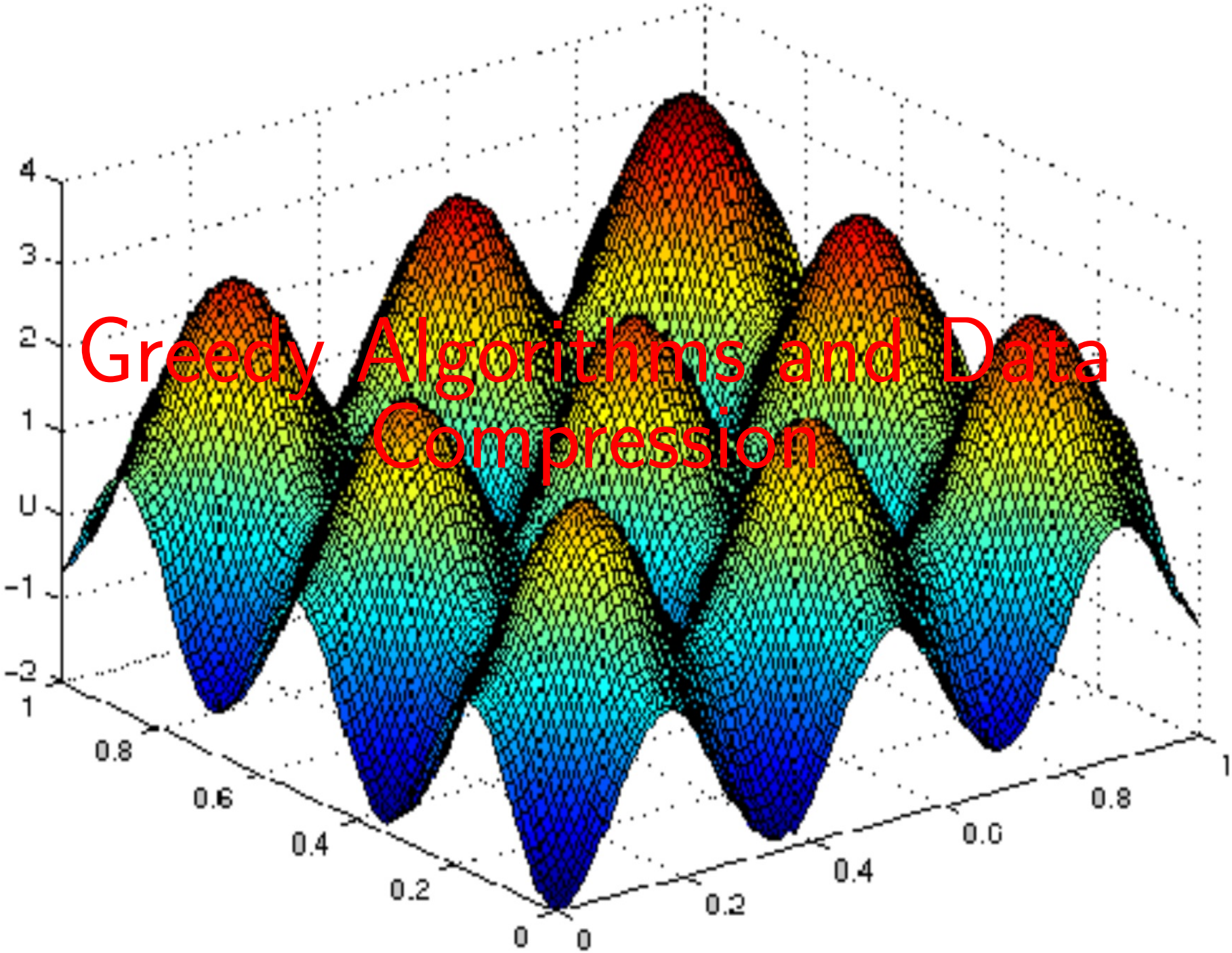
Greedy Algorithms and Data Compression



Task or Activity Scheduling problems

Setting: A set of n tasks (with different characteristics) to be optimally processed by a single (mono) processor (according to different constrains).

