Outline

Greedy Algorithms

Greedy algorithms is another useful way for solving optimization problems.

Greedy Algorithms

Greedy algorithms is another useful way for solving optimization problems.

Optimization Problems

- For the given input, we are seeking solutions that must satisfy certain conditions.

Greedy Algorithms

Greedy algorithms is another useful way for solving optimization problems.

Optimization Problems

- For the given input, we are seeking solutions that must satisfy certain conditions.
- These solutions are called feasible solutions. (In general, there are many feasible solutions.)

Greedy algorithms is another useful way for solving optimization problems.

Optimization Problems

- For the given input, we are seeking solutions that must satisfy certain conditions.
- These solutions are called feasible solutions. (In general, there are many feasible solutions.)
- We have an optimization measure defined for each feasible solution.

Greedy algorithms is another useful way for solving optimization problems.

Optimization Problems

- For the given input, we are seeking solutions that must satisfy certain conditions.
- These solutions are called feasible solutions. (In general, there are many feasible solutions.)
- We have an optimization measure defined for each feasible solution.
- We are looking for a feasible solution that optimizes (either maximum or minimum) the optimization measure.

Examples

Matrix Chain Product Problem

- A feasible solution is any valid parenthesization of an n-term chain.

Examples

Matrix Chain Product Problem

- A feasible solution is any valid parenthesization of an n-term chain.
- The optimization measure is the total number of scalar multiplications for the parenthesization.

Examples

Matrix Chain Product Problem

- A feasible solution is any valid parenthesization of an n-term chain.
- The optimization measure is the total number of scalar multiplications for the parenthesization.
- Goal: Minimize the the total number of scalar multiplications.

Examples

Matrix Chain Product Problem

- A feasible solution is any valid parenthesization of an n-term chain.
- The optimization measure is the total number of scalar multiplications for the parenthesization.
- Goal: Minimize the the total number of scalar multiplications.

0/1 Knapsack Problem

- A feasible solution is any subset of items whose total weight is at most the knapsack capacity K.

Examples

Matrix Chain Product Problem

- A feasible solution is any valid parenthesization of an n-term chain.
- The optimization measure is the total number of scalar multiplications for the parenthesization.
- Goal: Minimize the the total number of scalar multiplications.

0/1 Knapsack Problem

- A feasible solution is any subset of items whose total weight is at most the knapsack capacity K.
- The optimization measure is the total item profit of the subset.

Examples

Matrix Chain Product Problem

- A feasible solution is any valid parenthesization of an n-term chain.
- The optimization measure is the total number of scalar multiplications for the parenthesization.
- Goal: Minimize the the total number of scalar multiplications.

0/1 Knapsack Problem

- A feasible solution is any subset of items whose total weight is at most the knapsack capacity K.
- The optimization measure is the total item profit of the subset.
- Goal: Maximize the the total profit.

General Description

- Given an optimization problem P, we seek an optimal solution.

Greedy Algorithms

General Description

- Given an optimization problem P, we seek an optimal solution.
- The solution is obtained by a sequence of steps.

Greedy Algorithms

General Description

- Given an optimization problem P, we seek an optimal solution.
- The solution is obtained by a sequence of steps.
- In each step, we select an “item” to be included into the solution.

General Description

- Given an optimization problem P, we seek an optimal solution.
- The solution is obtained by a sequence of steps.
- In each step, we select an "item" to be included into the solution.
- At each step, the decision is made based on the selections we have already made so far, that looks the best choice for achieving the optimization goal.

Greedy Algorithms

General Description

- Given an optimization problem P, we seek an optimal solution.
- The solution is obtained by a sequence of steps.
- In each step, we select an “item” to be included into the solution.
- At each step, the decision is made based on the selections we have already made so far, that looks the best choice for achieving the optimization goal.
- Once a selection is made, it cannot be undone: The selected item cannot be removed from the solution.

Minimum Spanning Tree (MST) Problem

This is a classical graph problem. We will study graph algorithms in detail later. Here we use MST as an example of Greedy Algorithms.

