Greedy Algorithms

Lecture 12

March 3, 2015

Part I

Problems and Terminology

Problem Types

1. Decision Problem: Is the input a YES or NO input?
   Example: Given graph G, nodes s, t, is there a path from s to t in G?
2. Search Problem: Find a *solution* if input is a YES input.
   Example: Given graph G, nodes s, t, find an s-t path.
3. Optimization Problem: Find a *best* solution among all solutions for the input.
   Example: Given graph G, nodes s, t, find a shortest s-t path.

Problem Types

1. Decision Problem: Is the input a YES or NO input?
   Example: Given graph G, nodes s, t, is there a path from s to t in G?
2. Search Problem: Find a *solution* if input is a YES input.
   Example: Given graph G, nodes s, t, find an s-t path.
3. Optimization Problem: Find a *best* solution among all solutions for the input.
   Example: Given graph G, nodes s, t, find a shortest s-t path.

Problem Types

1. Decision Problem: Is the input a YES or NO input?
   Example: Given graph G, nodes s, t, is there a path from s to t in G?
2. Search Problem: Find a *solution* if input is a YES input.
   Example: Given graph G, nodes s, t, find an s-t path.
3. Optimization Problem: Find a *best* solution among all solutions for the input.
   Example: Given graph G, nodes s, t, find a shortest s-t path.

Terminology

1. A problem Π consists of an *infinite* collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.
2. The size of an instance I is the number of bits in its representation.
3. I: instance. $sol(I)$: set of feasible solutions to I.
4. *Implicit assumption:* given I, $y \in \Sigma^*$, one can check (efficiently!) if $y \in sol(I)$.
5. $\implies$ Problem is in **NP**. (More on this later in the course.)
6. Optimization problems: $\forall$ solution $s \in sol(I)$ has associated value.
7. *Implicit assumption:* given s, can compute value efficiently.

Terminology

1. A problem Π consists of an *infinite* collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.
2. The size of an instance I is the number of bits in its representation.
3. I: instance. $sol(I)$: set of feasible solutions to I.
4. *Implicit assumption:* given I, $y \in \Sigma^*$, one can check (efficiently!) if $y \in sol(I)$.
5. $\implies$ Problem is in **NP**. (More on this later in the course.)
6. Optimization problems:
   $\forall$ solution $s \in sol(I)$ has associated value.
7. *Implicit assumption:* given s, can compute value efficiently.

Terminology

1. A problem Π consists of an *infinite* collection of inputs $\{I_1, I_2, \ldots,\}$. Each input is referred to as an instance.
2. The size of an instance I is the number of bits in its representation.
3. I: instance. $sol(I)$: set of feasible solutions to I.
4. *Implicit assumption:* given I, $y \in \Sigma^*$, one can check (efficiently!) if $y \in sol(I)$.
5. $\implies$ Problem is in **NP**. (More on this later in the course.)
6. Optimization problems:
   $\forall$ solution $s \in sol(I)$ has associated value.
7. *Implicit assumption:* given s, can compute value efficiently.

Terminology

1. A problem Π consists of an *infinite* collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.
2. The size of an instance I is the number of bits in its representation.
3. I: instance. $sol(I)$: set of feasible solutions to I.
4. *Implicit assumption:* given I, $y \in \Sigma^*$, one can check (efficiently!) if $y \in sol(I)$.
5. $\Longrightarrow$ Problem is in **NP**. (More on this later in the course.)
6. Optimization problems:
   $\forall$ solution $s \in sol(I)$ has associated value.
7. *Implicit assumption:* given s, can compute value efficiently.

Terminology

1. A problem Π consists of an *infinite* collection of inputs $\{I_1, I_2, \ldots,\}$. Each input is referred to as an instance.
2. The size of an instance I is the number of bits in its representation.
3. I: instance. $sol(I)$: set of feasible solutions to I.
4. *Implicit assumption:* given I, $y \in \Sigma^*$, one can check (efficiently!) if $y \in sol(I)$.
5. $\implies$ Problem is in **NP**. (More on this later in the course.)
6. Optimization problems:
   $\forall$ solution $s \in sol(I)$ has associated value.
7. *Implicit assumption:* given s, can compute value efficiently.

Terminology

1. A problem Π consists of an *infinite* collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.
2. The size of an instance I is the number of bits in its representation.
3. I: instance. $sol(I)$: set of feasible solutions to I.
4. *Implicit assumption:* given I, $y \in \Sigma^*$, one can check (efficiently!) if $y \in sol(I)$.
5. $\implies$ Problem is in **NP**. (More on this later in the course.)
6. Optimization problems:
   $\forall$ solution $s \in sol(I)$ has associated value.
7. *Implicit assumption:* given s, can compute value efficiently.

Terminology

1. A problem Π consists of an *infinite* collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.
2. The size of an instance I is the number of bits in its representation.
3. I: instance. $sol(I)$: set of feasible solutions to I.
4. *Implicit assumption:* given I, $y \in \Sigma^*$, one can check (efficiently!) if $y \in sol(I)$.
5. $\implies$ Problem is in **NP**. (More on this later in the course.)
6. Optimization problems:
   $\forall$ solution $s \in sol(I)$ has associated value.
7. *Implicit assumption:* given s, can compute value efficiently.

Terminology

1. A problem Π consists of an *infinite* collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.
2. The size of an instance I is the number of bits in its representation.
3. I: instance. $sol(I)$: set of feasible solutions to I.
4. *Implicit assumption:* given I, $y \in \Sigma^*$, one can check (efficiently!) if $y \in sol(I)$.
5. $\implies$ Problem is in **NP**. (More on this later in the course.)
6. Optimization problems:
   $\forall$ solution $s \in sol(I)$ has associated value.
7. *Implicit assumption:* given s, can compute value efficiently.

Problem Types

Given instance I...

1. Decision Problem: Output whether $sol(I) = \emptyset$ or not.
2. Search Problem: Compute solution $s \in sol(I)$ if $sol(I) \neq \emptyset$.
3. Optimization Problem: Given I,
   1. Minimization problem. Find solution $s \in sol(I)$ of **min** value.
   2. Maximization problem. Find solution $s \in sol(I)$ of **max** value.
   3. Notation:
      $opt(I)$: denote the value of an optimum solution or some fixed optimum solution.

Problem Types

Given instance I...

1. Decision Problem: Output whether $sol(I) = \emptyset$ or not.
2. Search Problem: Compute solution $s \in sol(I)$ if $sol(I) \neq \emptyset$.
3. Optimization Problem: Given I,
   1. Minimization problem. Find solution $s \in sol(I)$ of **min** value.
   2. Maximization problem. Find solution $s \in sol(I)$ of **max** value.
   3. Notation:
      $opt(I)$: denote the value of an optimum solution or some fixed optimum solution.

Problem Types

Given instance I...

1. Decision Problem: Output whether $sol(I) = \emptyset$ or not.
2. Search Problem: Compute solution $s \in sol(I)$ if $sol(I) \neq \emptyset$.
3. Optimization Problem: Given I,
   1. Minimization problem. Find solution $s \in sol(I)$ of **min** value.
   2. Maximization problem. Find solution $s \in sol(I)$ of **max** value.
   3. Notation:
      $opt(I)$: denote the value of an optimum solution or some fixed optimum solution.

Problem Types

Given instance I...

1. Decision Problem: Output whether $sol(I) = \emptyset$ or not.
2. Search Problem: Compute solution $s \in sol(I)$ if $sol(I) \neq \emptyset$.
3. Optimization Problem: Given I,
   1. Minimization problem. Find solution $s \in sol(I)$ of **min** value.
   2. Maximization problem. Find solution $s \in sol(I)$ of **max** value.
   3. Notation:
      $opt(I)$: denote the value of an optimum solution or some fixed optimum solution.

Problem Types

Given instance I...

1. Decision Problem: Output whether $sol(I) = \emptyset$ or not.
2. Search Problem: Compute solution $s \in sol(I)$ if $sol(I) \neq \emptyset$.
3. Optimization Problem: Given I,
    1. Minimization problem. Find solution $s \in sol(I)$ of **min** value.
    2. Maximization problem. Find solution $s \in sol(I)$ of **max** value.
    3. Notation:

       $opt(I)$: denote the value of an optimum solution or some fixed optimum solution.

Problem Types

Given instance I...

1. Decision Problem: Output whether $sol(I) = \emptyset$ or not.
2. Search Problem: Compute solution $s \in sol(I)$ if $sol(I) \neq \emptyset$.
3. Optimization Problem: Given I,
    1. Minimization problem. Find solution $s \in sol(I)$ of **min** value.
    2. Maximization problem. Find solution $s \in sol(I)$ of **max** value.
    3. Notation:
       $opt(I)$: denote the value of an optimum solution or some fixed optimum solution.

Problem Types

Given instance I...

1. Decision Problem: Output whether $sol(I) = \emptyset$ or not.
2. Search Problem: Compute solution $s \in sol(I)$ if $sol(I) \neq \emptyset$.
3. Optimization Problem: Given I,
   1. Minimization problem. Find solution $s \in sol(I)$ of **min** value.
   2. Maximization problem. Find solution $s \in sol(I)$ of **max** value.
   3. Notation:
      $opt(I)$: denote the value of an optimum solution or some fixed optimum solution.