1. Interval scheduling problem: Tasks have start and finish times. The objective is to make an executable selection with maximum size.
2. Weighted Interval Scheduling problem: Tasks have start and finish times and its execution produce profits. The objective is to make an executable selection giving maximum profit.
3. Job Scheduling problem (Lateness minimization): Tasks have processing time (could start at any time) and a deadline, define the lateness of a task as the time of its execution that happens after its deadline. Find a executable schedule, including all the tasks, that minimizes the total lateness.

The Interval scheduling problem

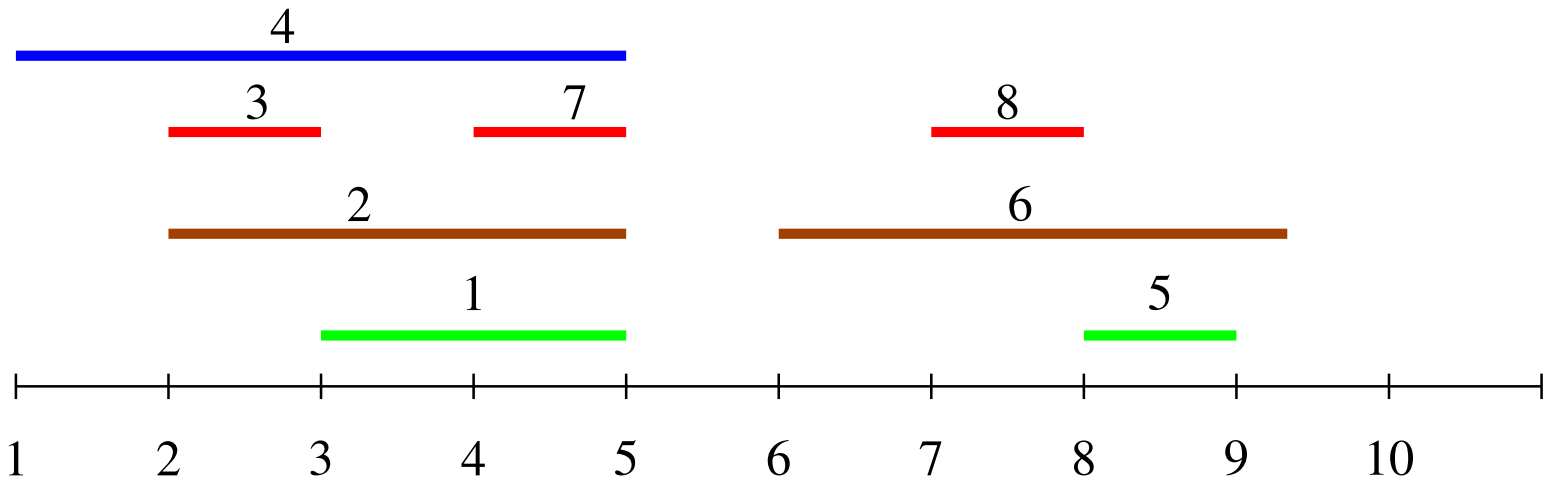
The Interval scheduling (aka Activity Selection problem)
We are given a set of n tasks where, for $i \in [n]$, task i has a start time s_i and a finish time f_i, with $f_i > s_i$.
The tasks have to be processed by a single machine, that can process only one task at a time, the task must be processed from its starting time to its finish time.
We want to find a set of mutually compatible tasks, where activities i and j are compatible if $[s_i f_i) \cap (s_j f_j] = \emptyset$, with maximum size.

We say that two tasks overlap if they are not mutually compatible.

Notice, a solution is a set of mutually compatible activities, and the objective function to maximize is the cardinality of the solution.

Example.

Task :	1	2	3	4	5	6	7	8
Start (s):	3	2	2	1	8	6	4	7
Finish (f):	5	5	3	5	9	9	5	8



To apply the greedy technique to a problem, we must take into consideration the following,

- A local criteria to allow the selection,
- a property to ensure that a partial solution can be completed, to an optimal solution.