Minimum Spanning Tree (MST) Problem

This is a classical graph problem. We will study graph algorithms in detail later. Here we use MST as an example of Greedy Algorithms.

Definition

A tree is a connected graph with no cycles.

Minimum Spanning Tree (MST) Problem

This is a classical graph problem. We will study graph algorithms in detail later. Here we use MST as an example of Greedy Algorithms.

Definition

A tree is a connected graph with no cycles.

Definition

Let $G = (V, E)$ be a graph. A spanning tree of G is a subgraph of G that contains all vertices of G and is a tree.

Minimum Spanning Tree (MST) Problem

This is a classical graph problem. We will study graph algorithms in detail later. Here we use MST as an example of Greedy Algorithms.

Definition

A tree is a connected graph with no cycles.

Definition

Let $G = (V, E)$ be a graph. A spanning tree of G is a subgraph of G that contains all vertices of G and is a tree.

Minimum Spanning Tree (MST) Problem

Input: An connected undirected graph $G = (V, E)$. Each edge $e \in E$ has a weight $w(e) \geq 0$.
Find: a spanning tree T of G such that $w(T) = \sum_{e \in T} w(e)$ is minimum.

Kruskal's Algorithm

Kruskal's Algorithm

Sort the edges by non-decreasing weight. Let $e_1, e_2, \ldots, e_m$ be the sorted edge list
$T \leftarrow \emptyset$
for $i = 1$ **to** m **do**
    if $T \cup \{e_i\}$ does not contain a cycle **then**
        $T \leftarrow T \cup \{e_i\}$
    else
        do nothing
    end if
end for
output T

Kruskal's Algorithm

- The algorithm goes through a sequence of steps.

- The algorithm goes through a sequence of steps.
- At each step, we consider the edge e_i, and decide whether add e_i into T.

Kruskal's Algorithm

- The algorithm goes through a sequence of steps.
- At each step, we consider the edge e_i, and decide whether add e_i into T.
- Since we are building a spanning tree T, T can not contain any cycle. So if adding e_i into T introduces a cycle in T, we do not add it into T.

Kruskal's Algorithm

- The algorithm goes through a sequence of steps.
- At each step, we consider the edge e_i, and decide whether add e_i into T.
- Since we are building a spanning tree T, T can not contain any cycle. So if adding e_i into T introduces a cycle in T, we do not add it into T.
- Otherwise, we add e_i into T. We are processing the edges in the order of increasing edge weight. So when e_i is added into T, it looks the best to achieve the goal (minimum total weight).

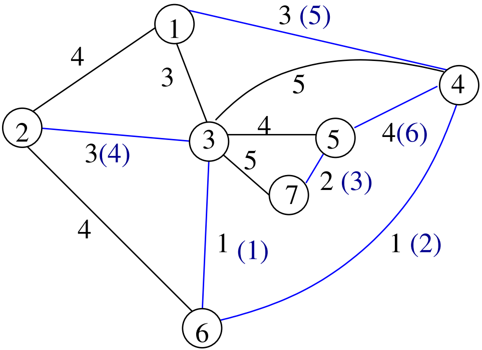
Kruskal's Algorithm

- The algorithm goes through a sequence of steps.
- At each step, we consider the edge e_i, and decide whether add e_i into T.
- Since we are building a spanning tree T, T can not contain any cycle. So if adding e_i into T introduces a cycle in T, we do not add it into T.
- Otherwise, we add e_i into T. We are processing the edges in the order of increasing edge weight. So when e_i is added into T, it looks the best to achieve the goal (minimum total weight).
- Once e_i is added, it is never removed and is included into the final tree T.

