Greedy Algorithms

Tao Hou

Outline

- Introduction
- Problems
  - Fractional Knapsack
  - Interval Scheduling
  - Interval Partitioning

- Algorithms for ***optimization*** problems typically go through a sequence of steps, with a set of choices at each step.
- A ***greedy algorithm*** is a very special type of algorithms for solving optimization problems in the sense that it always makes the choice that ***looks best at the moment***.
- That is, it makes a ***locally optimal choice*** at each step hoping that this will lead to a ***globally optimal solution***.

Greedy Algorithms: Introduction

- Algorithms for ***optimization*** problems typically go through a sequence of steps, with a set of choices at each step.
- A ***greedy algorithm*** is a very special type of algorithms for solving optimization problems in the sense that it always makes the choice that ***looks best at the moment***.
- That is, it makes a ***locally optimal choice*** at each step hoping that this will lead to a ***globally optimal solution***.
- A related technique for solving optimization problem but in dark contrast is ***dynamic programming*** (the next topic of this course), in which we typically enumerate all local/incremental choices at each step and select the best.
- However, for some optimization problems, dynamic programming is overkill: greedy algorithm can provide a simpler, more efficient solution.

Greedy Algorithms: Introduction

- Algorithms for ***optimization*** problems typically go through a sequence of steps, with a set of choices at each step.
- A ***greedy algorithm*** is a very special type of algorithms for solving optimization problems in the sense that it always makes the choice that ***looks best at the moment***.
- That is, it makes a ***locally optimal choice*** at each step hoping that this will lead to a ***globally optimal solution***.
- A related technique for solving optimization problem but in dark contrast is ***dynamic programming*** (the next topic of this course), in which we typically enumerate all local/incremental choices at each step and select the best.
- However, for some optimization problems, dynamic programming is overkill: greedy algorithm can provide a simpler, more efficient solution.
- Caution that a bunch of locally optimal choices usually ***do not*** lead to globally optimal choice: this is true ***only for certain problems***, and this need ***proofs***!

A further remark:

- In order for greedy algorithm to work, a problem typically should satisfy the ***optimal-substructure property***, i.e., we should be able to easily combine optimal solutions to subproblems to produce the optimal solution to the original problem
  - We will address this in more detail in the dynamic-programming section.

A further remark:

- In order for greedy algorithm to work, a problem typically should satisfy the ***optimal-substructure property***, i.e., we should be able to easily combine optimal solutions to subproblems to produce the optimal solution to the original problem
  - We will address this in more detail in the dynamic-programming section.

Characteristics of greedy algorithms:

- *Describing* a greedy algorithm is *easy*
- *Coming up* with an algorithm is *tricky*
  - wouldn't think that such simple strategy can actually work
  - don't actually know which (local) criterion to optimize on: a ***design choice*** you have to make
- *Proving* that the algorithm is correct is usually *hard*
  - requires deep understanding of the ***structure of the problem***
  - ***We will delve into a lot of proofs in this topic!***

- *Gift-selection problem*
  - out of a set $X = \{x_1, x_2, \ldots, x_n\}$ of valuable objects, where $v(x_i)$ is the value of x_i
  - you will be given, as a gift, k objects of your choice
  - how do you maximize the total value of your gifts?

- *Algorithm:* Sort the gifts by their values starting from the most valuable one, and choose the first k gifts
  - This is a greedy algorithm and it's easy to believe that it's correct

- The algorithms we shall study later are not so easy to see the correctness

***Fractional* Knapsack Problem**

Problem: Given n items and a "knapsack" with a capacity W s.t.

- Each item i has w_i units of weight and a profit v_i ($w_i, v_i > 0$)
- For each item, you can take ***any fraction*** of weight for that item and gain corresponding profits
- E.g., for an item with a weight 5 and a profit 6, you can take 2.2 units of the item gaining a profit of $2.2 * \frac{6}{5}$, which occupies 2.2 units of weight in the knapsack
  - $\frac{6}{5}$ is the ***unit profit*** for the item

Goal: Find a way to put the fractions of the items into the knapsack (i.e., total fractional weights of items is less than capacity) so that you gain the most profit

Fractional Knapsack: Solution

Idea:

- Decreasingly sort the items by their ***unit profits*** (v_i/w_i)
- Go over each item i in the above order, and put ***as many*** item i ***as you can*** into the knapsack, until the knapsack is full

Fractional Knapsack: Solution

Idea:

- Decreasingly sort the items by their ***unit profits*** (v_i/w_i)
- Go over each item i in the above order, and put ***as many*** item i ***as you can*** into the knapsack, until the knapsack is full

FracKnapsack($\{w_1, \ldots, w_n\}, \{v_1, \ldots, v_n\}, W$)

sort and renumber the items s.t.
    $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
$R = W$ // 'remaining' capacity
for $i = 1, \ldots, n$:
    if $R > w_i$
        put w_i units of item i into the knapsack
        $R = R - w_i$
    else
        put R units of item i into the knapsack
        break

Time complexity: $O(n \log n)$

Fractional Knapsack: Justification

- Is the previous algorithm correct? And if it is, how to show that the generated solution is optimal?

Fractional Knapsack: Justification

1. Assume the numbering of the objects satisfy $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$

Fractional Knapsack: Justification

1. Assume the numbering of the objects satisfy $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
2. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the ***greedy solution***, where p_i is the number of units of item i put into the knapsack

Fractional Knapsack: Justification

1. Assume the numbering of the objects satisfy $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
2. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the ***greedy solution***, where p_i is the number of units of item i put into the knapsack
3. Let $Q = \{q_1, q_2, \ldots, q_n\}$ be an ***optimal solution***, where q_i is similarly defined as previous

Fractional Knapsack: Justification

1. Assume the numbering of the objects satisfy $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
2. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the ***greedy solution***, where p_i is the number of units of item i put into the knapsack
3. Let $Q = \{q_1, q_2, \ldots, q_n\}$ be an ***optimal solution***, where q_i is similarly defined as previous
4. Let i be ***first*** index s.t. $p_i \neq q_i$

Fractional Knapsack: Justification

1. Assume the numbering of the objects satisfy $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
2. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the ***greedy solution***, where p_i is the number of units of item i put into the knapsack
3. Let $Q = \{q_1, q_2, \ldots, q_n\}$ be an ***optimal solution***, where q_i is similarly defined as previous
4. Let i be ***first*** index s.t. $p_i \neq q_i$
5. We must have $p_i > q_i$ because in the greedy solution P, we take "as many as we can" for each item, therefore Q cannot take more unit of item i than P

Fractional Knapsack: Justification

1. Assume the numbering of the objects satisfy $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
2. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the ***greedy solution***, where p_i is the number of units of item i put into the knapsack
3. Let $Q = \{q_1, q_2, \ldots, q_n\}$ be an ***optimal solution***, where q_i is similarly defined as previous
4. Let i be ***first*** index s.t. $p_i \neq q_i$
5. We must have $p_i > q_i$ because in the greedy solution P, we take "as many as we can" for each item, therefore Q cannot take more unit of item i than P
6. ***Modify*** solution Q as follows: take out $p_i - q_i$ units of any items after i in Q, and 'replace' them with $p_i - q_i$ units of item i.