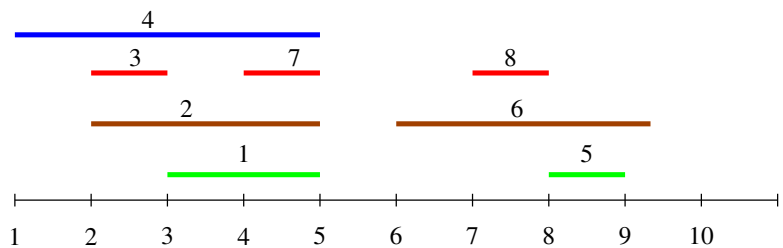
As for the FractionalKnapsack problem the algorithm uses a sorting criteria, and processes the tasks in this order.

The Interval Scheduling problem: algorithm

IntervalScheduling(A)
$S = \emptyset$; $T = [n]$;
while $T \neq \emptyset$ **do**
  Let i be the task that finishes earlier among those in T
  $S = S \cup \{i\}$;
  Remove i and all tasks overlapping i from T
end while
return S.

$task$:	3 4 2 7 8 5 6
s_i :	3 1 2 4 8 5 6
f_i :	3 5 5 5 8 9 9

$\Rightarrow$ SOL: 3 1 8 5



IntervalScheduling: correctness

Theorem

The **IntervalScheduling** *algorithm produces an optimal solution to the* Interval Scheduling *problem.*

Proof. We want to prove that:
There is an optimal solution that includes the task with the earlier finishing time.

- Let i be a task that finishes at the earliest finish time.
- Let S be an optimal solution with $i \notin S$. Let $k \in S$ be the task with the earlier finish time among those in S.
- Any task in S finishes after time $A[k].f$, so they start also after $A[k].f$. As $A[i].f \leq A[k].f$, $S' = (S - \{k\}) \cup \{i\}$ is a set of mutually compatible tasks.
- As $|S'| = |S|$, S' is an optimal solution that includes i.

IntervalScheduling: correctness

Optimal substructure

After each greedy choice, we are left with an optimization subproblem, of the same form as the original. In the subproblem we removed the selected task and all tasks that overlap with the selected one.

An optimal solution to the original problem is formed by the selected task (one that finishes earliest possible) and an optimal solution to the corresponding subproblem.

Interval Scheduling: cost

IntervalScheduling(A)
$S = \emptyset$; $T = [n]$; O(n)
while $T \neq \emptyset$ **do**
  Let i be the task that finishes earlier among those in T O(n)
  $S = S \cup \{i\}$;
  Remove i and all tasks overlapping i from T O(n)
end while
return S.

It takes $O(n^2)$

Interval Scheduling: cost

IntervalScheduling(A)
$S = \emptyset$; $T = [n]$; O(n)
while $T \neq \emptyset$ **do**
  Let i be the task that finishes earlier among those in T O(n)
  $S = S \cup \{i\}$;
  Remove i and all tasks overlapping i from T O(n)
end while
return S.

It takes $O(n^2)$ Too slow, a better implementation?

Interval Scheduling: cost

IntervalScheduling(A)
$S = \emptyset$; $T = [n]$; O(n)
while $T \neq \emptyset$ **do**
  Let i be the task that finishes earlier among those in T O(n)
  $S = S \cup \{i\}$;
  Remove i and all tasks overlapping i from T O(n)
end while
return S.

It takes $O(n^2)$ Too slow, a better implementation?

We have to find a quick selection algorithm and a quick way to discard overlapping tasks.

The Interval Scheduling problem: algorithm 2

IntervalScheduling2(A)
Sort A in increasing order of $A.f$
$S = \{0\}$
$j = 0$ {pointer to last task in solution}
for $i = 1$ **to** $n - 1$ **do**
  if $A[i].s \geq A[j].f$ **then**
    $S = S \cup \{i\}$; $j = i$;
  end if
end for
return S.

IntervalScheduling2: correctness

Theorem

The **IntervalScheduling2** *algorithm produces an optimal solution to the* Interval Scheduling *problem with cost* $O(n \log n)$

Proof.

