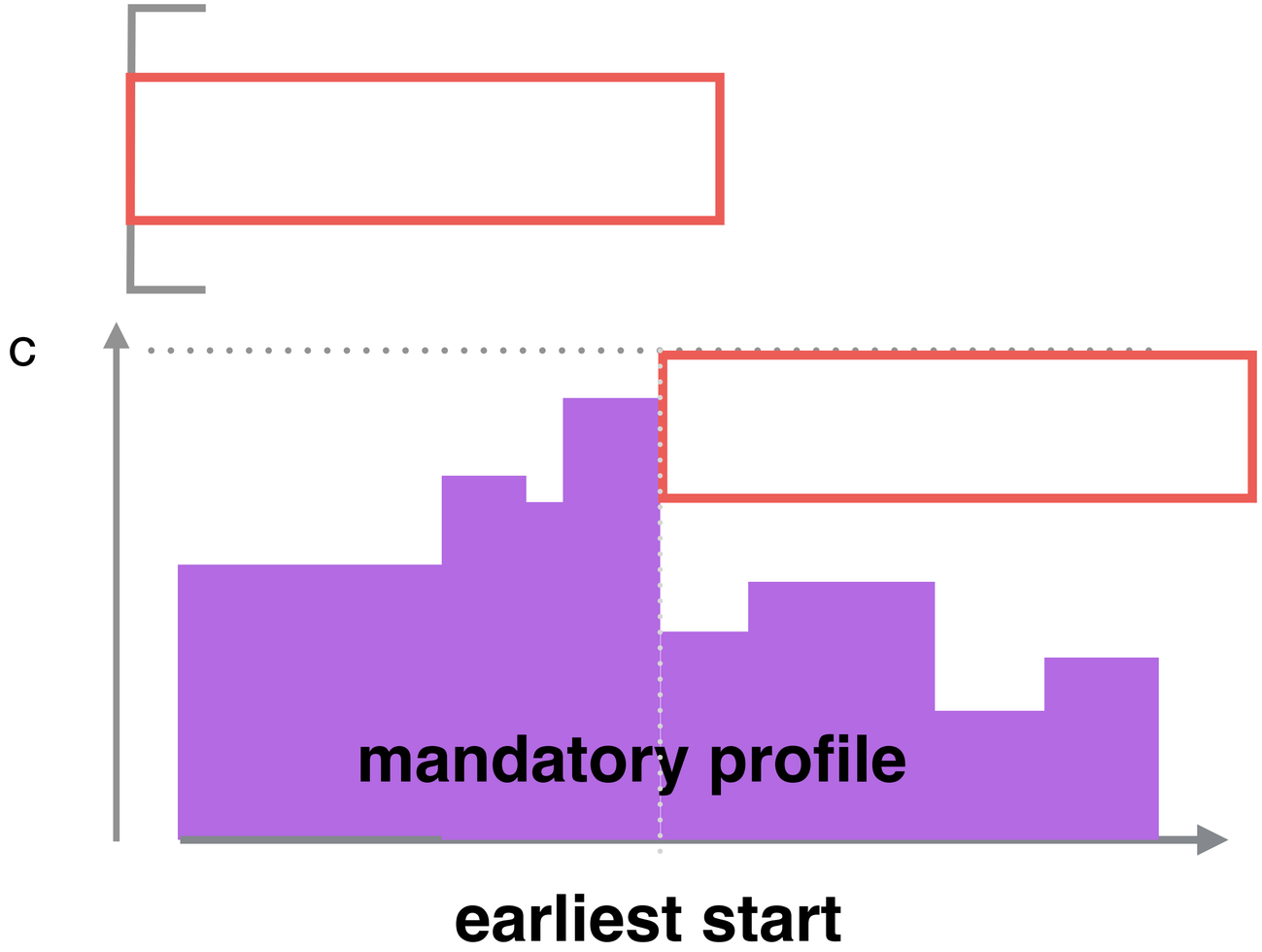


**Mandatory part:** we are sure that this activity will run during this time interval whatever its eventual start time.
Not every activity has a mandatory part.

The mandatory profile is optimistic,
as it is built solely from the mandatory parts of the activities.

# Timetable filtering

Update of the lower bound: update the minimum start time to the earliest time where it is not in conflict with the mandatory profile.