Part II

Greedy Algorithms: Tools and Techniques

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. Do the right thing. Locally.
2. Make decisions incrementally in small steps *no backtracking*.
3. Decision at each step based on improving *local or current* state in myopic fashion. ... without considering the *global* situation.
4. **myopia**: lack of understanding or foresight.
5. Decisions often based on some fixed and simple *priority* rules.

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. Do the right thing. Locally.
2. Make decisions incrementally in small steps *no backtracking*.
3. Decision at each step based on improving *local or current* state in myopic fashion. ... without considering the *global* situation.
4. **myopia**: lack of understanding or foresight.
5. Decisions often based on some fixed and simple *priority* rules.

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. Do the right thing. Locally.
2. Make decisions incrementally in small steps *no backtracking*.
3. Decision at each step based on improving *local or current* state in myopic fashion. ... without considering the *global* situation.
4. **myopia**: lack of understanding or foresight.
5. Decisions often based on some fixed and simple *priority* rules.

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. Do the right thing. Locally.
2. Make decisions incrementally in small steps *no backtracking*.
3. Decision at each step based on improving *local or current* state in myopic fashion. ... without considering the *global* situation.
4. **myopia**: lack of understanding or foresight.
5. Decisions often based on some fixed and simple *priority* rules.

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. Do the right thing. Locally.
2. Make decisions incrementally in small steps *no backtracking*.
3. Decision at each step based on improving *local or current* state in myopic fashion. ... without considering the *global* situation.
4. **myopia**: lack of understanding or foresight.
5. Decisions often based on some fixed and simple *priority* rules.

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. Do the right thing. Locally.
2. Make decisions incrementally in small steps *no backtracking*.
3. Decision at each step based on improving *local or current* state in myopic fashion. ... without considering the *global* situation.
4. **myopia**: lack of understanding or foresight.
5. Decisions often based on some fixed and simple *priority* rules.

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. Do the right thing. Locally.
2. Make decisions incrementally in small steps *no backtracking*.
3. Decision at each step based on improving *local or current* state in myopic fashion. ... without considering the *global* situation.
4. **myopia**: lack of understanding or foresight.
5. Decisions often based on some fixed and simple *priority* rules.

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. Do the right thing. Locally.
2. Make decisions incrementally in small steps *no backtracking*.
3. Decision at each step based on improving *local or current* state in myopic fashion. ... without considering the *global* situation.
4. **myopia**: lack of understanding or foresight.
5. Decisions often based on some fixed and simple *priority* rules.

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. Do the right thing. Locally.
2. Make decisions incrementally in small steps *no backtracking*.
3. Decision at each step based on improving *local or current* state in myopic fashion. ... without considering the *global* situation.
4. **myopia**: lack of understanding or foresight.
5. Decisions often based on some fixed and simple *priority* rules.

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work.
2. Easy to lull oneself into believing they work
3. Many greedy algorithms possible for a problem and no structured way to find effective ones.

CS 473: Every greedy algorithm needs a proof of correctness

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work.
2. Easy to lull oneself into believing they work
3. Many greedy algorithms possible for a problem and no structured way to find effective ones.

CS 473: Every greedy algorithm needs a proof of correctness

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work.
2. Easy to lull oneself into believing they work
3. Many greedy algorithms possible for a problem and no structured way to find effective ones.

CS 473: Every greedy algorithm needs a proof of correctness

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work.
2. Easy to lull oneself into believing they work
3. Many greedy algorithms possible for a problem and no structured way to find effective ones.

CS 473: Every greedy algorithm needs a proof of correctness

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work.
2. Easy to lull oneself into believing they work
3. Many greedy algorithms possible for a problem and no structured way to find effective ones.

CS 473: Every greedy algorithm needs a proof of correctness

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work.
2. Easy to lull oneself into believing they work
3. Many greedy algorithms possible for a problem and no structured way to find effective ones.

CS 473: Every greedy algorithm needs a proof of correctness

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work.
2. Easy to lull oneself into believing they work
3. Many greedy algorithms possible for a problem and no structured way to find effective ones.

CS 473: Every greedy algorithm needs a proof of correctness

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work.
2. Easy to lull oneself into believing they work
3. Many greedy algorithms possible for a problem and no structured way to find effective ones.

CS 473: Every greedy algorithm needs a proof of correctness

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work.
2. Easy to lull oneself into believing they work
3. Many greedy algorithms possible for a problem and no structured way to find effective ones.

CS 473: Every greedy algorithm needs a proof of correctness

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work.
2. Easy to lull oneself into believing they work
3. Many greedy algorithms possible for a problem and no structured way to find effective ones.

CS 473: Every greedy algorithm needs a proof of correctness

Greedy Algorithm Types

1. Crude classification:
   1. Non-adaptive: fix ordering of decisions a priori and stick with it.
   2. Adaptive: make decisions adaptively but greedily/locally at each step.
2. Plan:
   1. See several examples
   2. Pick up some proof techniques

Greedy Algorithm Types

1. Crude classification:
   1. Non-adaptive: fix ordering of decisions a priori and stick with it.
   2. Adaptive: make decisions adaptively but greedily/locally at each step.
2. Plan:
   1. See several examples
   2. Pick up some proof techniques

Greedy Algorithm Types

1. Crude classification:
   1. Non-adaptive: fix ordering of decisions a priori and stick with it.
   2. Adaptive: make decisions adaptively but greedily/locally at each step.
2. Plan:
   1. See several examples
   2. Pick up some proof techniques

Greedy Algorithm Types

1. Crude classification:
    1. Non-adaptive: fix ordering of decisions a priori and stick with it.
    2. Adaptive: make decisions adaptively but greedily/locally at each step.
2. Plan:
    1. See several examples
    2. Pick up some proof techniques

Greedy Algorithm Types

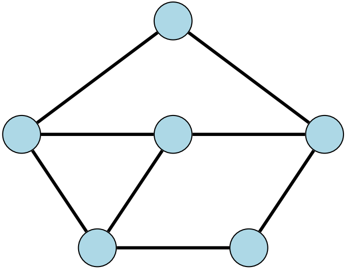
1. Crude classification:
    1. Non-adaptive: fix ordering of decisions a priori and stick with it.
    2. Adaptive: make decisions adaptively but greedily/locally at each step.
2. Plan:
    1. See several examples
    2. Pick up some proof techniques

Greedy Algorithm Types

1. Crude classification:
   1. Non-adaptive: fix ordering of decisions a priori and stick with it.
   2. Adaptive: make decisions adaptively but greedily/locally at each step.
2. Plan:
   1. See several examples
   2. Pick up some proof techniques

Vertex Cover

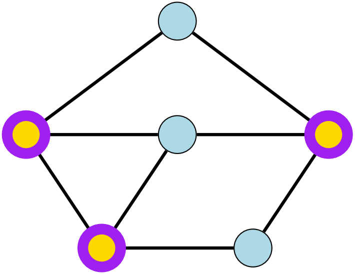
Given a graph $G = (V, E)$, a set of vertices S is:



Vertex Cover

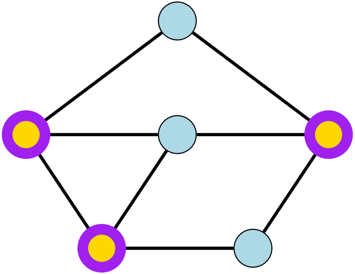
Given a graph $G = (V, E)$, a set of vertices S is:

1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:



Natural algorithms for computing vertex cover?

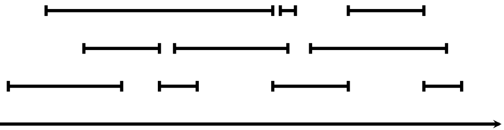
Interval Scheduling

Problem (Interval Scheduling)

Input: *A set of jobs with start and finish times to be scheduled on a resource (example: classes and class rooms).*

Goal: *Schedule as many jobs as possible*

1. *Two jobs with overlapping intervals cannot both be scheduled!*



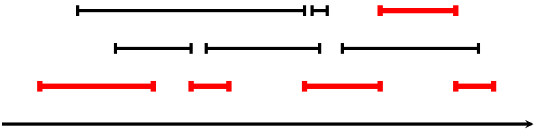
Interval Scheduling

Problem (Interval Scheduling)

Input: *A set of jobs with start and finish times to be scheduled on a resource (example: classes and class rooms).*

Goal: *Schedule as many jobs as possible*

1. *Two jobs with overlapping intervals cannot both be scheduled!*



Greedy Template

```
R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty do
    choose i ∈ R
    add i to X
    remove from R all requests that overlap with i
return the set X
```

Main task: Decide the order in which to process requests in R

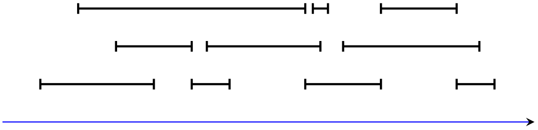
Greedy Template

```
R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty do
    choose i ∈ R
    add i to X
    remove from R all requests that overlap with i
return the set X
```