Fractional Knapsack: Justification

1. Assume the numbering of the objects satisfy $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
2. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the ***greedy solution***, where p_i is the number of units of item i put into the knapsack
3. Let $Q = \{q_1, q_2, \ldots, q_n\}$ be an ***optimal solution***, where q_i is similarly defined as previous
4. Let i be ***first*** index s.t. $p_i \neq q_i$
5. We must have $p_i > q_i$ because in the greedy solution P, we take "as many as we can" for each item, therefore Q cannot take more unit of item i than P
6. ***Modify*** solution Q as follows: take out $p_i - q_i$ units of any items after i in Q, and 'replace' them with $p_i - q_i$ units of item i.
7. This produces another solution whose value is no smaller than the original Q, because we are swapping in items whose unit values are no smaller.

***Fractional* Knapsack: Justification**

1. Assume the numbering of the objects satisfy $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
2. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the ***greedy solution***, where p_i is the number of units of item i put into the knapsack
3. Let $Q = \{q_1, q_2, \ldots, q_n\}$ be an ***optimal solution***, where q_i is similarly defined as previous
4. Let i be ***first*** index s.t. $p_i \neq q_i$
5. We must have $p_i > q_i$ because in the greedy solution P, we take "as many as we can" for each item, therefore Q cannot take more unit of item i than P
6. ***Modify*** solution Q as follows: take out $p_i - q_i$ units of any items after i in Q, and 'replace' them with $p_i - q_i$ units of item i.
7. This produces another solution whose value is no smaller than the original Q, because we are swapping in items whose unit values are no smaller.
8. So this "new" solution Q is also an optimal solution, which we also name it as Q. But this time we have that $p_i = q_i$.

Fractional Knapsack: Justification

1. Assume the numbering of the objects satisfy $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
2. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the ***greedy solution***, where p_i is the number of units of item i put into the knapsack
3. Let $Q = \{q_1, q_2, \ldots, q_n\}$ be an ***optimal solution***, where q_i is similarly defined as previous
4. Let i be ***first*** index s.t. $p_i \neq q_i$
5. We must have $p_i > q_i$ because in the greedy solution P, we take "as many as we can" for each item, therefore Q cannot take more unit of item i than P
6. ***Modify*** solution Q as follows: take out $p_i - q_i$ units of any items after i in Q, and 'replace' them with $p_i - q_i$ units of item i.
7. This produces another solution whose value is no smaller than the original Q, because we are swapping in items whose unit values are no smaller.
8. So this "new" solution Q is also an optimal solution, which we also name it as Q. But this time we have that $p_i = q_i$.
9. With this new optimal Q, if we find the first index s.t. $p_i \neq q_i$ as in Step 4, such a "first index" is going to increase

Fractional **Knapsack: Justification**

1. Assume the numbering of the objects satisfy $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
2. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the ***greedy solution***, where p_i is the number of units of item i put into the knapsack
3. Let $Q = \{q_1, q_2, \ldots, q_n\}$ be an ***optimal solution***, where q_i is similarly defined as previous
4. Let i be ***first*** index s.t. $p_i \neq q_i$
5. We must have $p_i > q_i$ because in the greedy solution P, we take "as many as we can" for each item, therefore Q cannot take more unit of item i than P
6. ***Modify*** solution Q as follows: take out $p_i - q_i$ units of any items after i in Q, and 'replace' them with $p_i - q_i$ units of item i.
7. This produces another solution whose value is no smaller than the original Q, because we are swapping in items whose unit values are no smaller.
8. So this "new" solution Q is also an optimal solution, which we also name it as Q. But this time we have that $p_i = q_i$.
9. With this new optimal Q, if we find the first index s.t. $p_i \neq q_i$ as in Step 4, such a "first index" is going to increase
10. If we repeatedly perform Step 4-6, the first index such that P and Q differ will keep on increasing, until $P = Q$. So P is optimal

- A conference room is shared among different activities
  - $S = \{1, 2, \ldots, n\}$ is the set of proposed activities
  - activity *i* has a *start time* s_i and a *finish time* f_i
  - activities *i* and *j* are *compatible* if either $f_i \le s_j$ or $f_j \le s_i$ (i.e., their time intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap)

- A conference room is shared among different activities
  - $S = \{1, 2, \ldots, n\}$ is the set of proposed activities
  - activity i has a *start time* s_i and a *finish time* f_i
  - activities i and j are *compatible* if either $f_i \leq s_j$ or $f_j \leq s_i$ (i.e., their time intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap)

Problem: find the largest subset of compatible activities

- A conference room is shared among different activities
  - $S = \{1, 2, \ldots, n\}$ is the set of proposed activities
  - activity i has a *start time* s_i and a *finish time* f_i
  - activities i and j are *compatible* if either $f_i \leq s_j$ or $f_j \leq s_i$ (i.e., their time intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap)

Problem: find the largest subset of compatible activities

- Example

activity	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*
start	8	0	2	3	5	1	5	3	12	6	8
finish	12	6	13	5	7	4	9	8	14	10	11

Interval Scheduling Problem

The previous problem can be also formalized as an ***interval scheduling*** problem

- Given a set of n intervals: $[s_1, f_1), [s_2, f_2), \ldots, [s_n, f_n)$
- Find the largest subset of *dis-joint* intervals

Interval Scheduling: Naive Solutions

- The most naive method is to *enumerate each subset* of the intervals and check the compatibility, which is in exponential time
- There also exists a *dynamic-programming* algorithm for the problem
- But we will look at a ***greedy algorithm*** which is much *simpler* and *faster*

Interval Scheduling: Greedy Solution

Idea:

- Order the intervals by their ***finishing time***.
- Go over each interval in the order, select the interval if it is compatible with the ones already selected

Interval Scheduling: Greedy Solution

Idea:

- Order the intervals by their ***finishing time***.
- Go over each interval in the order, select the interval if it is compatible with the ones already selected

GreedyIntervSched($\{s_1, \ldots, s_n\}, \{f_1, \ldots, f_n\}$)

```
sort and renumber the intervals s.t.
    f_1 ≤ f_2 ≤ ··· ≤ f_n
C = ∅ // selected intervals
for i = 1, ..., n:
    if interval i is compatible with intervals in C
        C = C ∪ {i}
return C
```

Example

Implementing the algorithm

How do we efficiently implement the algorithm?

```
GreedyIntervSched({s_1, ..., s_n}, {f_1, ..., f_n})
sort and rename the intervals s.t.
    f_1 ≤ f_2 ≤ ··· ≤ f_n
C = ∅ // selected intervals
for i = 1, ..., n:
    if interval i is compatible with intervals in C
        C = C ∪ {i}
return C
```

Implementing the algorithm

- In the previous algorithm, in each step, we need to check whether an interval i is compatible with intervals in C.