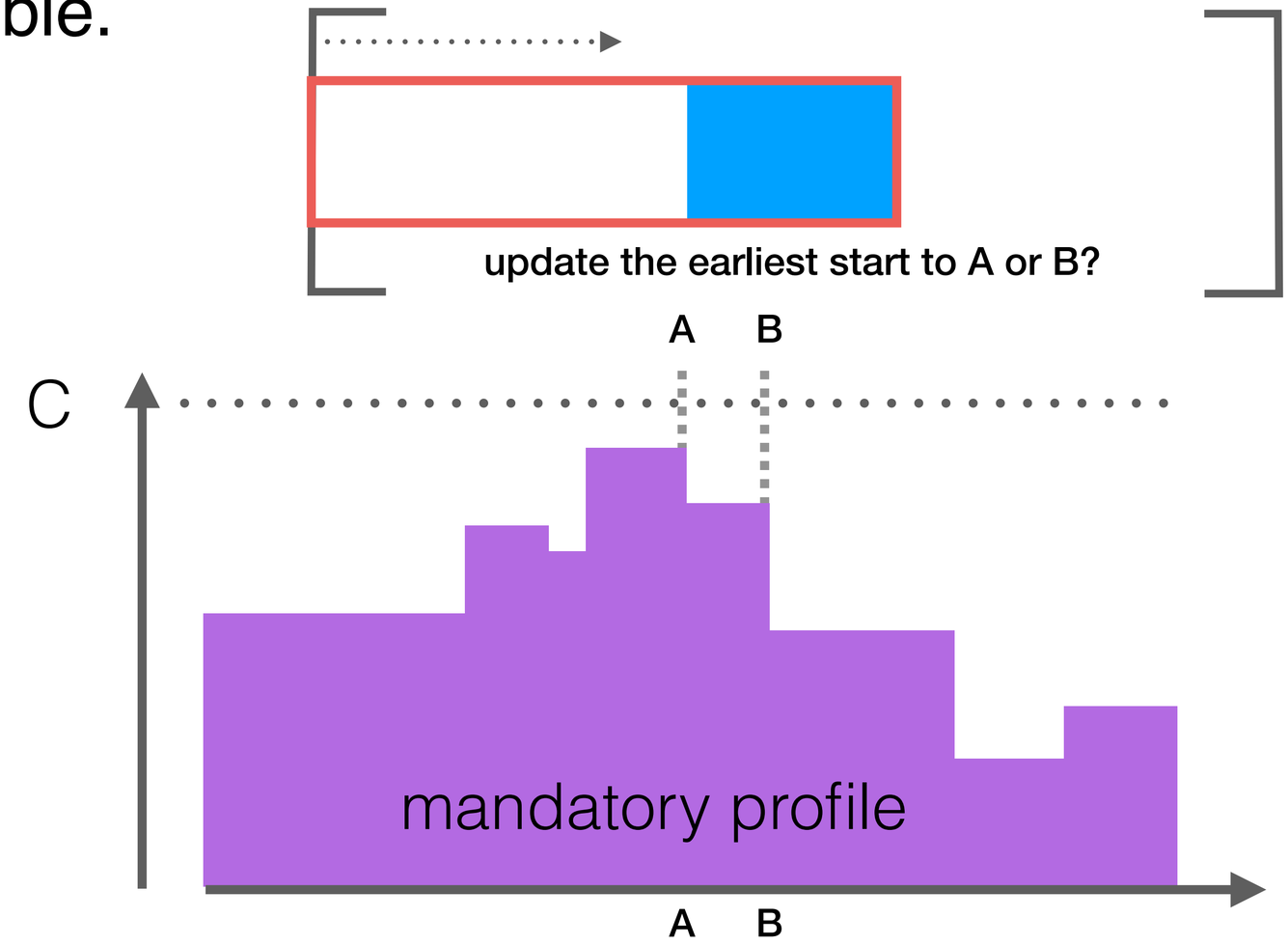

Now with mandatory part: update mandatory profile!

Time complexity?

O(n) time per activity since the mandatory profile has O(n) intervals: $O(n^2)$ time overall.

# Be careful with activities with mandatory part

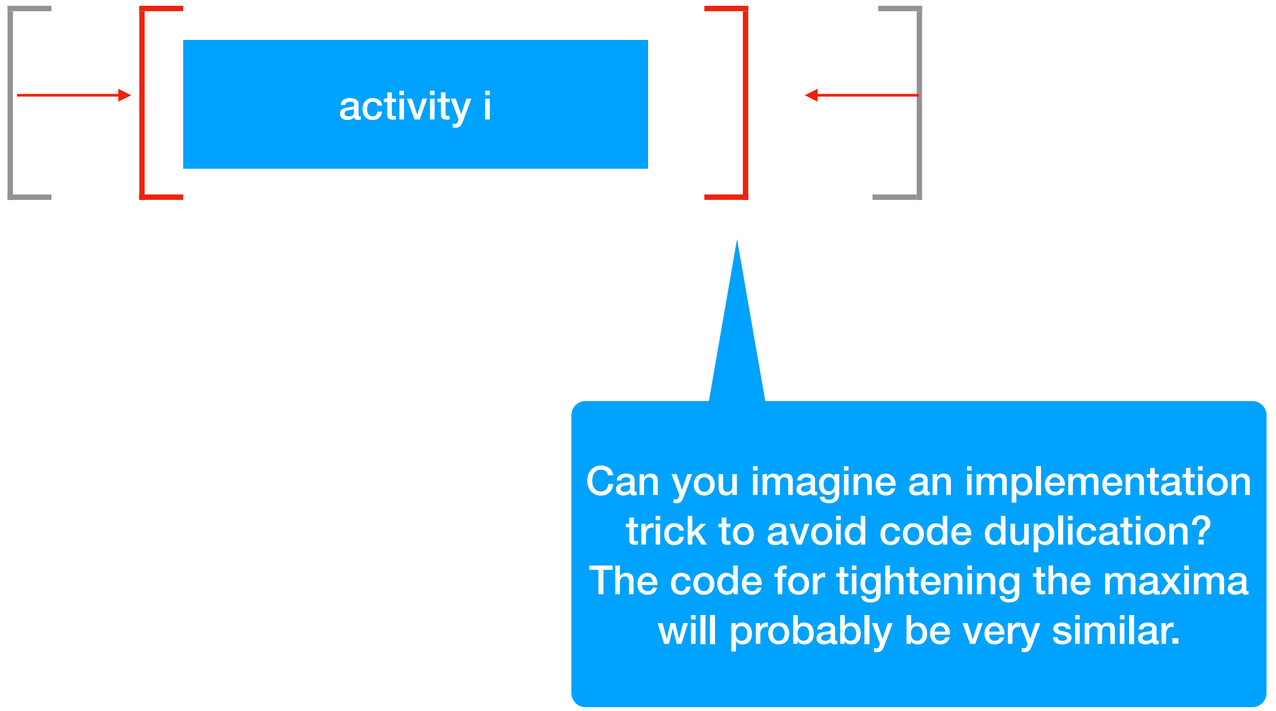
Do not push the start of an activity beyond its mandatory part (which would be infeasible), because the latter is in the mandatory profile and thus proven to be feasible.