- A tasks that does not verify the condition in the if statement overlaps with task $j \in S$ and cannot be part of a solution together with j.
- As the tasks are sorted by finish time at each step, we select, among those tasks that start later than j, the one that finishes earlier.
- The algorithm follows the same greedy choice as **IntervalScheduling**, therefore it computes an optimal solution.
- The most costly step is the sorting algorithm that can be done in $O(n \log n)$ using Merge sort.

IntervalScheduling2

IntervalScheduling2

If we know that the tasks start and finish at integer numbers, as those numbers corresponds to seconds, and we know that the time lapsed between the first starting time and the last finish time is less than 24 hours,

IntervalScheduling2 can be implemented with cost

IntervalScheduling2

If we know that the tasks start and finish at integer numbers, as those numbers corresponds to seconds, and we know that the time lapsed between the first starting time and the last finish time is less than 24 hours,

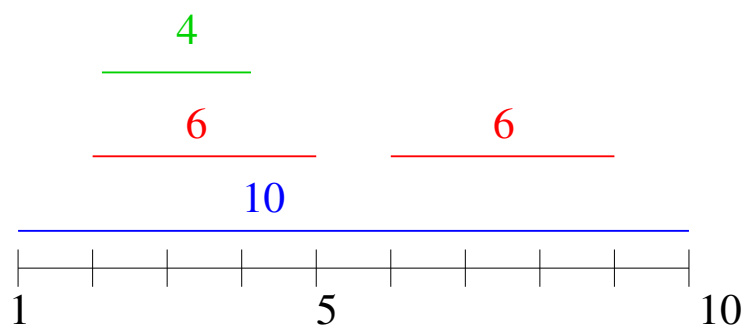
IntervalScheduling2 can be implemented with cost $O(n)$

Adding weights: greedy choice does not always work.

Weighted Activity Selection problem: Given as input a set of $[n]$ activities to be processed by a single resource, where each activity i has a start time s_i and a finish time f_i, with $f_i > s_i$, and a weight w_i. We want to find a set S of mutually compatible activities so that $\sum_{i \in S} w_i$ is maximum among all such sets.

Adding weights: greedy choice does not always work.

Weighted Activity Selection problem: Given as input a set of $[n]$ activities to be processed by a single resource, where each activity i has a start time s_i and a finish time f_i, with $f_i > s_i$, and a weight w_i. We want to find a set S of mutually compatible activities so that $\sum_{i \in S} w_i$ is maximum among all such sets.



IntervalScheduling2 will select the green and the second red activity with weight 10.

What about maximizing locally the selected weight?

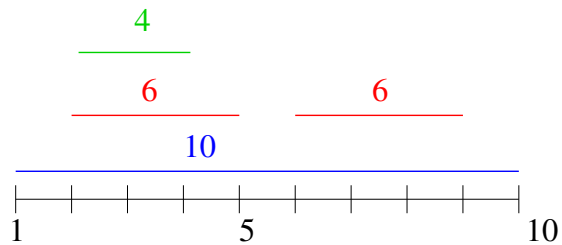
WeightedAS-max-weight (A)
$S = \emptyset$; $T = [n]$;
while $T \neq \emptyset$ **do**
  Let i be the task with highest weight among those in T.
  $S = S \cup \{i\}$
  Remove i and all tasks overlapping i from T
end while
return S

Correctness?

What about maximizing locally the selected weight?

WeightedAS-max-weight (A)
$S = \emptyset$; $T = [n]$;
while $T \neq \emptyset$ **do**
  Let i be the task with highest weight among those in T.
  $S = S \cup \{i\}$
  Remove i and all tasks overlapping i from T
end while
return S

Correctness?



The algorithm chooses the blue task with weight 10, and the optimal solution is formed by the two red intervals with total weight of 12

Job Scheduling problem

Lateness Minimization problem.

We have a single resource (processor) and n requests to use the resource, each request i taking a time t_i. Once a request starts to be served it continues using the resource until its completion.