- In the previous algorithm, in each step, we need to check whether an interval i is compatible with intervals in C.
- Let $C = \{a_1, a_2, \ldots, a_j\}$.

Implementing the algorithm

- In the previous algorithm, in each step, we need to check whether an interval i is compatible with intervals in C.
- Let $C = \{a_1, a_2, \ldots, a_j\}$.
- Since intervals in C are compatible with each other, we can assume:

$$[s_{a_1}, f_{a_1}) \leq [s_{a_2}, f_{a_2}) \leq \cdots \leq [s_{a_j}, f_{a_j})$$

Implementing the algorithm

- In the previous algorithm, in each step, we need to check whether an interval i is compatible with intervals in C.
- Let $C = \{a_1, a_2, \ldots, a_j\}$.
- Since intervals in C are compatible with each other, we can assume:

$$[s_{a_1}, f_{a_1}) \leq [s_{a_2}, f_{a_2}) \leq \cdots \leq [s_{a_j}, f_{a_j})$$

- Since we ordered the intervals by finishing time, we have $f_i \geq f_{a_j}$

Implementing the algorithm

- In the previous algorithm, in each step, we need to check whether an interval i is compatible with intervals in C.
- Let $C = \{a_1, a_2, \dots, a_j\}$.
- Since intervals in C are compatible with each other, we can assume:

$$[s_{a_1}, f_{a_1}) \leq [s_{a_2}, f_{a_2}) \leq \cdots \leq [s_{a_j}, f_{a_j})$$

- Since we ordered the intervals by finishing time, we have $f_i \geq f_{a_j}$
- Then interval i is compatible with all intervals in C if and only if it is compatible with the last interval a_j

Implementing the algorithm

- In the previous algorithm, in each step, we need to check whether an interval i is compatible with intervals in C.
- Let $C = \{a_1, a_2, \ldots, a_j\}$.
- Since intervals in C are compatible with each other, we can assume:

$$[s_{a_1}, f_{a_1}) \leq [s_{a_2}, f_{a_2}) \leq \cdots \leq [s_{a_j}, f_{a_j})$$

- Since we ordered the intervals by finishing time, we have $f_i \geq f_{a_j}$
- Then interval i is compatible with all intervals in C if and only if it is compatible with the last interval a_j
- So we only need check whether $s_i \geq f_{a_j}$

Implementing the algorithm

- In the previous algorithm, in each step, we need to check whether an interval i is compatible with intervals in C.
- Let $C = \{a_1, a_2, \ldots, a_j\}$.
- Since intervals in C are compatible with each other, we can assume:

$$[s_{a_1}, f_{a_1}) \leq [s_{a_2}, f_{a_2}) \leq \cdots \leq [s_{a_j}, f_{a_j})$$

- Since we ordered the intervals by finishing time, we have $f_i \geq f_{a_j}$
- Then interval i is compatible with all intervals in C if and only if it is compatible with the last interval a_j
- So we only need check whether $s_i \geq f_{a_j}$
- Therefore, in the algorithm, we will have a variable F keeping the finishing time of the last interval in C, and at each iteration we check whether the starting time of interval i is later than F

More detailed pseudocodes

GreedyIntervSched$(\{s_1, \ldots, s_n\}, \{f_1, \ldots, f_n\})$

sort and rename the intervals s.t.
    $f_1 \leq f_2 \leq \cdots \leq f_n$
$C = \varnothing$ // selected intervals
$F = -\infty$ // finishing time of the last interval in C
for $i = 1, \ldots, n$:
    if $s_i \geq F$
        $C = C \cup \{i\}$
        $F = f_i$
return C

Time complexity: $O(n \log n)$

More detailed pseudocodes

GreedyIntervSched$(\{s_1, \ldots, s_n\}, \{f_1, \ldots, f_n\})$

sort and rename the intervals s.t.
    $f_1 \leq f_2 \leq \cdots \leq f_n$
$C = \varnothing$ // selected intervals
$F = -\infty$ // finishing time of the last interval in C
for $i = 1, \ldots, n$:
    if $s_i \geq F$
        $C = C \cup \{i\}$
        $F = f_i$
return C

Time complexity: $O(n \log n)$

Question: Is the above greedy algorithm correct? How do we prove it always produce the optimal solution?

Interval Scheduling: Justification

We first show that at each step of the greedy algorithm, the set of selected intervals C is ***always contained*** in an optimal solution. This is shown ***inductively*** based on the following proposition:

Interval Scheduling: Justification

We first show that at each step of the greedy algorithm, the set of selected intervals C is ***always contained*** in an optimal solution. This is shown ***inductively*** based on the following proposition:

Proposition

- In the greedy algorithm, suppose at a certain step i, we add an interval i into C.
- If before adding i, C is ***contained in*** an optimal solution, then after adding i to C, C is ***also contained in*** an optimal solution.

Interval Scheduling: Justification

We first show that at each step of the greedy algorithm, the set of selected intervals C is ***always contained*** in an optimal solution. This is shown ***inductively*** based on the following proposition:

Proposition

- In the greedy algorithm, suppose at a certain step i, we add an interval i into C.
- If before adding i, C is ***contained in*** an optimal solution, then after adding i to C, C is ***also contained in*** an optimal solution.

What this proposition ***implies***:

- We have that initially, $C = \varnothing$ is contained in an optimal solution.
- So by induction, ***at each step of the algorithm, after adding an interval into C, C is contained in an optimal solution***, due to the proposition
- Specifically, the ***final solution*** returned by the greed algorithm is contained in an optimal solution

- Suppose before adding i to C, $C = \{a_1, a_2, \ldots, a_j\}$

Proof of the Proposition

- Suppose before adding i to C, $C = \{a_1, a_2, \ldots, a_j\}$
- By the assumption, we have that $C = \{a_1, a_2, \ldots, a_j\}$ is contained in an optimal solution, and we want to show that $\{a_1, a_2, \ldots, a_j, i\}$ is contained in an optimal solution.