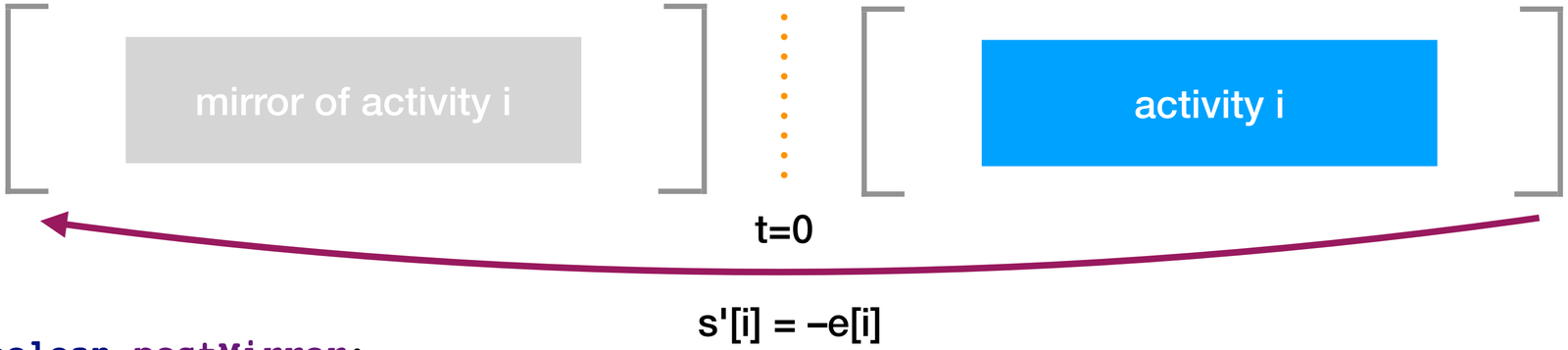
# TimeTable Filtering for Cumulative: Implementation Trick for Simplifying the Code

# Avoid code duplication

- We update the earliest start times of activities.
- Can we also update their latest completion times?



# Mirroring of activities

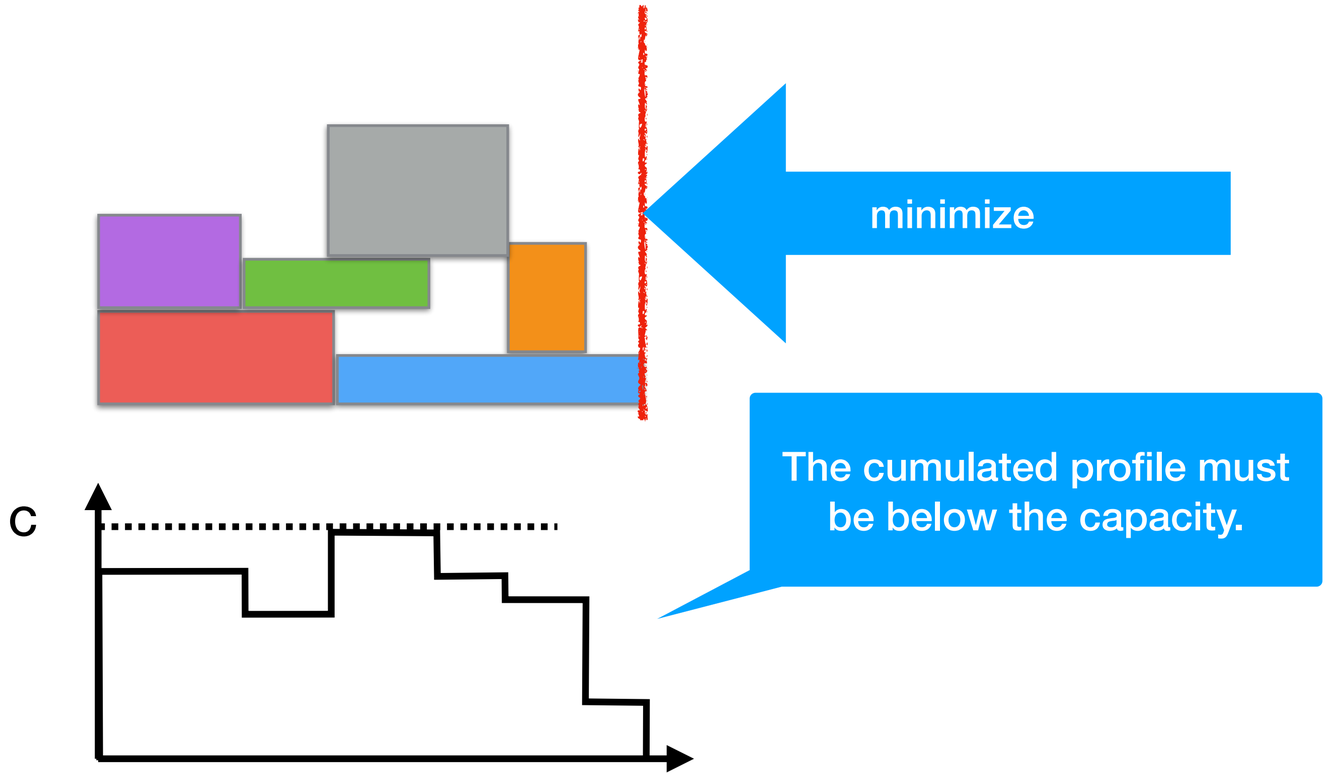



```
private final boolean postMirror;

public Cumulative(IntVar[] start, int[] duration, int[] requirement, int capa) {
    this(start, duration, requirement, capa, true);
}
private Cumulative(IntVar[] start, int[] duration, int[] requirement, int capa, boolean postMirror) {
    // …
}
@Override
public void post() {
    for (int i = 0; i < start.length; i++) {
        start[i].propagateOnBoundChange(this);
    }

    if (postMirror) {
        IntVar[] startMirror = Factory.makeIntVarArray(start.length, i -> minus(end[i]));
        getSolver().post(new Cumulative(startMirror, duration, demand, capa, false), false);
    }
    propagate();
}
```