Main task: Decide the order in which to process requests in R

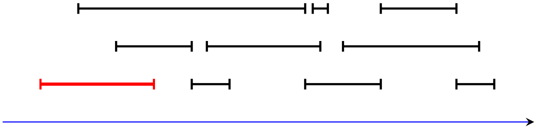
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



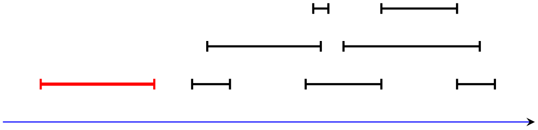
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



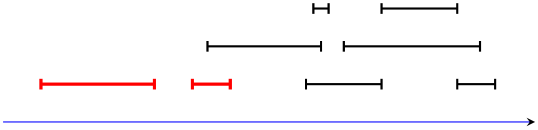
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



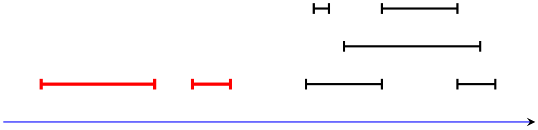
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



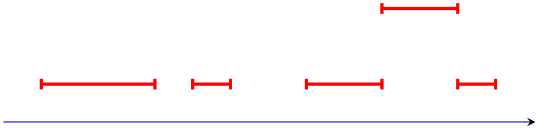
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.



Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

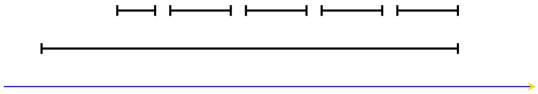


Figure: Counter example for earliest start time

Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

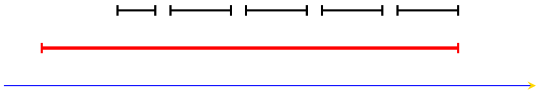


Figure: Counter example for earliest start time

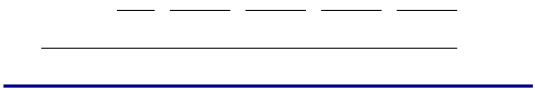
Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

Figure: Counter example for earliest start time

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.



Smallest Processing Time

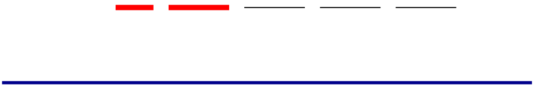
Process jobs in the order of processing time, starting with jobs that require the shortest processing.

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

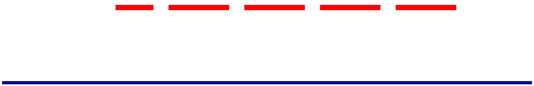
Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.



Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.



Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

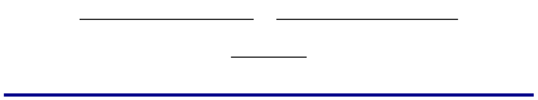


Figure: Counter example for smallest processing time

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

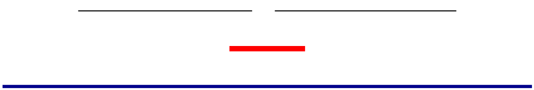


Figure: Counter example for smallest processing time

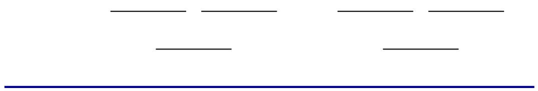
Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

Figure: Counter example for smallest processing time

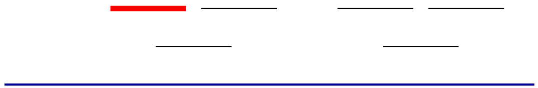
Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.



Fewest Conflicts

Process jobs in that have the fewest “conflicts” first.



Fewest Conflicts

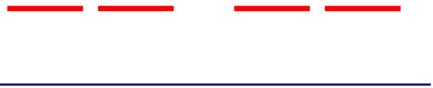
Process jobs in that have the fewest “conflicts” first.

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.



Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

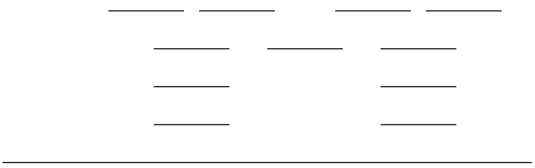


Figure: Counter example for fewest conflicts

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

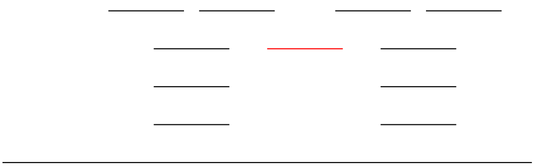


Figure: Counter example for fewest conflicts

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

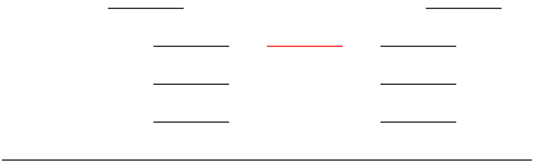


Figure: Counter example for fewest conflicts

Fewest Conflicts

Process jobs in that have the fewest “conflicts” first.

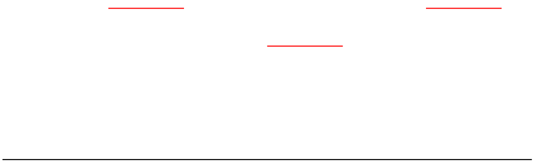
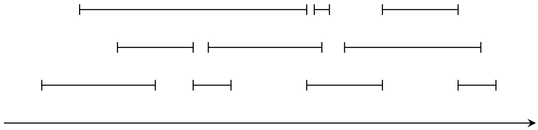


Figure: Counter example for fewest conflicts

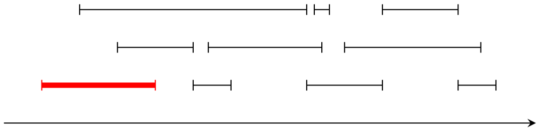
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



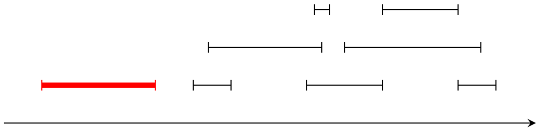
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



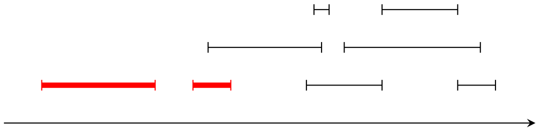
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



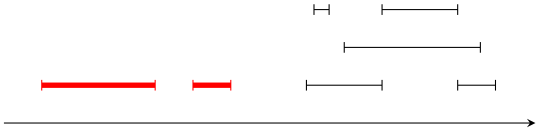
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



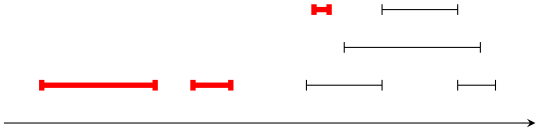
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



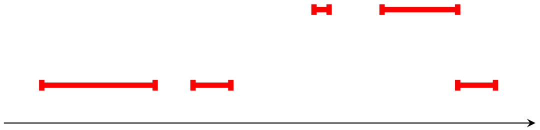
Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.



Optimal Greedy Algorithm

R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose $i \in R$ such that finishing time of i is least
    add i to X
    remove from R all requests that overlap with i
return X

Theorem

The greedy algorithm that picks jobs in the order of their finishing times is optimal.

Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests R, let O be an optimal set and let X be the set returned by the greedy algorithm. Then $O = X$? Not likely!

Instead we will show that $|O| = |X|$

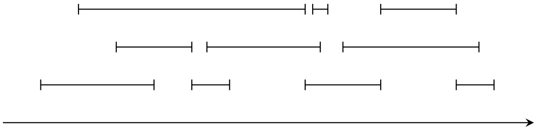
Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests R, let O be an optimal set and let X be the set returned by the greedy algorithm. Then $O = X$? Not likely!

Instead we will show that $|O| = |X|$

Proving Optimality

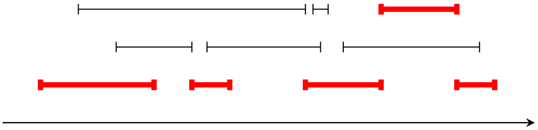
1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests R, let O be an optimal set and let X be the set returned by the greedy algorithm. Then $O = X$? Not likely!



Instead we will show that $|O| = |X|$

Proving Optimality

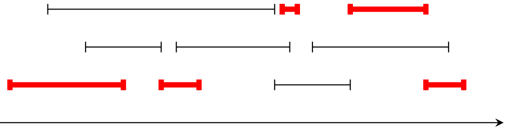
1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests R, let O be an optimal set and let X be the set returned by the greedy algorithm. Then $O = X$? Not likely!



Instead we will show that $|O| = |X|$

Proving Optimality

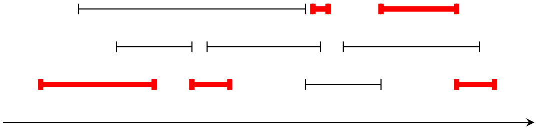
1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests R, let O be an optimal set and let X be the set returned by the greedy algorithm. Then $O = X$? Not likely!