Kruskal's Algorithm

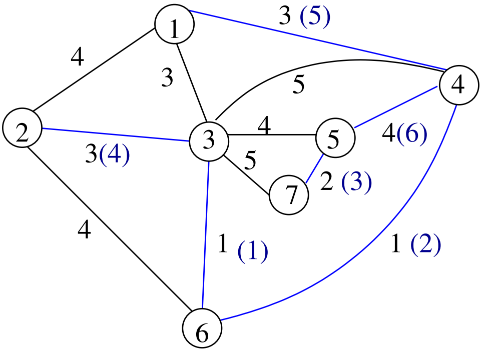
- The algorithm goes through a sequence of steps.
- At each step, we consider the edge e_i, and decide whether add e_i into T.
- Since we are building a spanning tree T, T can not contain any cycle. So if adding e_i into T introduces a cycle in T, we do not add it into T.
- Otherwise, we add e_i into T. We are processing the edges in the order of increasing edge weight. So when e_i is added into T, it looks the best to achieve the goal (minimum total weight).
- Once e_i is added, it is never removed and is included into the final tree T.
- This is a perfect example of greedy algorithms.

An Example



- The number near an edge is its weight. The blue edges are in the MST constructed by Kruskal's algorithm.

An Example



- The number near an edge is its weight. The blue edges are in the MST constructed by Kruskal's algorithm.
- The blue numbers in () indicate the order in which the edges are added into MST.

Kruskal's Algorithm

- For a given graph $G = (V, E)$, its MST is not unique. However, the weight of any two MSTs of G must be the same.

Kruskal's Algorithm

- For a given graph $G = (V, E)$, its MST is not unique. However, the weight of any two MSTs of G must be the same.
- In Kruskal's algorithm, two edges e_i and e_{i+1} may have the same weight. If we process e_{i+1} before e_i, we may get a different MST.

- For a given graph $G = (V, E)$, its MST is not unique. However, the weight of any two MSTs of G must be the same.
- In Kruskal's algorithm, two edges e_i and e_{i+1} may have the same weight. If we process e_{i+1} before e_i, we may get a different MST.
- Runtime of Kruskal's algorithm:
  - Sorting of edge list takes $\Theta(m \log m)$ time.

- For a given graph $G = (V, E)$, its MST is not unique. However, the weight of any two MSTs of G must be the same.
- In Kruskal's algorithm, two edges e_i and e_{i+1} may have the same weight. If we process e_{i+1} before e_i, we may get a different MST.
- Runtime of Kruskal's algorithm:
  - Sorting of edge list takes $\Theta(m \log m)$ time.
  - Then we process the edges one by one. So the loop iterates m time.

Kruskal's Algorithm

- For a given graph $G = (V, E)$, its MST is not unique. However, the weight of any two MSTs of G must be the same.
- In Kruskal's algorithm, two edges e_i and e_{i+1} may have the same weight. If we process e_{i+1} before e_i, we may get a different MST.
- Runtime of Kruskal's algorithm:
  - Sorting of edge list takes $\Theta(m \log m)$ time.
  - Then we process the edges one by one. So the loop iterates m time.
  - When processing an edge e_i, we check if $T \cup \{e_i\}$ contains a cycle or not. If not, add e_i into T. If yes, do nothing.

Kruskal's Algorithm

- For a given graph $G = (V, E)$, its MST is not unique. However, the weight of any two MSTs of G must be the same.
- In Kruskal's algorithm, two edges e_i and e_{i+1} may have the same weight. If we process e_{i+1} before e_i, we may get a different MST.
- Runtime of Kruskal's algorithm:
  - Sorting of edge list takes $\Theta(m \log m)$ time.
  - Then we process the edges one by one. So the loop iterates m time.
  - When processing an edge e_i, we check if $T \cup \{e_i\}$ contains a cycle or not. If not, add e_i into T. If yes, do nothing.
  - By using proper data structures, the processing of an edge e_i can be done in $O(\log n)$ time. (The detail was discussed in CSE250).