# LNS for Cumulative Scheduling

# LNS for scheduling

- Assume makespan minimization, where
  makespan = latest completion time of all the activities.
- Decision variables = start times of the activities.



# LNS for scheduling

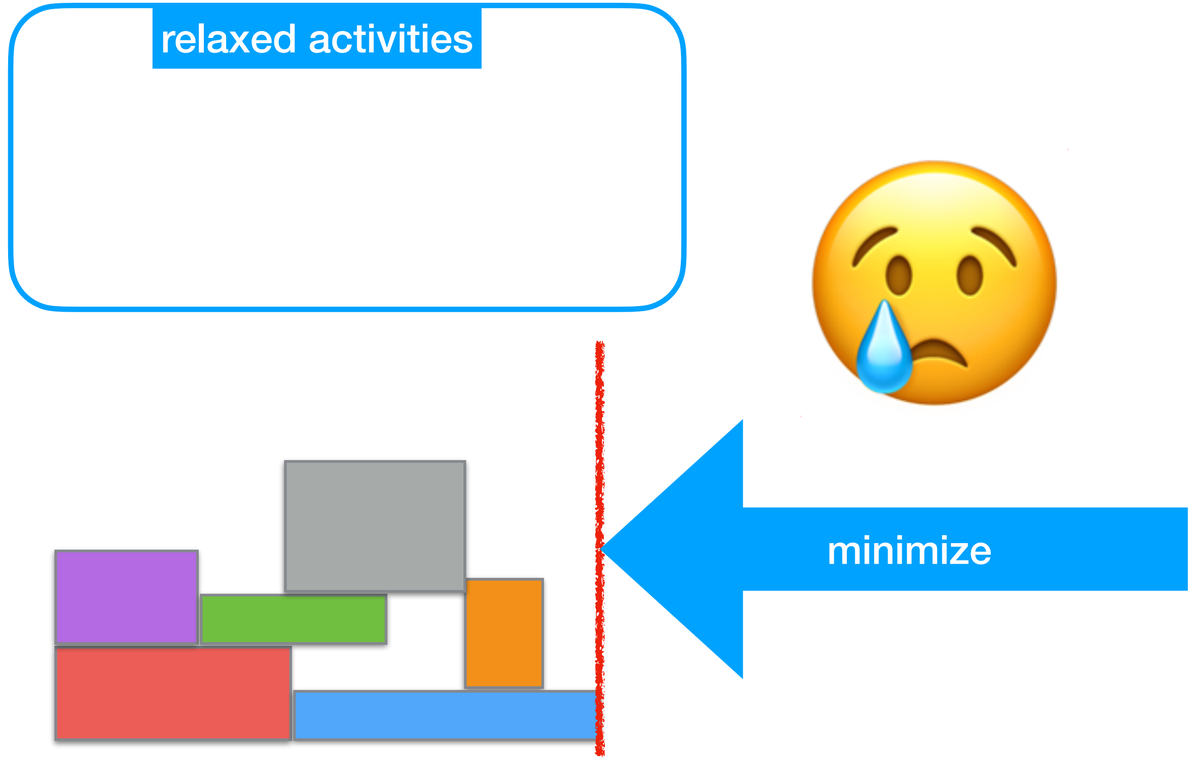
```
int nRestarts = 1000;
int failureLimit = 100;
Random rand = new java.util.Random(0);

for (int i = 0; i < nRestarts; i++) {
    if (i % 10 == 0)
        System.out.println("restart number #" + i);

    dfs.optimizeSubjectTo(obj, statistics -> statistics.numberOfFailures() >= failureLimit, () -> {
                // Assign the fragment 5% of the variables randomly chosen
                for (int j = 0; j < n; j++) {
                    if (rand.nextInt(100) < 75) {
                        // after the solveSubjectTo those constraints are removed
                        cp.post(equal(x[j], xBest[j]));
                    }
                }
            }
    );
}
```

Do you think the LNS relaxation we used for the QAP would be good for scheduling?