Each request has a deadline d_i. The goal is to schedule all the request, determine the time at which each request starts having exclusive access to the resource, that minimizes, over all the requests, the maximum amount of time that a request exceeds its deadline.

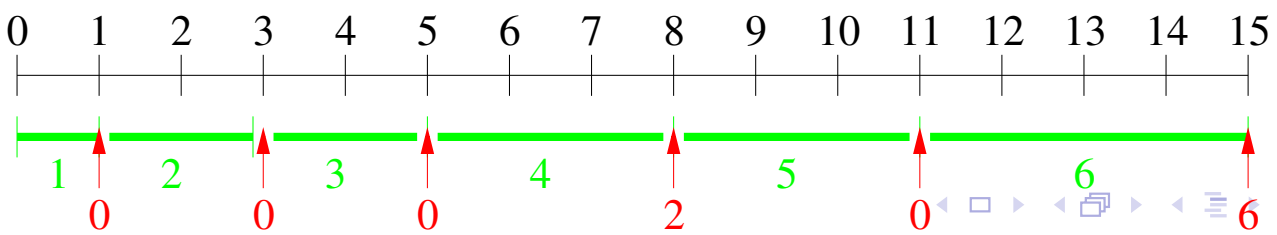
Minimize Lateness: jobs-processor view

- We have a single processor
- We have n jobs such that job i:
  - requires $t_i > 0$ units of processing time,
  - it has to be finished by time d_i,
- Lateness of i:

$$L_i = \begin{cases} 0 & \text{if } f_i \leq d_i, \\ f_i - d_i & \text{otherwise.} \end{cases}$$

i	t_i	d_i
1	1	9
2	2	8
3	2	15
4	3	6
5	3	14
6	4	9

Goal: schedule the jobs to minimize the maximum lateness, over all the jobs. The function to optimize is $\max L_i$.
We must assign starting time s_i to each i, making sure that the processor only processes a job at a time, in such a way that $\max_i L_i$ is minimum.



Minimize Lateness

We can try different job selection criteria. This will give the sorting criteria for the algorithm, the schedule will follow this order assigning the resource to the next request as soon as possible.

LatenessXX (A)
Sort A according to XX
$S[0] = 0$; $t = A[0].t$; $L = \max(0, t - A[0].d)$;
for $i = 1$ **to** $n - 1$ **do**
  $S[i] = t$
  $t = t + A[i].t$
  $L = \max(L, \max(0, t - A[i].d))$
end for
return (S, L)

Minimize Lateness: selection criteria

Process jobs with short time first

i	t_i	d_i
1	1	6
2	5	5

1 at time 0 and 2 at time 1 lateness 1, but
2 at time 0 and 1 at time 5 has lateness 0.
It does not work.

Process first jobs with smaller $d_i - t_i$ time

i	t_i	d_i	$d_1 - t_i$
1	1	2	1
2	10	10	0

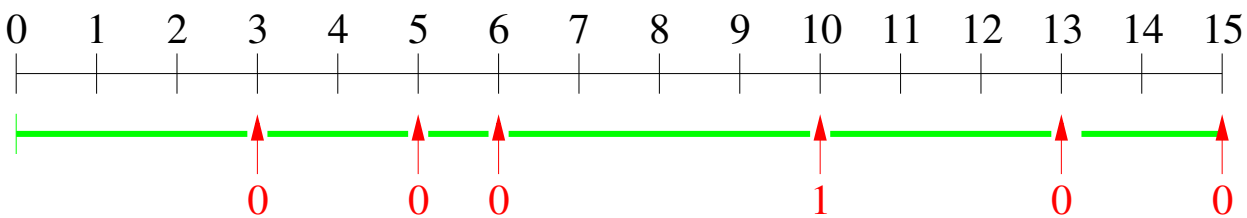
In this case, job 2 should be processed first, which doesn't minimise lateness.

Process urgent jobs first

Sort in increasing order of d_i.