Instead we will show that $|O| = |X|$

Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests R, let O be an optimal set and let X be the set returned by the greedy algorithm. Then $O = X$? Not likely!



Instead we will show that $|O| = |X|$

Proof of Optimality: Key Lemma

Lemma

Let i_1 be first interval picked by Greedy. There exists an optimum solution that contains i_1.

Proof.

Let O be an *arbitrary* optimum solution. If $i_1 \in O$ we are done.

Claim: If $i_1 \notin O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1. (proof later)

1. Form a new set O' by removing j_1 from O and adding i_1, that is $O' = (O - \{j_1\}) \cup \{i_1\}$.
2. From claim, O' is a *feasible* solution (no conflicts).
3. Since $|O'| = |O|$, O' is also an optimum solution and it contains i_1. □

Proof of Optimality: Key Lemma

Lemma

Let i_1 be first interval picked by Greedy. There exists an optimum solution that contains i_1.

Proof.

Let O be an *arbitrary* optimum solution. If $i_1 \in O$ we are done.
Claim: If $i_1 \notin O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1. (proof later)

1. Form a new set O' by removing j_1 from O and adding i_1, that is $O' = (O - \{j_1\}) \cup \{i_1\}$.
2. From claim, O' is a *feasible* solution (no conflicts).
3. Since $|O'| = |O|$, O' is also an optimum solution and it contains i_1.

$\square$

Proof of Optimality: Key Lemma

Lemma

Let i_1 be first interval picked by Greedy. There exists an optimum solution that contains i_1.

Proof.

Let O be an *arbitrary* optimum solution. If $i_1 \in O$ we are done.
Claim: If $i_1 \not\in O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1. (proof later)

1. Form a new set O' by removing j_1 from O and adding i_1, that is $O' = (O - \{j_1\}) \cup \{i_1\}$.
2. From claim, O' is a *feasible* solution (no conflicts).
3. Since $|O'| = |O|$, O' is also an optimum solution and it contains i_1. □

Proof of Optimality: Key Lemma

Lemma

Let i_1 be first interval picked by Greedy. There exists an optimum solution that contains i_1.

Proof.

Let O be an *arbitrary* optimum solution. If $i_1 \in O$ we are done.
Claim: If $i_1 \notin O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1. (proof later)

1. Form a new set O' by removing j_1 from O and adding i_1, that is $O' = (O - \{j_1\}) \cup \{i_1\}$.
2. From claim, O' is a *feasible* solution (no conflicts).
3. Since $|O'| = |O|$, O' is also an optimum solution and it contains i_1. □

Proof of Optimality: Key Lemma

Lemma

Let i_1 be first interval picked by Greedy. There exists an optimum solution that contains i_1.

Proof.

Let O be an *arbitrary* optimum solution. If $i_1 \in O$ we are done.
Claim: If $i_1 \notin O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1. (proof later)

1. Form a new set O' by removing j_1 from O and adding i_1, that is $O' = (O - \{j_1\}) \cup \{i_1\}$.
2. From claim, O' is a *feasible* solution (no conflicts).
3. Since $|O'| = |O|$, O' is also an optimum solution and it contains i_1. □

Proof of Optimality: Key Lemma

Lemma

Let i_1 be first interval picked by Greedy. There exists an optimum solution that contains i_1.

Proof.

Let O be an *arbitrary* optimum solution. If $i_1 \in O$ we are done.

Claim: If $i_1 \notin O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1. (proof later)

1. Form a new set O' by removing j_1 from O and adding i_1, that is $O' = (O - \{j_1\}) \cup \{i_1\}$.
2. From claim, O' is a *feasible* solution (no conflicts).
3. Since $|O'| = |O|$, O' is also an optimum solution and it contains i_1. $\square$

Proof of Claim

Claim

If $i_1 \notin O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1.

Proof.

1. Suppose $j_1, j_2 \in O$ such that $j_1 \neq j_2$ and both j_1 and j_2 conflict with i_1.
2. Since i_1 has earliest finish time, j_1 and i_1 overlap at $f(i_1)$.
3. For same reason j_2 also overlaps with i_1 at $f(i_1)$.
4. Implies that j_1, j_2 overlap at $f(i_1)$ contradicting the feasibility of O.

See figure in next slide.

Proof of Claim

Claim

If $i_1 \notin O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1.

Proof.

1. Suppose $j_1, j_2 \in O$ such that $j_1 \neq j_2$ and both j_1 and j_2 conflict with i_1.
2. Since i_1 has earliest finish time, j_1 and i_1 overlap at $f(i_1)$.
3. For same reason j_2 also overlaps with i_1 at $f(i_1)$.
4. Implies that j_1, j_2 overlap at $f(i_1)$ contradicting the feasibility of O.

See figure in next slide.

Proof of Claim

Claim

If $i_1 \notin O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1.

Proof.

1. Suppose $j_1, j_2 \in O$ such that $j_1 \neq j_2$ and both j_1 and j_2 conflict with i_1.
2. Since i_1 has earliest finish time, j_1 and i_1 overlap at $f(i_1)$.
3. For same reason j_2 also overlaps with i_1 at $f(i_1)$.
4. Implies that j_1, j_2 overlap at $f(i_1)$ contradicting the feasibility of O.

See figure in next slide.

Proof of Claim

Claim

If $i_1 \notin O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1.

Proof.

1. Suppose $j_1, j_2 \in O$ such that $j_1 \neq j_2$ and both j_1 and j_2 conflict with i_1.
2. Since i_1 has earliest finish time, j_1 and i_1 overlap at $f(i_1)$.
3. For same reason j_2 also overlaps with i_1 at $f(i_1)$.
4. Implies that j_1, j_2 overlap at $f(i_1)$ contradicting the feasibility of O.

See figure in next slide.

Proof of Claim

Claim

If $i_1 \not\in O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1.

Proof.

1. Suppose $j_1, j_2 \in O$ such that $j_1 \neq j_2$ and both j_1 and j_2 conflict with i_1.
2. Since i_1 has earliest finish time, j_1 and i_1 overlap at $f(i_1)$.
3. For same reason j_2 also overlaps with i_1 at $f(i_1)$.
4. Implies that j_1, j_2 overlap at $f(i_1)$ contradicting the feasibility of O.

See figure in next slide. □

Figure for proof of Claim

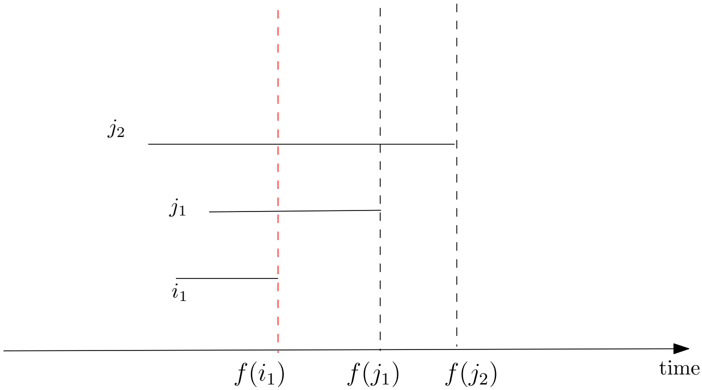


Figure: Since i_1 has the earliest finish time, any interval that conflicts with it does so at $f(i_1)$. This implies j_1 and j_2 conflict.

Proof of Optimality of Earliest Finish Time First

Proof by Induction on number of intervals.

Base Case: $n = 1$. Trivial since Greedy picks one interval.
Induction Step: Assume theorem holds for $i < n$.
Let I be an instance with n intervals
I': I with i_1 and all intervals that overlap with i_1 removed
$G(I), G(I')$: Solution produced by Greedy on I and I'
From Lemma, there is an optimum solution O to I and $i_1 \in O$.
Let $O' = O - \{i_1\}$. O' is a solution to I'.

$$\begin{aligned} |G(I)| &= 1 + |G(I')| \quad \text{(from Greedy description)} \\ &\leq 1 + |O'| \quad \text{(By induction, } G(I') \text{ is optimum for } I'\text{)} \\ &= |O| \end{aligned}$$

$\square$

Implementation and Running Time

```
Initially R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in X
        add i to X
    remove i from R
return the set X
```

1. Presort all requests based on finishing time. $O(n \log n)$ time
2. Now choosing least finishing time is $O(1)$
3. Keep track of the finishing time of the last request added to A. Then check if starting time of i later than that
4. Thus, checking non-overlapping is $O(1)$
5. Total time $O(n \log n + n) = O(n \log n)$

Implementation and Running Time

```
Initially R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in X
        add i to X
    remove i from R
return the set X
```

1. Presort all requests based on finishing time. $O(n \log n)$ time
2. Now choosing least finishing time is $O(1)$
3. Keep track of the finishing time of the last request added to A. Then check if starting time of i later than that
4. Thus, checking non-overlapping is $O(1)$
5. Total time $O(n \log n + n) = O(n \log n)$

Implementation and Running Time

```
Initially R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in X
        add i to X
    remove i from R
return the set X
```

1. Presort all requests based on finishing time. $O(n \log n)$ time
2. Now choosing least finishing time is $O(1)$
3. Keep track of the finishing time of the last request added to A. Then check if starting time of i later than that
4. Thus, checking non-overlapping is $O(1)$
5. Total time $O(n \log n + n) = O(n \log n)$

Implementation and Running Time

```
Initially R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in X
        add i to X
    remove i from R
return the set X
```

1. Presort all requests based on finishing time. $O(n \log n)$ time
2. Now choosing least finishing time is $O(1)$
3. Keep track of the finishing time of the last request added to A. Then check if starting time of i later than that
4. Thus, checking non-overlapping is $O(1)$
5. Total time $O(n \log n + n) = O(n \log n)$

Implementation and Running Time

```
Initially R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in X
        add i to X
    remove i from R
return the set X
```

1. Presort all requests based on finishing time. $O(n \log n)$ time
2. Now choosing least finishing time is $O(1)$
3. Keep track of the finishing time of the last request added to A. Then check if starting time of i later than that
4. Thus, checking non-overlapping is $O(1)$
5. Total time $O(n \log n + n) = O(n \log n)$

Comments

1. Interesting Exercise: smallest interval first picks at least half the optimum number of intervals.
2. All requests need not be known at the beginning. Such *online* algorithms are a subject of research.