# LNS for scheduling

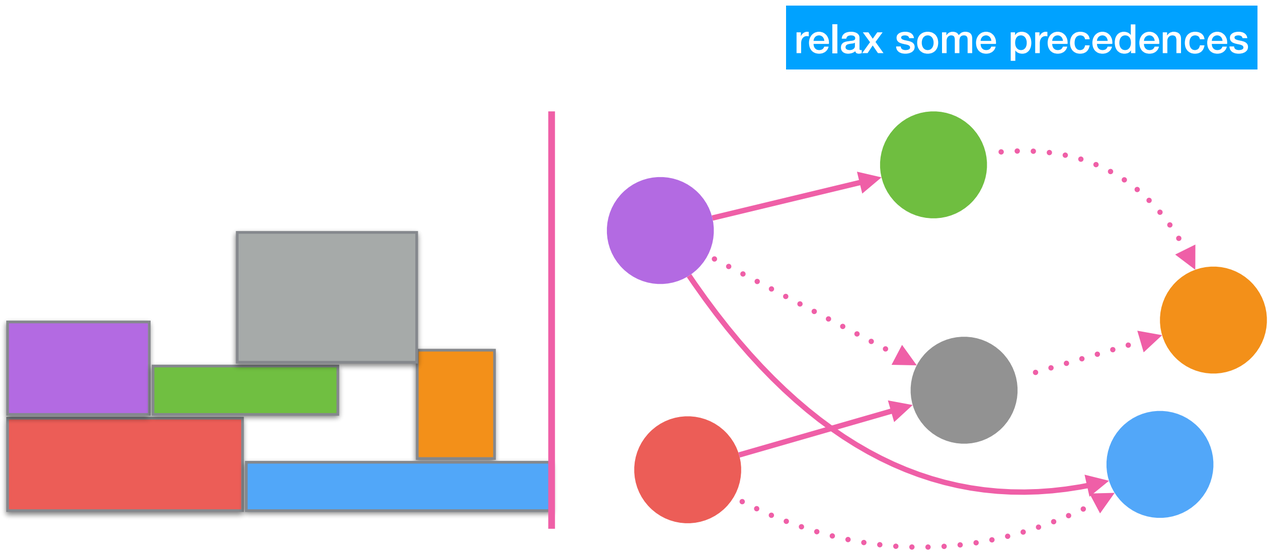
- Fixing some variables is a bad idea for scheduling problems because there is almost zero chance to improve the objective value, and there is a very high chance to reconstruct exactly the same solution as before.

- It is a too rigid relaxation for scheduling problems.
  We rather need to be able to *reorder* activities.



# Partial-order schedule

- *Do not* fix the start times of activities at the next restart.
- Instead keep part of the structure (the relative positions of activities) of the current best solution.  For instance, preserve some precedences that are observed in the current best solution.
- This is the notion of partial-order schedule [Laborie and Godard 2005].



# Generalizations of Cumulative Scheduling

# Generalizations of Cumulative

So far, we have assumed that only each start time was a variable,
but in practice…

- … the durations of activities can also be variables:
  - Use timetable filtering but with min(d[i]) instead of d[i] for the filtering of the s[i].
- … an activity can optionally execute or not on the resource:
  - Use a Boolean variable to represent the status, and it must also be filtered.

# Producer-Consumer Problems

- An initial quantity $Q_0$ is available at time 0.
- Each producer $P_i$ produces a given quantity $Q_{P_i}$ at time $T_{P_i}$ (a variable).
- Each consumer $C_j$ consumes a given quantity $Q_{C_j}$ at time $T_{C_j}$ (a variable).
- Constraint: At any time, the remaining quantity is non-negative:

$$\forall t : Q_0 + \sum_{i:T_{P_i} \leq t} Q_{P_i} - \sum_{j:T_{C_j} \leq t} Q_{C_j} \geq 0$$

# Naïve Model with Decomposition

$S_t$ = quantity available at time $t$ :