LatenessUrgent (A)
Sort A by increasing order of $A.d$
$S[0] = 0$; $t = A[0].t$;
$L = \max(0, t - A[0].d)$;
for $i = 1$ **to** $n - 1$ **do**
  $S[i] = t$
  $t = t + A[i].t$
  $L = \max(L, \max(0, t - A[i].d))$
end for
return (S, L)

i	t_i	d_i	sorted i
1	1	9	3
2	2	8	2
3	2	15	6
4	3	6	1
5	3	14	5
6	4	9	4



d: 6 8 9 9 14 15
i: 1 2 3 4 5 6

Process urgent jobs first: Complexity

LatenessUrgent (A)
Sort A by increasing order of $A.d$
$S[0] = 0$; $t = A[0].t$; $L = \max(0, t - A[0].d)$;
for $i = 1$ **to** $n - 1$ **do**
  $S[i] = t$
  $t = t + A[i].t$
  $L = \max(L, \max(0, t - A[i].d))$
end for
return (S, L)

Time complexity

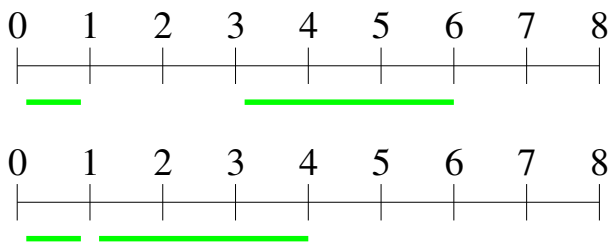
Running-time of the algorithm without sorting $O(n)$

Total running-time: $O(n \lg n)$

Process urgent jobs first: Correctness

There is an optimal schedule minimizing lateness that does not have idle steps

From a schedule with idle steps, we always can eliminate gaps to obtain another schedule with the same or better lateness:



LatenessUrgent has no idle steps.

Inversions and exchange argument

A schedule S has an inversion if i is scheduled before j and $d_j < d_i$.

Lemma
Exchanging two adjacent inverted jobs reduces the number of inversions by 1 and does not increase the max lateness.

Proof Assume that in schedule S, i is scheduled just before j and that they form an inversion.
Let S' be the schedule obtained from S interchanging i with j.

- S and S' coincide in all the jobs scheduled before i or after j.
- Only i and j can change lateness.
- Let L_i, L_j and L'_i, L'_j be the lateness of jobs i and j in S and S', respectively.
- Let f_i, f_j and f'_i, f'_j be the finish times of jobs i and j in S and S', respectively.

Inversions and exchange argument: Cont.

- We have $f_i < f_j$, $f'_j < f'_i$, $f'_i = f_j$, and $f'_j < f_j$. Also $d_j < d_i$,
- If $f_j < d_j$, $L_i = L_j = L'_i = L'_j = 0$
- If $d_i < f_i$,

$$L'_i = f'_i - d_i = f_j - d_i < f_j - d_j = L_i$$
$$L'_j = f'_j - d_j < f_j - d_j = L_j$$

- if $f_i \leq d_i < d_j \leq f_j$, $f'_j \leq d_i < d_j \leq f'_i = f_j$

$$L'_i = 0 \leq L_j$$
$$L'_j = f'_j - d_j < f_j - d_j = L_j$$

Therefore, the swapping does not increase the maximum lateness of the schedule.

Correctness of **LatenessUrgent**

Theorem

Algorithm **LatenessUrgent** *solves correctly the* Lateness Minimization *problem.*

Proof.

The S produced by **LatenessUrgent** has no inversions and no idle steps.

Assume $\hat{S}$ is an optimal schedule. We can assume that it has no idle steps.

- If $\hat{S}$ has 0 inversions then $\hat{S} = S$.
- Otherwise, $\hat{S}$ has an inversion on two adjacent jobs. Let i, j be an adjacent inversion.
  As we have seen, exchanging i and j does not increase lateness but it decreases the number of inversions.
  As $\hat{S}$ is optimal, the new schedule is also optimal but has one inversion less.
- Repeating, if needed the interchange of adjacent inversions, we will reach an optimal schedule with no inversions.
  Therefore, S is optimal.