Kruskal's Algorithm

- For a given graph $G = (V, E)$, its MST is not unique. However, the weight of any two MSTs of G must be the same.
- In Kruskal's algorithm, two edges e_i and e_{i+1} may have the same weight. If we process e_{i+1} before e_i, we may get a different MST.
- Runtime of Kruskal's algorithm:
  - Sorting of edge list takes $\Theta(m \log m)$ time.
  - Then we process the edges one by one. So the loop iterates m time.
  - When processing an edge e_i, we check if $T \cup \{e_i\}$ contains a cycle or not. If not, add e_i into T. If yes, do nothing.
  - By using proper data structures, the processing of an edge e_i can be done in $O(\log n)$ time. (The detail was discussed in CSE250).
  - So the loop takes $O(m \log n)$ time.

Kruskal's Algorithm

- For a given graph $G = (V, E)$, its MST is not unique. However, the weight of any two MSTs of G must be the same.
- In Kruskal's algorithm, two edges e_i and e_{i+1} may have the same weight. If we process e_{i+1} before e_i, we may get a different MST.
- Runtime of Kruskal's algorithm:
  - Sorting of edge list takes $\Theta(m \log m)$ time.
  - Then we process the edges one by one. So the loop iterates m time.
  - When processing an edge e_i, we check if $T \cup \{e_i\}$ contains a cycle or not. If not, add e_i into T. If yes, do nothing.
  - By using proper data structures, the processing of an edge e_i can be done in $O(\log n)$ time. (The detail was discussed in CSE250).
  - So the loop takes $O(m \log n)$ time.
  - Since G is connected, $m \geq n$. The total runtime is $\Theta(m \log m + m \log n) = \Theta(m \log m)$.

Outline

- Are we done?

- Are we done?
- No! A big task is not done yet: How do we know Kruskal's algorithm is correct?

Elements of Greedy Algorithms

- Are we done?
- No! A big task is not done yet: How do we know Kruskal's algorithm is correct?
- Namely, how do we know the tree constructed by Kruskal's algorithm is indeed a MST?

Elements of Greedy Algorithms

- Are we done?
- No! A big task is not done yet: How do we know Kruskal's algorithm is correct?
- Namely, how do we know the tree constructed by Kruskal's algorithm is indeed a MST?
- You may have convinced yourself that we are using an obvious strategy towards the optimization goal.

- Are we done?
- No! A big task is not done yet: How do we know Kruskal's algorithm is correct?
- Namely, how do we know the tree constructed by Kruskal's algorithm is indeed a MST?
- You may have convinced yourself that we are using an obvious strategy towards the optimization goal.
- In this case, we are lucky: our intuition is correct.

Elements of Greedy Algorithms

- Are we done?
- No! A big task is not done yet: How do we know Kruskal's algorithm is correct?
- Namely, how do we know the tree constructed by Kruskal's algorithm is indeed a MST?
- You may have convinced yourself that we are using an obvious strategy towards the optimization goal.
- In this case, we are lucky: our intuition is correct.
- But in other cases, the strategies that seem equally obvious may lead to wrong solutions.

Elements of Greedy Algorithms

- Are we done?
- No! A big task is not done yet: How do we know Kruskal's algorithm is correct?
- Namely, how do we know the tree constructed by Kruskal's algorithm is indeed a MST?
- You may have convinced yourself that we are using an obvious strategy towards the optimization goal.
- In this case, we are lucky: our intuition is correct.
- But in other cases, the strategies that seem equally obvious may lead to wrong solutions.
- In general, the correctness of a greedy algorithm requires proof.

Correctness Proof of Algorithms

- An algorithm A is correct, if it works on **all** inputs.

Correctness Proof of Algorithms

- An algorithm A is correct, if it works on **all** inputs.
- If A works on some inputs, but not on some other inputs, then A is incorrect.

Correctness Proof of Algorithms

- An algorithm A is correct, if it works on **all** inputs.
- If A works on some inputs, but not on some other inputs, then A is incorrect.
- To show A is correct, you must argue that for all inputs, A produces intended solution.

Correctness Proof of Algorithms

- An algorithm A is correct, if it works on **all** inputs.
- If A works on some inputs, but not on some other inputs, then A is incorrect.
- To show A is correct, you must argue that for all inputs, A produces intended solution.
- To show A is incorrect, you only need to give a counter example input I: You show that, for this particular input I, the output from A is not the intended solution.