Proof of the Proposition

- Suppose before adding i to C, $C = \{a_1, a_2, \ldots, a_j\}$
- By the assumption, we have that $C = \{a_1, a_2, \ldots, a_j\}$ is contained in an optimal solution, and we want to show that $\{a_1, a_2, \ldots, a_j, i\}$ is contained in an optimal solution.
- Let $O = \{a_1, a_2, \ldots, a_j, b_{j+1}, b_{j+2}, \ldots, b_\ell\}$ be an optimal solution containing C

- Suppose before adding i to C, $C = \{a_1, a_2, \dots, a_j\}$
- By the assumption, we have that $C = \{a_1, a_2, \dots, a_j\}$ is contained in an optimal solution, and we want to show that $\{a_1, a_2, \dots, a_j, i\}$ is contained in an optimal solution.
- Let $O = \{a_1, a_2, \dots, a_j, b_{j+1}, b_{j+2}, \dots, b_\ell\}$ be an optimal solution containing C
- Since intervals in O are compatible, we can assume they are ordered:

$$[s_{a_1}, f_{a_1}) \le [s_{a_2}, f_{a_2}) \le \cdots \le [s_{a_j}, f_{a_j}) \le [s_{b_{j+1}}, f_{b_{j+1}}) \le [s_{b_{j+2}}, f_{b_{j+2}}) \le \cdots \le [s_{b_\ell}, f_{b_\ell})$$

- Suppose before adding i to C, $C = \{a_1, a_2, \ldots, a_j\}$
- By the assumption, we have that $C = \{a_1, a_2, \ldots, a_j\}$ is contained in an optimal solution, and we want to show that $\{a_1, a_2, \ldots, a_j, i\}$ is contained in an optimal solution.
- Let $O = \{a_1, a_2, \ldots, a_j, b_{j+1}, b_{j+2}, \ldots, b_\ell\}$ be an optimal solution containing C
- Since intervals in O are compatible, we can assume they are ordered:

$$[s_{a_1}, f_{a_1}) \le [s_{a_2}, f_{a_2}) \le \cdots \le [s_{a_j}, f_{a_j}) \le [s_{b_{j+1}}, f_{b_{j+1}}) \le [s_{b_{j+2}}, f_{b_{j+2}}) \le \cdots \le [s_{b_\ell}, f_{b_\ell})$$

- Since b_{j+1} is compatible with a_j, we must have $f_{b_{j+1}} \ge f_i$,

- Suppose before adding i to C, $C = \{a_1, a_2, \ldots, a_j\}$
- By the assumption, we have that $C = \{a_1, a_2, \ldots, a_j\}$ is contained in an optimal solution, and we want to show that $\{a_1, a_2, \ldots, a_j, i\}$ is contained in an optimal solution.
- Let $O = \{a_1, a_2, \ldots, a_j, b_{j+1}, b_{j+2}, \ldots, b_\ell\}$ be an optimal solution containing C
- Since intervals in O are compatible, we can assume they are ordered:

$$[s_{a_1}, f_{a_1}) \leq [s_{a_2}, f_{a_2}) \leq \cdots \leq [s_{a_j}, f_{a_j}) \leq [s_{b_{j+1}}, f_{b_{j+1}}) \leq [s_{b_{j+2}}, f_{b_{j+2}}) \leq \cdots \leq [s_{b_\ell}, f_{b_\ell})$$

- Since b_{j+1} is compatible with a_j, we must have $f_{b_{j+1}} \geq f_i$,
  - If $f_{b_{j+1}} < f_i$, then b_{j+1} would have been processed before i in the algorithm;

Proof of the Proposition

- Suppose before adding i to C, $C = \{a_1, a_2, \ldots, a_j\}$
- By the assumption, we have that $C = \{a_1, a_2, \ldots, a_j\}$ is contained in an optimal solution, and we want to show that $\{a_1, a_2, \ldots, a_j, i\}$ is contained in an optimal solution.
- Let $O = \{a_1, a_2, \ldots, a_j, b_{j+1}, b_{j+2}, \ldots, b_\ell\}$ be an optimal solution containing C
- Since intervals in O are compatible, we can assume they are ordered:

$$[s_{a_1}, f_{a_1}) \leq [s_{a_2}, f_{a_2}) \leq \cdots \leq [s_{a_j}, f_{a_j}) \leq [s_{b_{j+1}}, f_{b_{j+1}}) \leq [s_{b_{j+2}}, f_{b_{j+2}}) \leq \cdots \leq [s_{b_\ell}, f_{b_\ell})$$

- Since b_{j+1} is compatible with a_j, we must have $f_{b_{j+1}} \geq f_i$,
  - If $f_{b_{j+1}} < f_i$, then b_{j+1} would have been processed before i in the algorithm;
  - When we process b_{j+1}, we would add b_{j+1} to $C = \{a_1, a_2, \ldots, a_j\}$, contradicting that i is the interval added to C after a_j.

Proof of the Proposition

- Suppose before adding i to C, $C = \{a_1, a_2, \ldots, a_j\}$
- By the assumption, we have that $C = \{a_1, a_2, \ldots, a_j\}$ is contained in an optimal solution, and we want to show that $\{a_1, a_2, \ldots, a_j, i\}$ is contained in an optimal solution.
- Let $O = \{a_1, a_2, \ldots, a_j, b_{j+1}, b_{j+2}, \ldots, b_\ell\}$ be an optimal solution containing C
- Since intervals in O are compatible, we can assume they are ordered:

$$[s_{a_1}, f_{a_1}) \leq [s_{a_2}, f_{a_2}) \leq \cdots \leq [s_{a_j}, f_{a_j}) \leq [s_{b_{j+1}}, f_{b_{j+1}}) \leq [s_{b_{j+2}}, f_{b_{j+2}}) \leq \cdots \leq [s_{b_\ell}, f_{b_\ell})$$

- Since b_{j+1} is compatible with a_j, we must have $f_{b_{j+1}} \geq f_i$,
  - If $f_{b_{j+1}} < f_i$, then b_{j+1} would have been processed before i in the algorithm;
  - When we process b_{j+1}, we would add b_{j+1} to $C = \{a_1, a_2, \ldots, a_j\}$, contradicting that i is the interval added to C after a_j.
- Since $f_{b_{j+1}} \geq f_i$, we could safely replace b_{j+1} with i in O, producing another optimal solution containing $\{a_1, a_2, \ldots, a_j, i\}$

Interval Scheduling: Justification

- Notice that the previous slides only tell you that the set of intervals C returned by the greedy algorithm is ***contained in*** an optimal solution O

- Notice that the previous slides only tell you that the set of intervals C returned by the greedy algorithm is ***contained in*** an optimal solution O
- But we need to show that C ***is*** the optimal solution O ($C = O$)

Interval Scheduling: Justification

- Notice that the previous slides only tell you that the set of intervals C returned by the greedy algorithm is ***contained in*** an optimal solution O
- But we need to show that C ***is*** the optimal solution O ($C = O$)
- Assume O has an addition interval b_{j+1} after $C = \{a_1, a_2, \ldots, a_j\}$, then by the algorithm, b_{j+1} must be added to C when processing b_{j+1}, contradicting that b_{j+1} is not in C

Why designing greedy algorithms is not easy

Greedy Choices that ***Do Not*** Work:

- Chose the activity that starts first
- Chose the shortest activity
- Chose the activity that overlaps with the fewest number of activities

Counter examples for previous strategies

counterexample for earliest start time

counterexample for shortest interval

counterexample for fewest conflicts

(Figure from Kleinberg & Tardos slides)