Comments

1. Interesting Exercise: smallest interval first picks at least half the optimum number of intervals.
2. All requests need not be known at the beginning. Such *online* algorithms are a subject of research.

Comments

1. Interesting Exercise: smallest interval first picks at least half the optimum number of intervals.
2. All requests need not be known at the beginning. Such *online* algorithms are a subject of research.

12.3: Interval Partitioning

Scheduling all Requests

Input A set of lectures, with start and end times

Goal Find the minimum number of classrooms, needed to schedule all the lectures such two lectures do not occur at the same time in the same room.

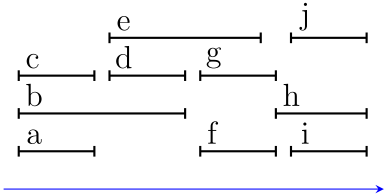


Figure: A schedule requiring 3 classrooms

Figure: A schedule requiring 4 classrooms

Scheduling all Requests

Input A set of lectures, with start and end times

Goal Find the minimum number of classrooms, needed to schedule all the lectures such two lectures do not occur at the same time in the same room.

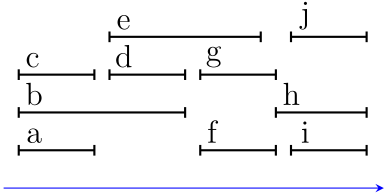


Figure: A schedule requiring 4 classrooms

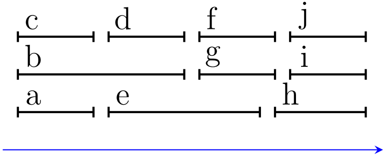


Figure: A schedule requiring 3 classrooms

Scheduling all Requests

Input A set of lectures, with start and end times

Goal Find the minimum number of classrooms, needed to schedule all the lectures such two lectures do not occur at the same time in the same room.

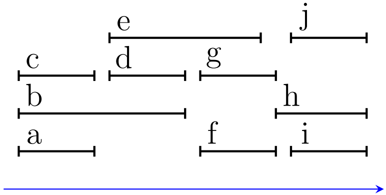


Figure: A schedule requiring 4 classrooms

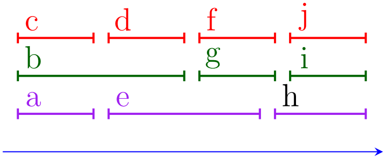


Figure: A schedule requiring 3 classrooms

Greedy Algorithm

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty do
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1
                and schedule lecture i in d + 1
        d = d + 1
```

What order should we process requests in? According to start times (breaking ties arbitrarily)

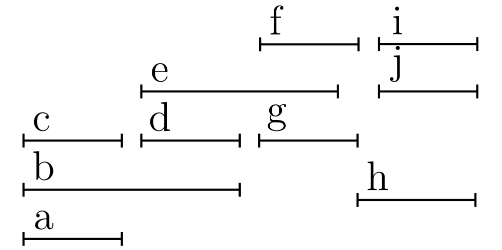
Greedy Algorithm

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty do
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1
                and schedule lecture i in d + 1
        d = d + 1
```

What order should we process requests in? According to start times (breaking ties arbitrarily)

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



Example of algorithm execution

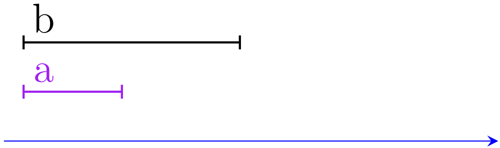
"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

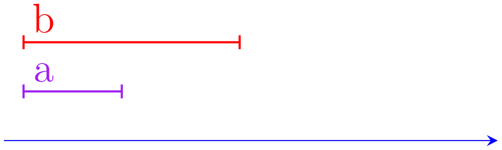
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



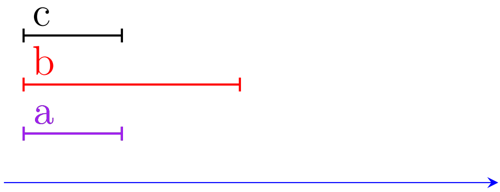
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



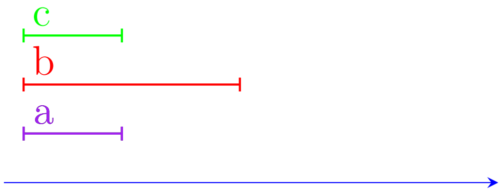
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



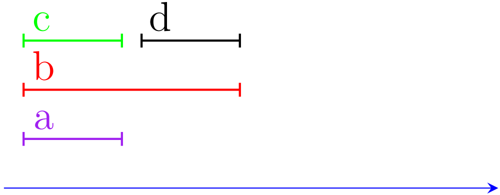
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



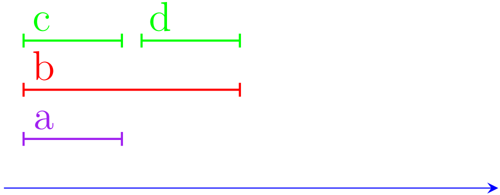
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



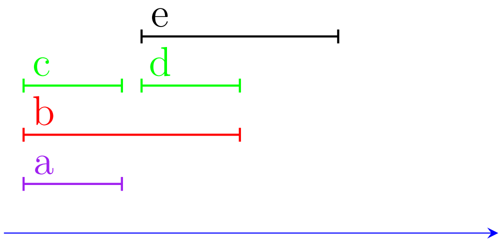
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



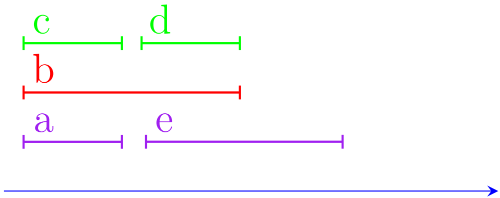
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



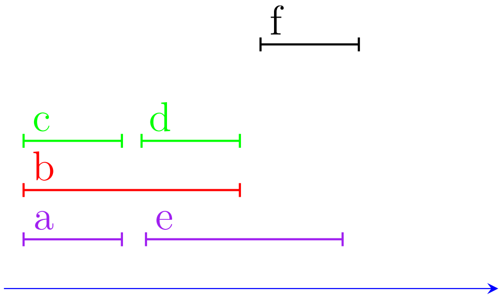
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



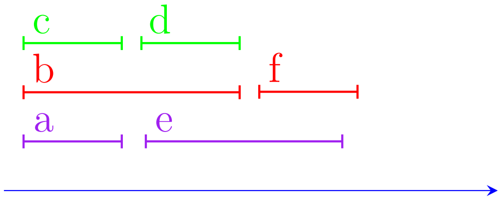
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



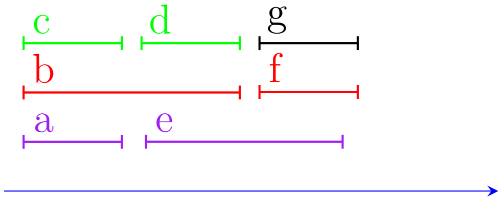
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



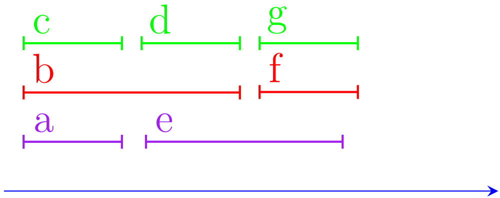
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



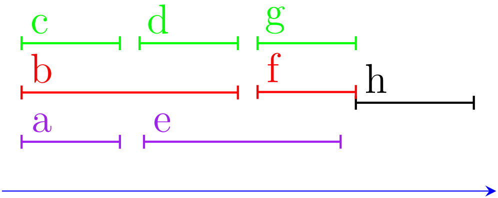
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