Correctness Proof of Algorithms

- An algorithm A is correct, if it works on **all** inputs.
- If A works on some inputs, but not on some other inputs, then A is incorrect.
- To show A is correct, you must argue that for all inputs, A produces intended solution.
- To show A is incorrect, you only need to give a counter example input I: You show that, for this particular input I, the output from A is not the intended solution.
- Strictly speaking, all algorithms need correctness proof.

Correctness Proof of Algorithms

- An algorithm A is correct, if it works on **all** inputs.
- If A works on some inputs, but not on some other inputs, then A is incorrect.
- To show A is correct, you must argue that for all inputs, A produces intended solution.
- To show A is incorrect, you only need to give a counter example input I: You show that, for this particular input I, the output from A is not the intended solution.
- Strictly speaking, all algorithms need correctness proof.
- For DaC, it's often so straightforward that the correctness proof is unnecessary/omitted. (Example: MergeSort)

Correctness Proof of Algorithms

- An algorithm A is correct, if it works on **all** inputs.
- If A works on some inputs, but not on some other inputs, then A is incorrect.
- To show A is correct, you must argue that for all inputs, A produces intended solution.
- To show A is incorrect, you only need to give a counter example input I: You show that, for this particular input I, the output from A is not the intended solution.
- Strictly speaking, all algorithms need correctness proof.
- For DaC, it's often so straightforward that the correctness proof is unnecessary/omitted. (Example: MergeSort)
- For dynamic programming algorithms, the correctness proof is less obvious than the DaC algorithms. But in most time, it is quite easy to convince people (i.e. informal proof) the algorithm is correct.

Correctness Proof of Algorithms

- An algorithm A is correct, if it works on **all** inputs.
- If A works on some inputs, but not on some other inputs, then A is incorrect.
- To show A is correct, you must argue that for all inputs, A produces intended solution.
- To show A is incorrect, you only need to give a counter example input I: You show that, for this particular input I, the output from A is not the intended solution.
- Strictly speaking, all algorithms need correctness proof.
- For DaC, it's often so straightforward that the correctness proof is unnecessary/omitted. (Example: MergeSort)
- For dynamic programming algorithms, the correctness proof is less obvious than the DaC algorithms. But in most time, it is quite easy to convince people (i.e. informal proof) the algorithm is correct.
- For greedy algorithms, the correctness proof can be very tricky.

Elements of Greedy Algorithms

For a greedy strategy to work, it must have the following two properties.

Elements of Greedy Algorithms

For a greedy strategy to work, it must have the following two properties.

Optimal Substructure Property

An optimal solution of the problem contains within it the optimal solutions of subproblems.

Elements of Greedy Algorithms

For a greedy strategy to work, it must have the following two properties.

Optimal Substructure Property

An optimal solution of the problem contains within it the optimal solutions of subproblems.

- This is the same property required by the dynamic programming algorithms.

Greedy Choice Property

A global optimal solution can be obtained by making a locally optimal choice that seems the best toward the optimization goal when the choice is made. (Namely: The choice is made based on the choices we have already made, **not** based on the future choices we might make.)

- This property is harder to describe exactly.

Elements of Greedy Algorithms

For a greedy strategy to work, it must have the following two properties.

Optimal Substructure Property

An optimal solution of the problem contains within it the optimal solutions of subproblems.

- This is the same property required by the dynamic programming algorithms.

Greedy Choice Property

A global optimal solution can be obtained by making a locally optimal choice that seems the best toward the optimization goal when the choice is made. (Namely: The choice is made based on the choices we have already made, **not** based on the future choices we might make.)

- This property is harder to describe exactly.
- Best way to understand it is by examples.

Example

Optimal Substructure Property for MST

- Let $G = (V, E)$ be a connected graph with edge weight.