Interval Partitioning

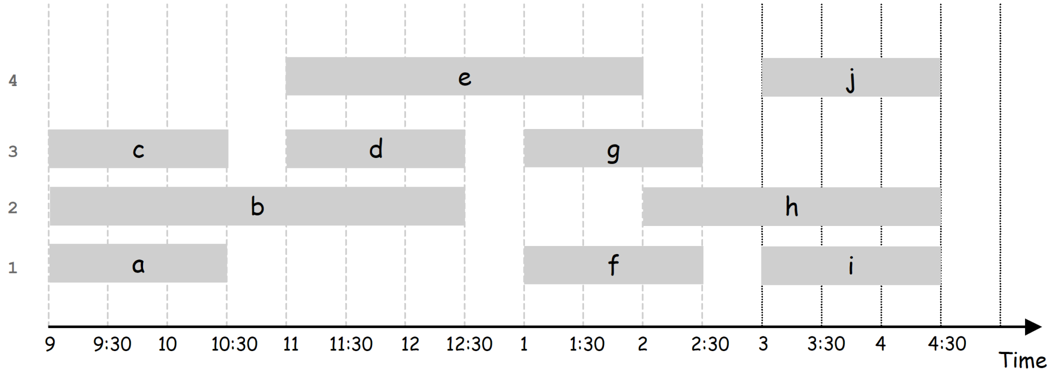
- We have n lectures; each lecture i starts at s_i and finishes at f_i (i.e., happens in $[s_i, f_i)$)
- Goal: find minimum number of classrooms to schedule all lectures so that lectures in the same room are compatible (disjoint)

Interval Partitioning

- We have n lectures; each lecture i starts at s_i and finishes at f_i (i.e., happens in $[s_i, f_i)$)
- Goal: find minimum number of classrooms to schedule all lectures so that lectures in the same room are compatible (disjoint)
- This is called 'interval partitioning' because we are trying to partition the given set of intervals into a few subsets s.t. intervals in each subset are compatible
- From now on, 'intervals' and 'lectures' are used interchangeably

Example

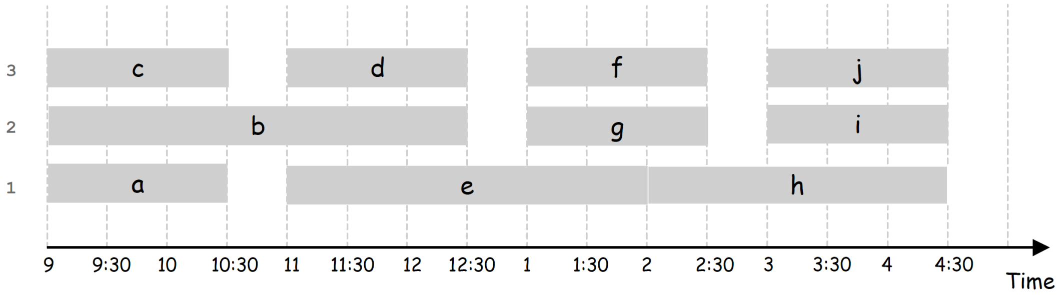
This partitioning uses 4 classrooms to schedule 10 lectures:



(Figure from from Kleinberg & Tardos slides)

Example

This partitioning uses only 3 classrooms:



(Figure from from Kleinberg & Tardos slides)

Important Concept: Depth

Definition

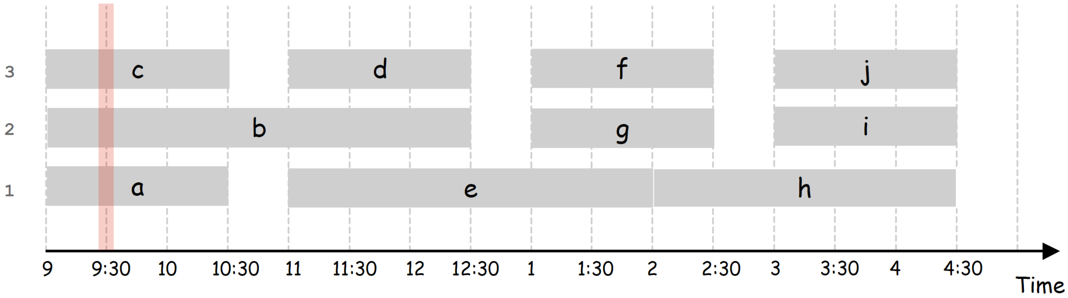
The ***depth*** of a given set of lectures (intervals) is the maximum number of lectures held at the same time

Important Concept: Depth

Definition

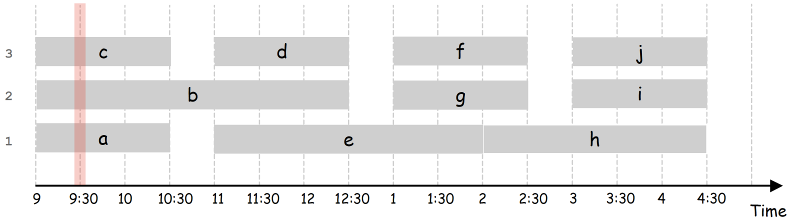
The ***depth*** of a given set of lectures (intervals) is the maximum number of lectures held at the same time

Example: depth of the previous set of lectures is 3



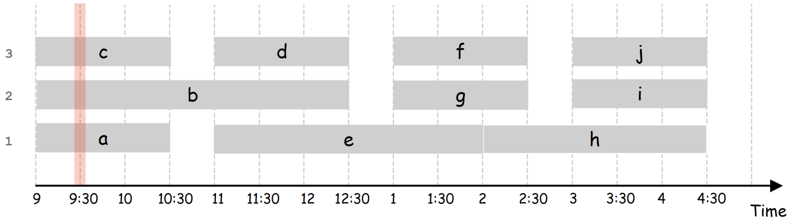
(Figure from from Kleinberg & Tardos slides)

Important Concept: Depth



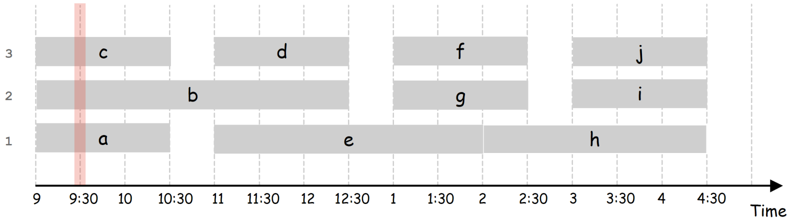
- Why do we care about the ***depth*** of a set of lectures?