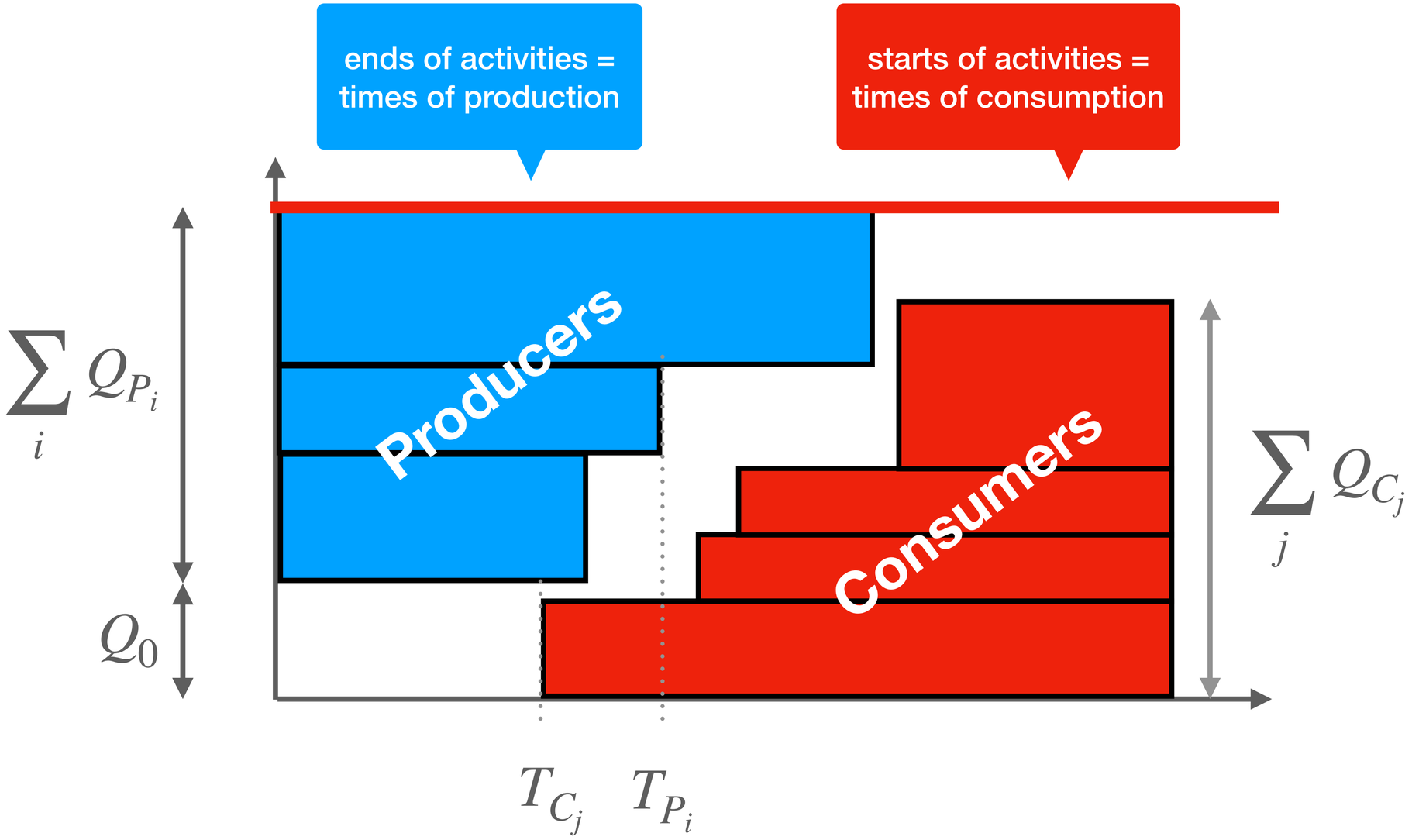
$$S_0 = Q_0$$

$$\forall t : S_t \geq 0$$

$$\forall t : S_{t+1} = S_t + \sum_{i:T_{P_i}=t} Q_{P_i} - \sum_{j:T_{C_j}=t} Q_{C_j}$$

Again a sum of reifications of constraints for each time step.
Can we do better than this heavy decomposition?