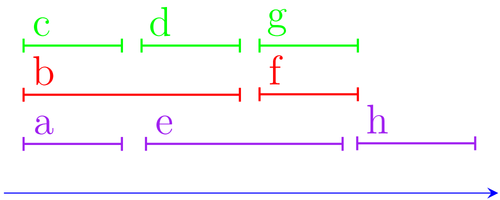
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



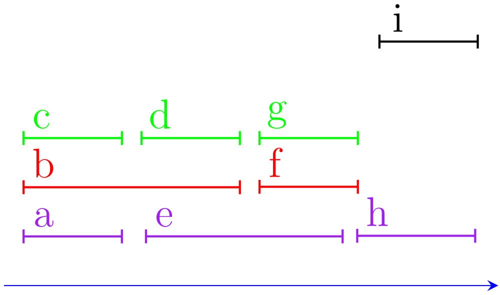
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



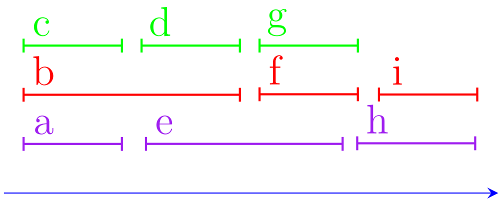
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



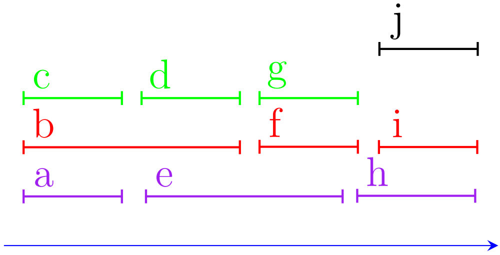
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



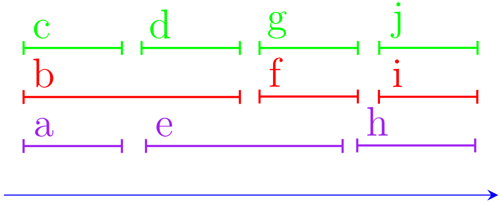
Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain



Depth of Lectures

Definition

1. For a set of lectures R, k are said to be in conflict if there is some time t such that there are k lectures going on at time t.
2. The depth of a set of lectures R is the maximum number of lectures in conflict at any time.

Depth of Lectures

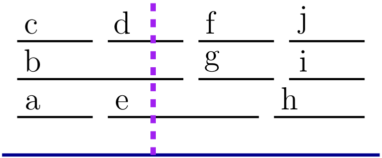
Definition

1. For a set of lectures R, k are said to be in conflict if there is some time t such that there are k lectures going on at time t.
2. The depth of a set of lectures R is the maximum number of lectures in conflict at any time.

Depth of Lectures

Definition

1. For a set of lectures R, k are said to be in conflict if there is some time t such that there are k lectures going on at time t.
2. The depth of a set of lectures R is the maximum number of lectures in conflict at any time.



Depth and Number of Class-rooms

Lemma

For any set R of lectures, the number of class-rooms required is at least the depth of R.

Proof.

All lectures that are in conflict must be scheduled in different rooms. $\square$

Depth and Number of Class-rooms

Lemma

For any set R of lectures, the number of class-rooms required is at least the depth of R.

Proof.

All lectures that are in conflict must be scheduled in different rooms. □

Number of Class-rooms used by Greedy Algorithm

Lemma

Let d be the depth of the set of lectures R. The number of class-rooms used by the greedy algorithm is d.

Proof.

1. Suppose the greedy algorithm uses more that d rooms. Let j be the first lecture that is scheduled in room $d + 1$.
2. Since we process lectures according to start times, there are d lectures that start (at or) before j and which conflict with j.
3. Thus, *at the start time of j*, there are at least $d + 1$ lectures in conflict, which contradicts the fact that the depth is d. ∎

Number of Class-rooms used by Greedy Algorithm

Lemma

Let d be the depth of the set of lectures R. The number of class-rooms used by the greedy algorithm is d.

Proof.

1. Suppose the greedy algorithm uses more that d rooms. Let j be the first lecture that is scheduled in room $d + 1$.
2. Since we process lectures according to start times, there are d lectures that start (at or) before j and which conflict with j.
3. Thus, *at the start time of j*, there are at least $d + 1$ lectures in conflict, which contradicts the fact that the depth is d. □

Number of Class-rooms used by Greedy Algorithm

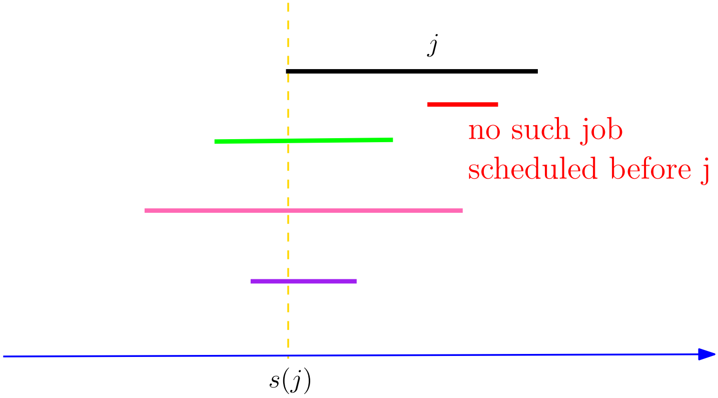
Lemma

Let d be the depth of the set of lectures R. The number of class-rooms used by the greedy algorithm is d.

Proof.

1. Suppose the greedy algorithm uses more that d rooms. Let j be the first lecture that is scheduled in room $d + 1$.
2. Since we process lectures according to start times, there are d lectures that start (at or) before j and which conflict with j.
3. Thus, *at the start time of j*, there are at least $d + 1$ lectures in conflict, which contradicts the fact that the depth is d. $\square$

Figure



Correctness

Observation

The greedy algorithm does not schedule two overlapping lectures in the same room.

Theorem

The greedy algorithm is correct and uses the optimal number of class-rooms.

Implementation and Running Time

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1 and schedule lecture i i
        d = d + 1
```

1. Presort according to start times. Picking lecture with earliest start time can be done in $O(1)$ time.
2. Keep track of the finish time of last lecture in each room.
3.
4. Total time

Implementation and Running Time

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1 and schedule lecture i i
        d = d + 1
```

1. Presort according to start times. Picking lecture with earliest start time can be done in $O(1)$ time.
2. Keep track of the finish time of last lecture in each room.
3.
4. Total time

Implementation and Running Time

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1 and schedule lecture i i
        d = d + 1
```

1. Presort according to start times. Picking lecture with earliest start time can be done in $O(1)$ time.
2. Keep track of the finish time of last lecture in each room.
3.
4. Total time

Implementation and Running Time

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1 and schedule lecture i i
        d = d + 1
```

1. Presort according to start times. Picking lecture with earliest start time can be done in $O(1)$ time.
2. Keep track of the finish time of last lecture in each room.
3. Checking conflict takes $O(d)$ time.
4. Total time $= O(n \log n + nd)$

Implementation and Running Time

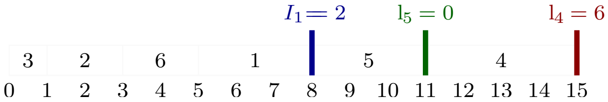
```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1 and schedule lecture i i
        d = d + 1
```

1. Presort according to start times. Picking lecture with earliest start time can be done in $O(1)$ time.
2. Keep track of the finish time of last lecture in each room.
3. With priority queues, checking conflict takes $O(\log d)$ time.
4. Total time $= O(n \log n + n \log d) = O(n \log n)$

Scheduling to Minimize Lateness

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time. d_i: deadline.
3. The lateness of a job is $l_i = \max(0, f_i - d_i)$.
4. Schedule all jobs such that $L = \max l_i$ is minimized.

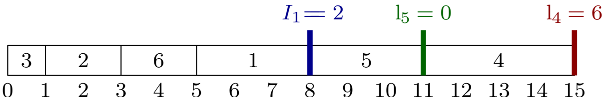
	1	2	3	4	5	6
t_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15



Scheduling to Minimize Lateness

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time. d_i: deadline.
3. The lateness of a job is $l_i = \max(0, f_i - d_i)$.
4. Schedule all jobs such that $L = \max l_i$ is minimized.

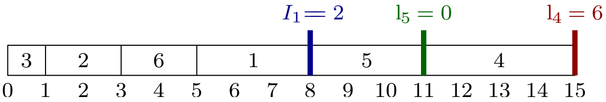
	1	2	3	4	5	6
t_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15



Scheduling to Minimize Lateness

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time. d_i: deadline.
3. The lateness of a job is $l_i = \max(0, f_i - d_i)$.
4. Schedule all jobs such that $L = \max l_i$ is minimized.

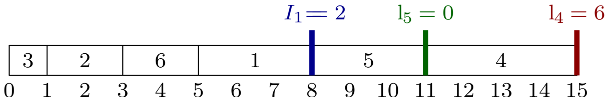
	1	2	3	4	5	6
t_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15



Scheduling to Minimize Lateness

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time. d_i: deadline.
3. The lateness of a job is $l_i = \max(0, f_i - d_i)$.
4. Schedule all jobs such that $L = \max l_i$ is minimized.