Important Concept: Depth



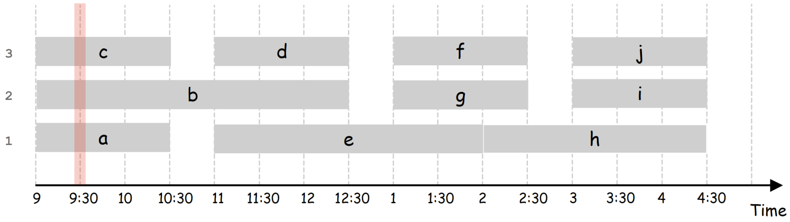
- Why do we care about the ***depth*** of a set of lectures?
- Observe that the number of classrooms needed ***cannot be smaller*** than the depth

Important Concept: Depth



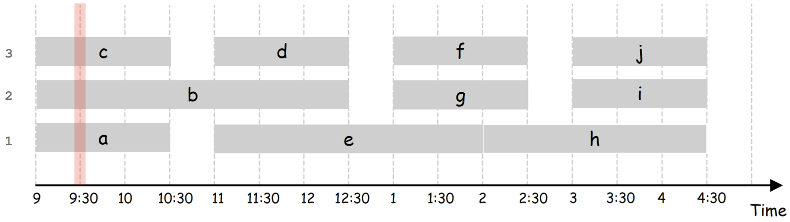
- Why do we care about the ***depth*** of a set of lectures?
- Observe that the number of classrooms needed ***cannot be smaller*** than the depth
  - If depth $= d$, this means that there are d lectures held at the same time
  - Each of the d lectures must be in a separate classroom

Important Concept: Depth



- Why do we care about the ***depth*** of a set of lectures?
- Observe that the number of classrooms needed ***cannot be smaller*** than the depth
  - If depth $= d$, this means that there are d lectures held at the same time
  - Each of the d lectures must be in a separate classroom
- So if we are able to schedule (partition) the lectures into d classrooms, this scheduling must be minimum (see the example above)

Important Concept: Depth



- Why do we care about the ***depth*** of a set of lectures?
- Observe that the number of classrooms needed ***cannot be smaller*** than the depth
  - If depth $= d$, this means that there are d lectures held at the same time
  - Each of the d lectures must be in a separate classroom
- So if we are able to schedule (partition) the lectures into d classrooms, this scheduling must be minimum (see the example above)
- We shall see a greedy algorithm which ***always*** schedules the lectures into d classrooms

Greedy algorithm. Go over each lecture in ***increasing order of start time***:

- assign each lecture to any compatible classroom you already have
- if there is no compatible classroom, allocate a new one

Interval Partitioning: Greedy Algorithm

Greedy algorithm. Go over each lecture in ***increasing order of start time***:

- assign each lecture to any compatible classroom you already have
- if there is no compatible classroom, allocate a new one

GreedyIntervPartition$(\{s_1, \ldots, s_n\}, \{f_1, \ldots, f_n\})$

sort and renumber the lectures s.t.
    $s_1 \leq s_2 \leq \cdots \leq s_n$
$C = 0$ // number of classrooms allocated
for $i = 1, \ldots, n$:
    if lecture i is compatible with lectures in a classroom k already allocated
        schedule lecture i in classroom k
    else
        allocate a new classroom
        schedule lecture i in the new classroom
        $C = C + 1$
return C

Greedy Algorithm: Correctness

- Let C be the number of classrooms schedule by the greedy algorithm

Greedy Algorithm: Correctness

- Let C be the number of classrooms schedule by the greedy algorithm
- Consider the step i where the ***last classroom*** is allocated

Greedy Algorithm: Correctness

- Let C be the number of classrooms schedule by the greedy algorithm
- Consider the step i where the ***last classroom*** is allocated
- We have that there are $C - 1$ lectures in the existing $C - 1$ classrooms which are incompatible with lecture i

Greedy Algorithm: Correctness

- Let C be the number of classrooms schedule by the greedy algorithm
- Consider the step i where the ***last classroom*** is allocated
- We have that there are $C - 1$ lectures in the existing $C - 1$ classrooms which are incompatible with lecture i
- Since we process the lectures in the ***order of starting time***, we have that the $C - 1$ incompatible lectures must ***start before*** s_i

Greedy Algorithm: Correctness

- Let C be the number of classrooms schedule by the greedy algorithm
- Consider the step i where the ***last classroom*** is allocated
- We have that there are $C - 1$ lectures in the existing $C - 1$ classrooms which are incompatible with lecture i
- Since we process the lectures in the ***order of starting time***, we have that the $C - 1$ incompatible lectures must ***start before*** s_i
- So these $C - 1$ incompatible lectures must ***end after*** s_i

Greedy Algorithm: Correctness

- Let C be the number of classrooms schedule by the greedy algorithm
- Consider the step i where the ***last classroom*** is allocated
- We have that there are $C - 1$ lectures in the existing $C - 1$ classrooms which are incompatible with lecture i
- Since we process the lectures in the ***order of starting time***, we have that the $C - 1$ incompatible lectures must ***start before*** s_i
- So these $C - 1$ incompatible lectures must ***end after*** s_i
- So at time s_i the $C - 1$ lectures and lecture i are being ***held together***

Greedy Algorithm: Correctness

- Let C be the number of classrooms schedule by the greedy algorithm
- Consider the step i where the ***last classroom*** is allocated
- We have that there are $C - 1$ lectures in the existing $C - 1$ classrooms which are incompatible with lecture i
- Since we process the lectures in the ***order of starting time***, we have that the $C - 1$ incompatible lectures must ***start before*** s_i
- So these $C - 1$ incompatible lectures must ***end after*** s_i
- So at time s_i the $C - 1$ lectures and lecture i are being ***held together***
- The ***depth*** of all lectures is $\geq C$

Greedy Algorithm: Correctness

- Let C be the number of classrooms schedule by the greedy algorithm
- Consider the step i where the ***last classroom*** is allocated
- We have that there are $C - 1$ lectures in the existing $C - 1$ classrooms which are incompatible with lecture i
- Since we process the lectures in the ***order of starting time***, we have that the $C - 1$ incompatible lectures must ***start before*** s_i
- So these $C - 1$ incompatible lectures must ***end after*** s_i
- So at time s_i the $C - 1$ lectures and lecture i are being ***held together***
- The ***depth*** of all lectures is $\geq C$
- So there is ***no scheduling*** with number of classrooms $< C$

Implementing the Interval Partitioning Algorithm

- In Line 4 of the greedy algorithm, we need to test whether lecture i is compatible a classroom k already allocated
- To implement this efficiently is not trivial: the most naive way is to go over each lecture in each classroom, which takes $O(n)$ time in the worst case (so overall complexity is $O(n^2)$)
- The algorithm can be implemented in $O(n \log n)$ time by doing things smartly

Implementing the Interval Partitioning Algorithm

Idea:

- From the previous interval scheduling problem, we have that a lecture j is compatible with all lectures in a classroom i iff $F_i \leq s_j$, where F_i is the finishing time of the ***latest*** lecture in classroom i

Implementing the Interval Partitioning Algorithm

Idea:

- From the previous interval scheduling problem, we have that a lecture j is compatible with all lectures in a classroom i iff $F_i \leq s_j$, where F_i is the finishing time of the ***latest*** lecture in classroom i
- So we keep the time F_i for each classroom in our algorithm, and when we examine a lecture j, we only need to see whether there exists a classroom i whose $F_i \leq s_j$

Implementing the Interval Partitioning Algorithm

Idea:

- From the previous interval scheduling problem, we have that a lecture j is compatible with all lectures in a classroom i iff $F_i \leq s_j$, where F_i is the finishing time of the ***latest*** lecture in classroom i
- So we keep the time F_i for each classroom in our algorithm, and when we examine a lecture j, we only need to see whether there exists a classroom i whose $F_i \leq s_j$
- This is equivalent to doing the following: take the class ι whose F_ι is the ***smallest*** (earliest) among all classrooms, and check whether $F_\iota \leq s_j$

Implementing the Interval Partitioning Algorithm

Idea:

- From the previous interval scheduling problem, we have that a lecture j is compatible with all lectures in a classroom i iff $F_i \le s_j$, where F_i is the finishing time of the ***latest*** lecture in classroom i

- So we keep the time F_i for each classroom in our algorithm, and when we examine a lecture j, we only need to see whether there exists a classroom i whose $F_i \le s_j$

- This is equivalent to doing the following: take the class ι whose F_ι is the ***smallest*** (earliest) among all classrooms, and check whether $F_\iota \le s_j$

- We use a ***heap*** to keep all F_i's for the classrooms, and can retrieve the smallest finishing time F_ι in $O(\log n)$ time for the $O(n)$ classrooms

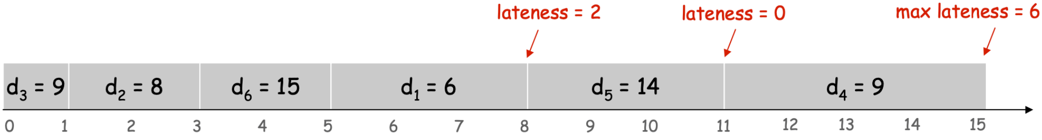
Minimizing Lateness Problem

- We have a bunch of jobs $1, 2, \dots, n$ and a single machine which processes one job at a time
- Each job j requires t_j units of time to process and has a due time d_j
  - i.e., if j starts at time s, it finishes at time $f_j = s + t_j$
- Suppose job j finishes at f_j. Define ***Lateness*** of job j as: $l_j = \max\{0, f_j - d_j\}$
- Goal: Find an order for executing the jobs to minimize maximum lateness $\max_{j=1,\dots,n}\{l_j\}$

Scheduling to Minimizing Lateness

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



(Figure from Kleinberg & Tardos slides)

Minimizing Lateness: Greedy Strategy

- The algorithms will be in very simple forms, i.e., we only need to figure out an order of the jobs based on certain criteria

Minimizing Lateness: Greedy Strategy

- The algorithms will be in very simple forms, i.e., we only need to figure out an order of the jobs based on certain criteria
- The problem is which criterion to use:
  - [Shortest processing time first]: Execute jobs in ***ascending order of processing time*** t_j

	1	2
t_j	1	10
d_j	100	10

counterexample

  - [Smallest slack]: Consider jobs in ***ascending order of slack*** $d_j - t_j$

	1	2
t_j	1	10
d_j	2	10

counterexample

- (Figures from Kleinberg & Tardos slides)

Minimizing Lateness: Greedy Strategy

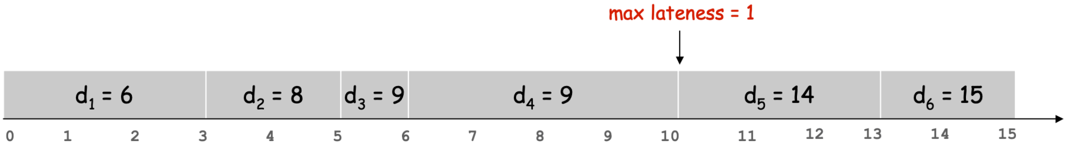
- The correct strategy is to simply execute the jobs by the ***ascending order of the due time***

Minimizing Lateness: Greedy Strategy

- The correct strategy is to simply execute the jobs by the ***ascending order of the due time***
- On the previous example:

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



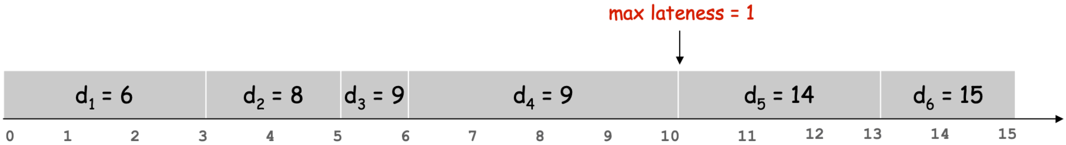
(Figure from Kleinberg & Tardos slides)

Minimizing Lateness: Greedy Strategy

- The correct strategy is to simply execute the jobs by the ***ascending order of the due time***
- On the previous example:

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



(Figure from Kleinberg & Tardos slides)

- ***Why is this?***

Justification of the Greedy Strategy

- Assume that jobs are numbered by their due time (i.e., $d_1 \le d_2 \le \cdots \le d_n$) and there is no gap between the execution of two jobs
  - If we have an optimal solution with gaps, then we can simply eliminate the gaps and get another optimal solution

- Assume that jobs are numbered by their due time (i.e., $d_1 \leq d_2 \leq \cdots \leq d_n$) and there is no gap between the execution of two jobs
  - If we have an optimal solution with gaps, then we can simply eliminate the gaps and get another optimal solution

Definition

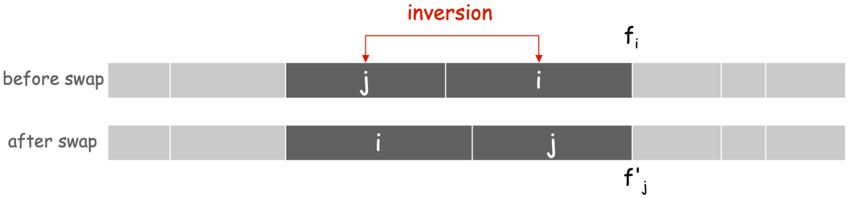
For an order of job execution, an ***inversion*** is a pair of jobs *i* and *j* such that $i < j$ but *j* scheduled before *i*

(Figure from Kleinberg & Tardos slides)

Justification of the Greedy Strategy

Proposition

Swapping a consecutive inversion in an execution does not increase the maximum lateness



(Figure from Kleinberg & Tardos slides)