# Model with a Cumulative Constraint



# Other Applications of Cumulative

# Rectangle packing

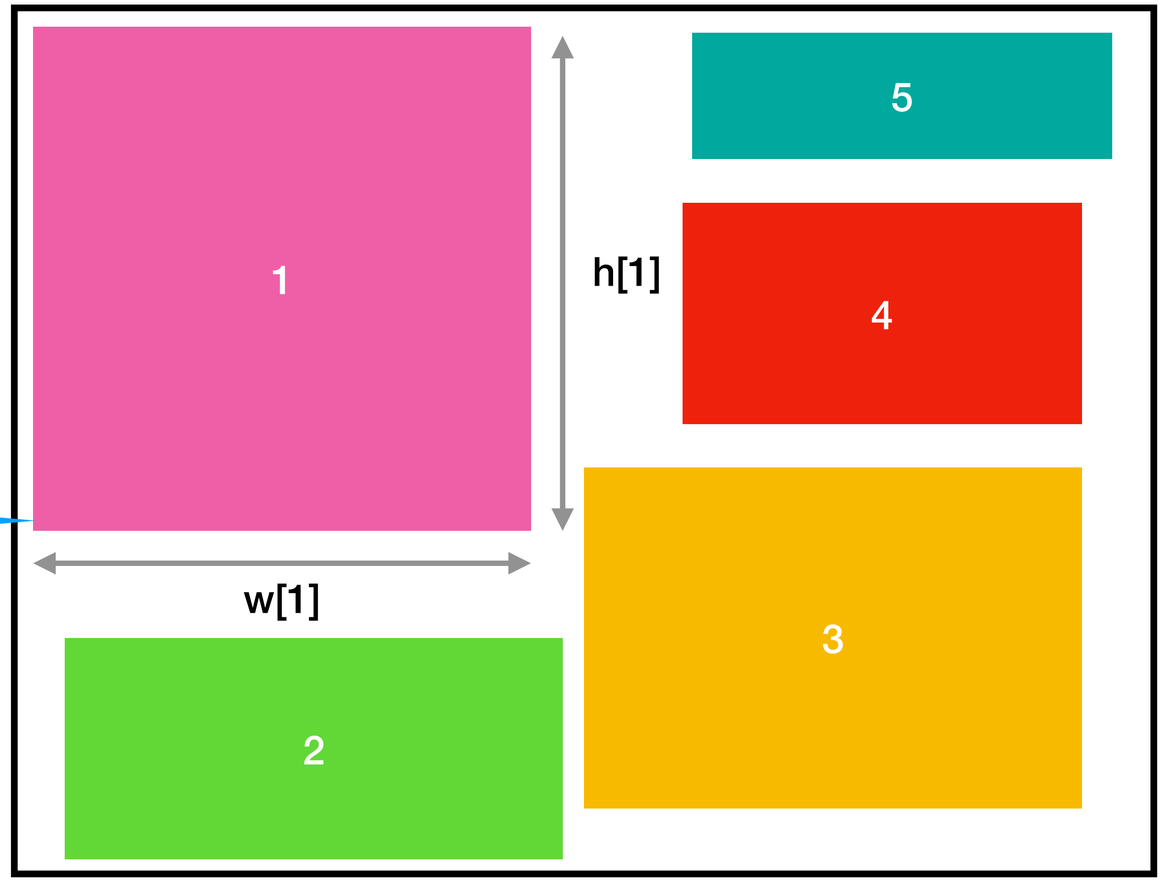
i left of j

j left of i

i below j

j below i

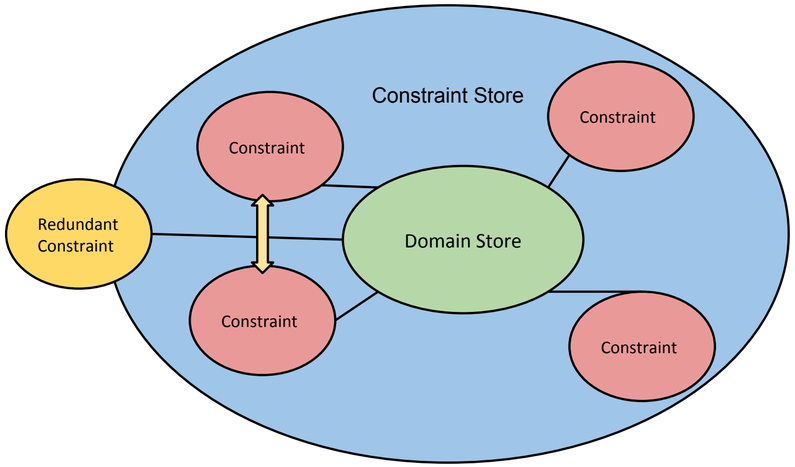
$$\forall i,j : x_i + w_i \leq x_j \vee x_j + w_j \leq x_i \vee y_i + h_i \leq y_j \vee y_j + h_j \leq y_i$$



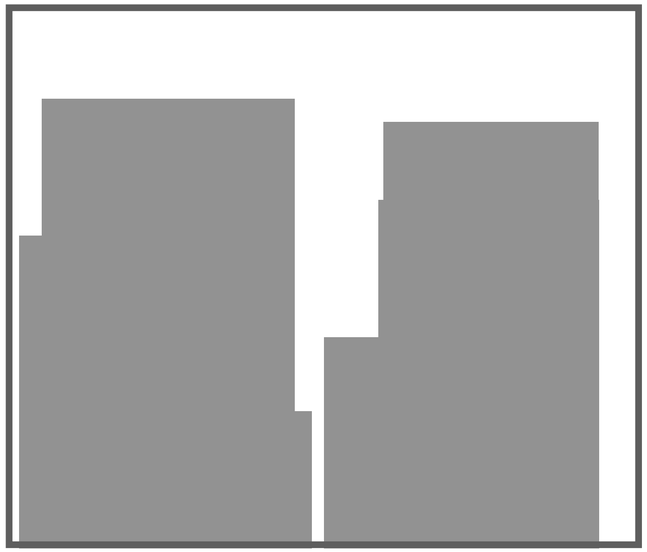

decision variables per rectangle = coordinates of its bottom-left corner: x[1], y[1]

# Redundant (aka Implied) Constraint

- Does not exclude any solution but is nevertheless useful since:
  - It improves the pruning.
  - It helps the communication between the existing constraints.
- Can you find some good redundant constraints for rectangle packing?