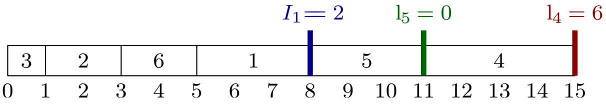
	1	2	3	4	5	6
t_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15



Scheduling to Minimize Lateness

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time. d_i: deadline.
3. The lateness of a job is $l_i = \max(0, f_i - d_i)$.
4. Schedule all jobs such that $L = \max l_i$ is minimized.

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



A Simpler Feasibility Problem

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time.
3. Schedule all jobs such that each of them completes before its deadline (in other words $L = \max_i l_i = 0$).

Definition

A schedule is feasible if all jobs finish before their deadline.

A Simpler Feasibility Problem

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time.
3. Schedule all jobs such that each of them completes before its deadline (in other words $L = \max_i l_i = 0$).

Definition

A schedule is feasible if all jobs finish before their deadline.

A Simpler Feasibility Problem

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time.
3. Schedule all jobs such that each of them completes before its deadline (in other words $L = \max_i l_i = 0$).

Definition

A schedule is feasible if all jobs finish before their deadline.

A Simpler Feasibility Problem

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time.
3. Schedule all jobs such that each of them completes before its deadline (in other words $L = \max_i l_i = 0$).

Definition

A schedule is feasible if all jobs finish before their deadline.

Greedy Template

```
Initially R is the set of all requests
curr_time = 0
while R is not empty do
    choose i ∈ R
    curr_time = curr_time + t_i
    if (curr_time > d_i) then
        return ``no feasible schedule''

return ``found feasible schedule''
```

Main task: Decide the order in which to process jobs in R

Greedy Template

```
Initially R is the set of all requests
curr_time = 0
while R is not empty do
    choose i ∈ R
    curr_time = curr_time + t_i
    if (curr_time > d_i) then
        return ``no feasible schedule''

return ``found feasible schedule''
```

Main task: Decide the order in which to process jobs in R

Three Algorithms

1. Shortest job first — sort according to t_i.
2. Shortest slack first — sort according to $d_i - t_i$.
3. EDF = Earliest deadline first — sort according to d_i.

Counter examples for first two: exercise

Three Algorithms

1. Shortest job first — sort according to t_i.
2. Shortest slack first — sort according to $d_i - t_i$.
3. EDF = Earliest deadline first — sort according to d_i.

Counter examples for first two: exercise

Three Algorithms

1. Shortest job first — sort according to t_i.
2. Shortest slack first — sort according to $d_i - t_i$.
3. EDF = Earliest deadline first — sort according to d_i.

Counter examples for first two: exercise

Three Algorithms

1. Shortest job first — sort according to t_i.
2. Shortest slack first — sort according to $d_i - t_i$.
3. EDF = Earliest deadline first — sort according to d_i.

Counter examples for first two: exercise

Three Algorithms

1. Shortest job first — sort according to t_i.
2. Shortest slack first — sort according to $d_i - t_i$.
3. EDF = Earliest deadline first — sort according to d_i.

Counter examples for first two: exercise

Earliest Deadline First

Theorem

Greedy with EDF *rule for picking requests correctly decides if there is a feasible schedule.*

Proof via an exchange argument.

Idle time: time during which machine is not working.

Lemma

If there is a feasible schedule then there is one with no idle time before all jobs are finished.

Earliest Deadline First

Theorem

Greedy with EDF *rule for picking requests correctly decides if there is a feasible schedule.*

Proof via an exchange argument.

Idle time: time during which machine is not working.

Lemma

If there is a feasible schedule then there is one with no idle time before all jobs are finished.

Earliest Deadline First

Theorem

Greedy with EDF *rule for picking requests correctly decides if there is a feasible schedule.*

Proof via an exchange argument.

Idle time: time during which machine is not working.

Lemma

If there is a feasible schedule then there is one with no idle time before all jobs are finished.

Earliest Deadline First

Theorem

Greedy with EDF *rule for picking requests correctly decides if there is a feasible schedule.*

Proof via an exchange argument.

Idle time: time during which machine is not working.

Lemma

If there is a feasible schedule then there is one with no idle time before all jobs are finished.

Inversions

Definition

A schedule S is said to have an inversion if there are jobs i and j such that S schedules i before j, but $d_i > d_j$.

Claim

If a schedule S has an inversion then there is an inversion between two adjacently *scheduled jobs.*

Proof: exercise.

Inversions

Definition

A schedule S is said to have an inversion if there are jobs i and j such that S schedules i before j, but $d_i > d_j$.

Claim

If a schedule S has an inversion then there is an inversion between two adjacently *scheduled jobs.*

Proof: exercise.

Main Lemma

Lemma

If there is a feasible schedule, then there is one with no inversions.

Proof Sketch.

Let S be a schedule with minimum number of inversions.

1. If S has 0 inversions, done.
2. Suppose S has one or more inversions. By claim there are two adjacent jobs i and j that define an inversion.
3. Swap positions of i and j.
4. New schedule is still feasible. (Why?)
5. New schedule has one fewer inversion — contradiction!

□

Main Lemma

Lemma

If there is a feasible schedule, then there is one with no inversions.

Proof Sketch.

Let S be a schedule with minimum number of inversions.

1. If S has $\mathbf{0}$ inversions, done.
2. Suppose S has one or more inversions. By claim there are two adjacent jobs i and j that define an inversion.
3. Swap positions of i and j.
4. New schedule is still feasible. (Why?)
5. New schedule has one fewer inversion — contradiction!

$\square$

Main Lemma

Lemma

If there is a feasible schedule, then there is one with no inversions.

Proof Sketch.

Let S be a schedule with minimum number of inversions.

1. If S has $\mathbf{0}$ inversions, done.
2. Suppose S has one or more inversions. By claim there are two adjacent jobs i and j that define an inversion.
3. Swap positions of i and j.
4. New schedule is still feasible. (Why?)
5. New schedule has one fewer inversion — contradiction!

Main Lemma

Lemma

If there is a feasible schedule, then there is one with no inversions.

Proof Sketch.

Let S be a schedule with minimum number of inversions.

1. If S has $\mathbf{0}$ inversions, done.
2. Suppose S has one or more inversions. By claim there are two adjacent jobs i and j that define an inversion.
3. Swap positions of i and j.
4. New schedule is still feasible. (Why?)
5. New schedule has one fewer inversion — contradiction!

Main Lemma

Lemma

If there is a feasible schedule, then there is one with no inversions.

Proof Sketch.

Let S be a schedule with minimum number of inversions.

1. If S has $\mathbf{0}$ inversions, done.
2. Suppose S has one or more inversions. By claim there are two adjacent jobs i and j that define an inversion.
3. Swap positions of i and j.
4. New schedule is still feasible. (Why?)
5. New schedule has one fewer inversion — contradiction!

Main Lemma

Lemma

If there is a feasible schedule, then there is one with no inversions.

Proof Sketch.

Let S be a schedule with minimum number of inversions.

1. If S has $\mathbf{0}$ inversions, done.
2. Suppose S has one or more inversions. By claim there are two adjacent jobs i and j that define an inversion.
3. Swap positions of i and j.
4. New schedule is still feasible. (Why?)
5. New schedule has one fewer inversion — contradiction!

$\square$

Back to Minimizing Lateness

Goal: schedule to minimize $L = \max_i l_i$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound L, can we check if there is a schedule such that $\max_i l_i \leq L$?

Yes! Set $d'_i = d_i + L$ for each job i. Use feasibility algorithm with new deadlines.

How can we find *minimum* L? Binary search!

Back to Minimizing Lateness

Goal: schedule to minimize $L = \max_i l_i$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound L, can we check if there is a schedule such that $\max_i l_i \leq L$?

Yes! Set $d'_i = d_i + L$ for each job i. Use feasibility algorithm with new deadlines.

How can we find *minimum* L? Binary search!

Back to Minimizing Lateness

Goal: schedule to minimize $L = \max_i l_i$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound L, can we check if there is a schedule such that $\max_i l_i \leq L$?

Yes! Set $d'_i = d_i + L$ for each job i. Use feasibility algorithm with new deadlines.

How can we find *minimum* L? Binary search!

Back to Minimizing Lateness

Goal: schedule to minimize $L = \max_i l_i$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound L, can we check if there is a schedule such that $\max_i l_i \leq L$?

Yes! Set $d'_i = d_i + L$ for each job i. Use feasibility algorithm with new deadlines.

How can we find *minimum* L? Binary search!

Back to Minimizing Lateness

Goal: schedule to minimize $L = \max_i l_i$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound L, can we check if there is a schedule such that $\max_i l_i \leq L$?

Yes! Set $d'_i = d_i + L$ for each job i. Use feasibility algorithm with new deadlines.

How can we find *minimum* L? Binary search!

Back to Minimizing Lateness

Goal: schedule to minimize $L = \max_i l_i$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound L, can we check if there is a schedule such that $\max_i l_i \leq L$?