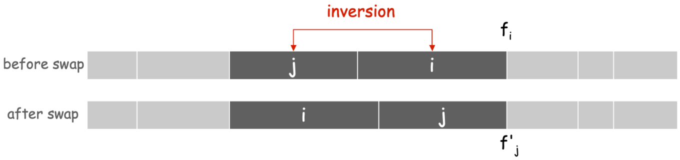
Proof:

- Let $f_1, \ldots, f_n$ be the finishing time of jobs before the swap, and let $f'_1, \ldots, f'_n$ be their finishing time after
- Let $l_1, \ldots, l_n$ be the lateness of jobs before the swap and $l'_1, \ldots, l'_n$ be the lateness after

Justification of the Greedy Strategy

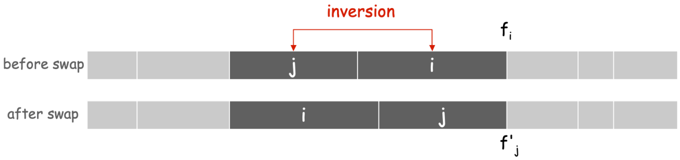
Proof:

- Let $f_1, \ldots, f_n$ be the finishing time of jobs before the swap, and let $f'_1, \ldots, f'_n$ be their finishing time after
- Let $l_1, \ldots, l_n$ be the lateness of jobs before the swap and $l'_1, \ldots, l'_n$ be the lateness after
- We have some easy facts: $l'_k = l_k$ for $k \neq i, j$ and $l'_i \leq l_i$



Proof:

- Let $f_1, \ldots, f_n$ be the finishing time of jobs before the swap, and let $f'_1, \ldots, f'_n$ be their finishing time after
- Let $l_1, \ldots, l_n$ be the lateness of jobs before the swap and $l'_1, \ldots, l'_n$ be the lateness after
- We have some easy facts: $l'_k = l_k$ for $k \neq i, j$ and $l'_i \leq l_i$

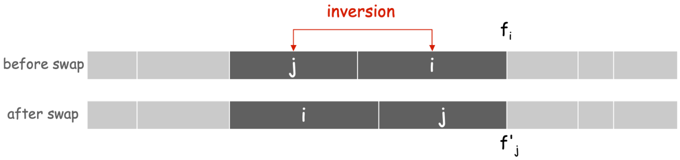


- So the only job that can make the max lateness to increase is j

Justification of the Greedy Strategy

Proof:

- Let $f_1, \ldots, f_n$ be the finishing time of jobs before the swap, and let $f'_1, \ldots, f'_n$ be their finishing time after
- Let $l_1, \ldots, l_n$ be the lateness of jobs before the swap and $l'_1, \ldots, l'_n$ be the lateness after
- We have some easy facts: $l'_k = l_k$ for $k \neq i, j$ and $l'_i \leq l_i$

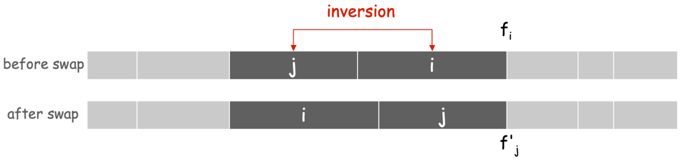


- So the only job that can make the max lateness to increase is j
- Consider the case that job j is late after the swap (i.e., $f'_j > d_j$)
  - The case that j is not late is easier and is omitted

Justification of the Greedy Strategy

Proof:

- Let $f_1, \ldots, f_n$ be the finishing time of jobs before the swap, and let $f'_1, \ldots, f'_n$ be their finishing time after
- Let $l_1, \ldots, l_n$ be the lateness of jobs before the swap and $l'_1, \ldots, l'_n$ be the lateness after
- We have some easy facts: $l'_k = l_k$ for $k \neq i, j$ and $l'_i \leq l_i$

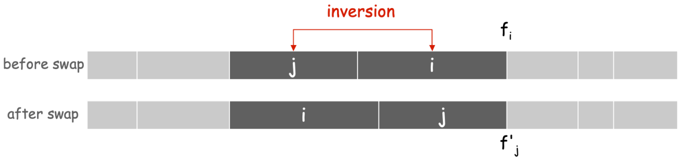


- So the only job that can make the max lateness to increase is j
- Consider the case that job j is late after the swap (i.e., $f'_j > d_j$)
  - The case that j is not late is easier and is omitted
- We have: $l'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i \leq l_i$

Justification of the Greedy Strategy

Proof:

- Let $f_1, \ldots, f_n$ be the finishing time of jobs before the swap, and let $f'_1, \ldots, f'_n$ be their finishing time after
- Let $l_1, \ldots, l_n$ be the lateness of jobs before the swap and $l'_1, \ldots, l'_n$ be the lateness after
- We have some easy facts: $l'_k = l_k$ for $k \neq i, j$ and $l'_i \leq l_i$

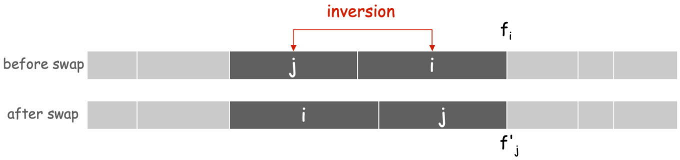


- So the only job that can make the max lateness to increase is j
- Consider the case that job j is late after the swap (i.e., $f'_j > d_j$)
  - The case that j is not late is easier and is omitted
- We have: $l'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i \leq l_i$
- We have shown that for each element in $L' = \{l'_1, \ldots, l'_n\}$, there is an element in $L = \{l_1, \ldots, l_n\}$ greater than or equal to it

Justification of the Greedy Strategy

Proof:

- Let $f_1, \ldots, f_n$ be the finishing time of jobs before the swap, and let $f'_1, \ldots, f'_n$ be their finishing time after
- Let $l_1, \ldots, l_n$ be the lateness of jobs before the swap and $l'_1, \ldots, l'_n$ be the lateness after
- We have some easy facts: $l'_k = l_k$ for $k \neq i, j$ and $l'_i \leq l_i$



- So the only job that can make the max lateness to increase is j
- Consider the case that job j is late after the swap (i.e., $f'_j > d_j$)
  - The case that j is not late is easier and is omitted
- We have: $l'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i \leq l_i$
- We have shown that for each element in $L' = \{l'_1, \ldots, l'_n\}$, there is an element in $L = \{l_1, \ldots, l_n\}$ greater than or equal to it
- So $\max L' \leq \max L$

Justification of the Greedy Strategy

Proposition

Executing the jobs by their ascending order of due time yields a solution which minimizes the maximum lateness

Justification of the Greedy Strategy

Proposition

Executing the jobs by their ascending order of due time yields a solution which minimizes the maximum lateness

Proof:

- Let O be an optimal solution
- If O is not the greedy solution (i.e., job are not ordered by their numbers), we can always transform O into the greedy solution by swapping consecutive inverted jobs.
- Since the swap does not increase the max lateness, we still get an optimal solution after the swap
- This means that the greedy solution is an optimal solution