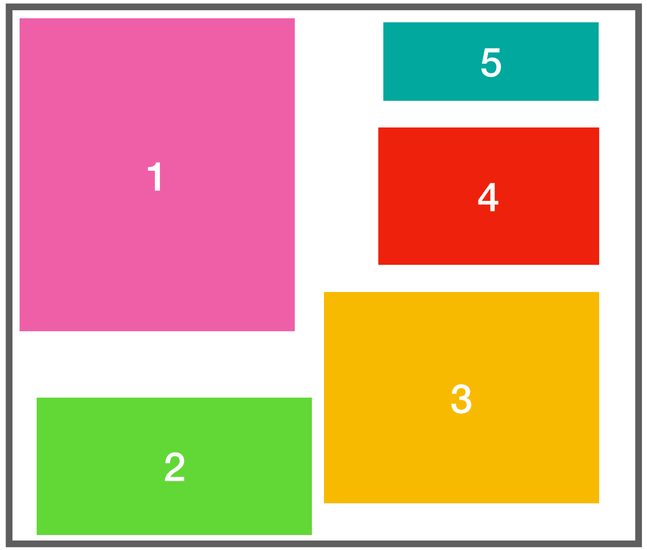


# Rectangle Packing



redundant Cumulative along y axis

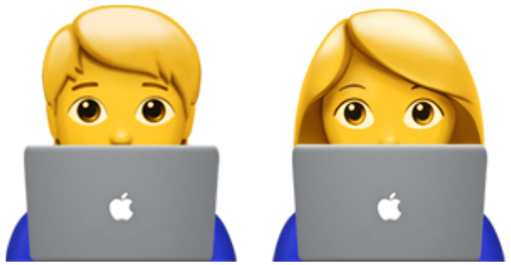
redundant Cumulative along x axis




# Project

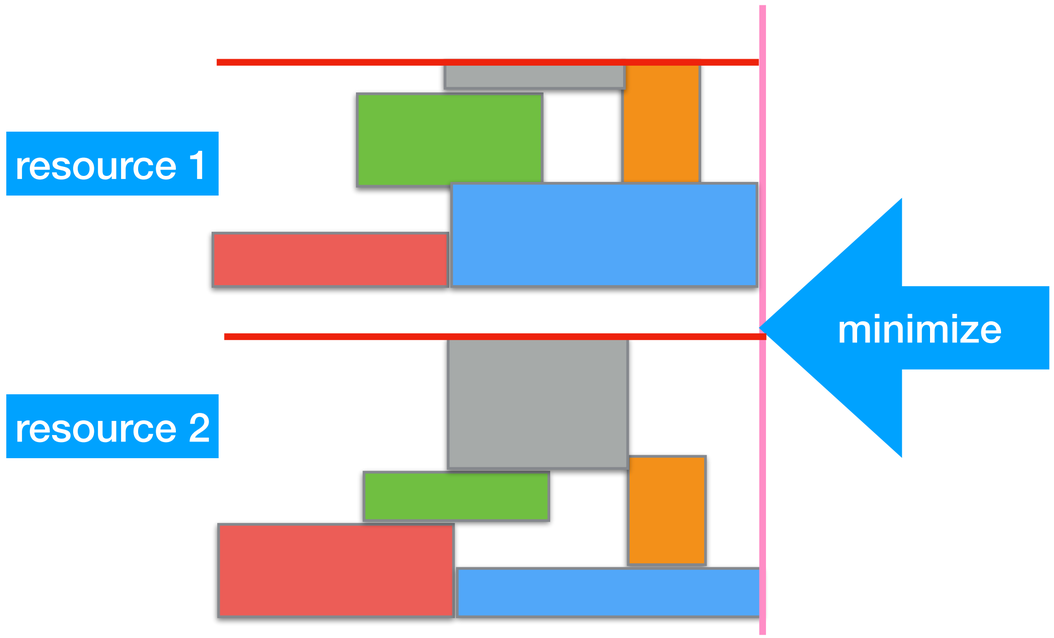
# Implement Cumulative

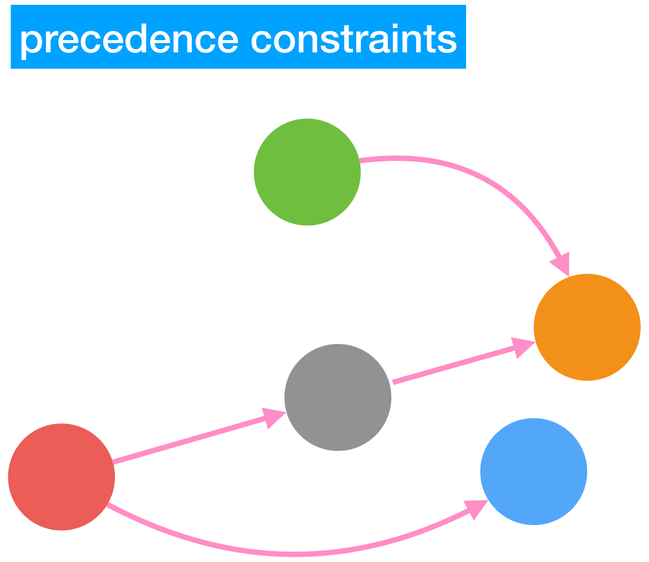
1. Build the cumulated profile.
2. Check that it never exceeds the capacity.
3. Filter the start times.



# Solve the Resource-Constrained Project Scheduling Problem

- Each activity executes on several (cumulative) resources and the requirement on each resource is possibly different.
- Some precedence constraints must hold.
- The objective is to minimize the makespan.



# Other Filtering for Cumulative

# Timetable filtering is not the only filtering

- But timetable filtering is the most scalable filtering and is in practice always used (sometimes with additional, stronger filtering).
- Interested to know more: The best up-to-date literature review is https://school.a4cp.org/summer2017/slidedecks/Resource-Constraints-for-Scheduling.pdf

- Energetic reasoning
- Timetable edge-finding
- Not-first not-last
- …