Yes! Set $d'_i = d_i + L$ for each job i. Use feasibility algorithm with new deadlines.

How can we find *minimum* L? Binary search!

Binary search for finding minimum lateness

$L = L_{\min} = 0$
$L_{\max} = \sum_i t_i$ // *why is this sufficient?*
While $L_{\min} < L_{\max}$ do
    $L = \lfloor (L_{\max} + L_{\min})/2 \rfloor$
    check if there is a feasible schedule with lateness L
    if ''yes'' then $L_{\max} = L$
    else $L_{\min} = L + 1$
end while
return L

Running time: $O(n \log n \cdot \log T)$ where $T = \sum_i t_i$

1. $O(n \log n)$ for feasibility test (sort by deadlines)
2. $O(\log T)$ calls to feasibility test in binary search

Binary search for finding minimum lateness

```
L = L_min = 0
L_max = Σ_i t_i // why is this sufficient?
While L_min < L_max do
    L = ⌊(L_max + L_min)/2⌋
    check if there is a feasible schedule with lateness L
    if ``yes'' then L_max = L
    else L_min = L + 1
end while
return L
```

Running time: $O(n \log n \cdot \log T)$ where $T = \sum_i t_i$

1. $O(n \log n)$ for feasibility test (sort by deadlines)
2. $O(\log T)$ calls to feasibility test in binary search

Binary search for finding minimum lateness

```
L = L_min = 0
L_max = Σ_i t_i // why is this sufficient?
While L_min < L_max do
    L = ⌊(L_max + L_min)/2⌋
    check if there is a feasible schedule with lateness L
    if ``yes'' then L_max = L
    else L_min = L + 1
end while
return L
```

Running time: $O(n \log n \cdot \log T)$ where $T = \sum_i t_i$

1. $O(n \log n)$ for feasibility test (sort by deadlines)
2. $O(\log T)$ calls to feasibility test in binary search

Binary search for finding minimum lateness

$L = L_{\min} = 0$
$L_{\max} = \sum_i t_i$ // *why is this sufficient?*
While $L_{\min} < L_{\max}$ do
    $L = \lfloor (L_{\max} + L_{\min})/2 \rfloor$
    check if there is a feasible schedule with lateness L
    if ''yes'' then $L_{\max} = L$
    else $L_{\min} = L + 1$
end while
return L

Running time: $O(n \log n \cdot \log T)$ where $T = \sum_i t_i$

1. $O(n \log n)$ for feasibility test (sort by deadlines)
2. $O(\log T)$ calls to feasibility test in binary search

Do we need binary search?

What happens in each call?
EDF algorithm with deadlines $d'_i = d_i + L$.

Greedy with EDF schedules the jobs in the same order for all L!!!

Maybe there is a direct greedy algorithm for minimizing maximum lateness?

Do we need binary search?

What happens in each call?
EDF algorithm with deadlines $d'_i = d_i + L$.

Greedy with EDF schedules the jobs in the same order for all L!!!

Maybe there is a direct greedy algorithm for minimizing maximum lateness?

Do we need binary search?

What happens in each call?
EDF algorithm with deadlines $d'_i = d_i + L$.

Greedy with EDF schedules the jobs in the same order for all L!!!

Maybe there is a direct greedy algorithm for minimizing maximum lateness?

Greedy Algorithm for Minimizing Lateness

```
Initially R is the set of all requests
curr_time = 0
curr_late = 0
while R is not empty
    choose i ∈ R with earliest deadline
    curr_time = curr_time + t_i
    late = curr_time - d_i
    curr_late = max(late, curr_late)
return curr_late
```

Exercise: argue directly that above algorithm is correct

Can be easily implemented in $O(n \log n)$ time after sorting jobs.

Greedy Algorithm for Minimizing Lateness

```
Initially R is the set of all requests
curr_time = 0
curr_late = 0
while R is not empty
    choose i ∈ R with earliest deadline
    curr_time = curr_time + t_i
    late = curr_time − d_i
    curr_late = max(late, curr_late)
return curr_late
```

Exercise: argue directly that above algorithm is correct

Can be easily implemented in $O(n \log n)$ time after sorting jobs.

Greedy Algorithm for Minimizing Lateness

```
Initially R is the set of all requests
curr_time = 0
curr_late = 0
while R is not empty
    choose i ∈ R with earliest deadline
    curr_time = curr_time + t_i
    late = curr_time − d_i
    curr_late = max(late, curr_late)
return curr_late
```

Exercise: argue directly that above algorithm is correct

Can be easily implemented in $O(n \log n)$ time after sorting jobs.

Greedy Analysis: Overview

1. Greedy's first step leads to an optimum solution. Show that there is an optimum solution leading from the first step of Greedy and then use induction. Example, Interval Scheduling.
2. Greedy algorithm stays ahead. Show that after each step the solution of the greedy algorithm is at least as good as the solution of any other algorithm. Example, Interval scheduling.
3. Structural property of solution. Observe some structural bound of every solution to the problem, and show that greedy algorithm achieves this bound. Example, Interval Partitioning.
4. Exchange argument. Gradually transform any optimal solution to the one produced by the greedy algorithm, without hurting its optimality. Example, Minimizing lateness.

Greedy Analysis: Overview

1. Greedy's first step leads to an optimum solution. Show that there is an optimum solution leading from the first step of Greedy and then use induction. Example, Interval Scheduling.
2. Greedy algorithm stays ahead. Show that after each step the solution of the greedy algorithm is at least as good as the solution of any other algorithm. Example, Interval scheduling.
3. Structural property of solution. Observe some structural bound of every solution to the problem, and show that greedy algorithm achieves this bound. Example, Interval Partitioning.
4. Exchange argument. Gradually transform any optimal solution to the one produced by the greedy algorithm, without hurting its optimality. Example, Minimizing lateness.

Greedy Analysis: Overview

1. Greedy's first step leads to an optimum solution. Show that there is an optimum solution leading from the first step of Greedy and then use induction. Example, Interval Scheduling.
2. Greedy algorithm stays ahead. Show that after each step the solution of the greedy algorithm is at least as good as the solution of any other algorithm. Example, Interval scheduling.
3. Structural property of solution. Observe some structural bound of every solution to the problem, and show that greedy algorithm achieves this bound. Example, Interval Partitioning.
4. Exchange argument. Gradually transform any optimal solution to the one produced by the greedy algorithm, without hurting its optimality. Example, Minimizing lateness.

Greedy Analysis: Overview

1. Greedy's first step leads to an optimum solution. Show that there is an optimum solution leading from the first step of Greedy and then use induction. Example, Interval Scheduling.
2. Greedy algorithm stays ahead. Show that after each step the solution of the greedy algorithm is at least as good as the solution of any other algorithm. Example, Interval scheduling.
3. Structural property of solution. Observe some structural bound of every solution to the problem, and show that greedy algorithm achieves this bound. Example, Interval Partitioning.
4. Exchange argument. Gradually transform any optimal solution to the one produced by the greedy algorithm, without hurting its optimality. Example, Minimizing lateness.

Greedy Analysis: Overview

1. Greedy's first step leads to an optimum solution. Show that there is an optimum solution leading from the first step of Greedy and then use induction. Example, Interval Scheduling.
2. Greedy algorithm stays ahead. Show that after each step the solution of the greedy algorithm is at least as good as the solution of any other algorithm. Example, Interval scheduling.
3. Structural property of solution. Observe some structural bound of every solution to the problem, and show that greedy algorithm achieves this bound. Example, Interval Partitioning.
4. Exchange argument. Gradually transform any optimal solution to the one produced by the greedy algorithm, without hurting its optimality. Example, Minimizing lateness.

Takeaway Points

1. Greedy algorithms come naturally but often are incorrect. A proof of correctness is an absolute necessity.
2. *Exchange* arguments are often the key proof ingredient. Focus on why the first step of the algorithm is correct: need to show that there is an optimum/correct solution with the first step of the algorithm.
3. Thinking about correctness is also a good way to figure out which of the many greedy strategies is likely to work.

Takeaway Points

1. Greedy algorithms come naturally but often are incorrect. A proof of correctness is an absolute necessity.
2. *Exchange* arguments are often the key proof ingredient. Focus on why the first step of the algorithm is correct: need to show that there is an optimum/correct solution with the first step of the algorithm.
3. Thinking about correctness is also a good way to figure out which of the many greedy strategies is likely to work.

Takeaway Points

1. Greedy algorithms come naturally but often are incorrect. A proof of correctness is an absolute necessity.
2. *Exchange* arguments are often the key proof ingredient. Focus on why the first step of the algorithm is correct: need to show that there is an optimum/correct solution with the first step of the algorithm.
3. Thinking about correctness is also a good way to figure out which of the many greedy strategies is likely to work.

Takeaway Points

1. Greedy algorithms come naturally but often are incorrect. A proof of correctness is an absolute necessity.
2. *Exchange* arguments are often the key proof ingredient. Focus on why the first step of the algorithm is correct: need to show that there is an optimum/correct solution with the first step of the algorithm.
3. Thinking about correctness is also a good way to figure out which of the many greedy strategies is likely to work.

Notes

Notes

Notes

Notes