Optimal Substructure Property for MST

Example

Optimal Substructure Property for MST

- Let $G = (V, E)$ be a connected graph with edge weight.
- Let $e_1 = (x, y)$ be the edge with the smallest weigh. (Namely, e_1 is the first edge chosen by Kruskal's algorithm.)

Example

Optimal Substructure Property for MST

- Let $G = (V, E)$ be a connected graph with edge weight.
- Let $e_1 = (x, y)$ be the edge with the smallest weigh. (Namely, e_1 is the first edge chosen by Kruskal's algorithm.)
- Let $G' = (V', E')$ be the graph obtained from G by merging x and y:

Example

Optimal Substructure Property for MST

- Let $G = (V, E)$ be a connected graph with edge weight.
- Let $e_1 = (x, y)$ be the edge with the smallest weigh. (Namely, e_1 is the first edge chosen by Kruskal's algorithm.)
- Let $G' = (V', E')$ be the graph obtained from G by merging x and y:
  - x and y becomes a single new vertex z in G'.
  - Namely $V' = V - \{x, y\} \cup \{z\}$

Example

Optimal Substructure Property for MST

- Let $G = (V, E)$ be a connected graph with edge weight.
- Let $e_1 = (x, y)$ be the edge with the smallest weigh. (Namely, e_1 is the first edge chosen by Kruskal's algorithm.)
- Let $G' = (V', E')$ be the graph obtained from G by merging x and y:
  - x and y becomes a single new vertex z in G'.
  - Namely $V' = V - \{x, y\} \cup \{z\}$
  - e_1 is deleted from G.

Example

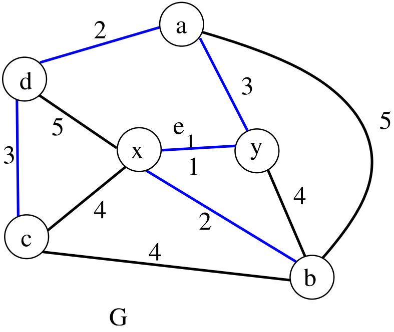
Optimal Substructure Property for MST

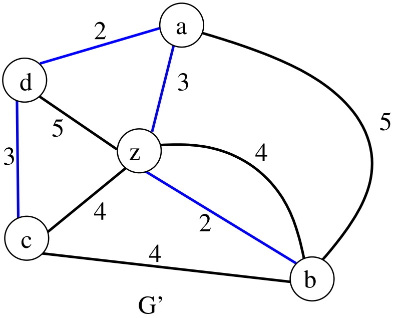
- Let $G = (V, E)$ be a connected graph with edge weight.
- Let $e_1 = (x, y)$ be the edge with the smallest weigh. (Namely, e_1 is the first edge chosen by Kruskal's algorithm.)
- Let $G' = (V', E')$ be the graph obtained from G by merging x and y:
  - x and y becomes a single new vertex z in G'.
  - Namely $V' = V - \{x, y\} \cup \{z\}$
  - e_1 is deleted from G.
  - Any edge e_i in G that was incident to x or y now is incident to z.

Example

Optimal Substructure Property for MST

- Let $G = (V, E)$ be a connected graph with edge weight.
- Let $e_1 = (x, y)$ be the edge with the smallest weigh. (Namely, e_1 is the first edge chosen by Kruskal's algorithm.)
- Let $G' = (V', E')$ be the graph obtained from G by merging x and y:
  - x and y becomes a single new vertex z in G'.
  - Namely $V' = V - \{x, y\} \cup \{z\}$
  - e_1 is deleted from G.
  - Any edge e_i in G that was incident to x or y now is incident to z.
  - The edge weights remain unchanged.





Optimal Substructure Property for MST

If T is a MST of G containing e_1, then $T' = T - \{e_1\}$ is a MST of G'.

Outline

Greedy Choice Property for Kruskal's Algorithm

- Let $e_1, e_2, \ldots, e_m$ be the edge list in the order of increasing weight. So e_1 is the first edge chosen by Kruskal's algorithm.

Greedy Choice Property for Kruskal's Algorithm

- Let $e_1, e_2, \ldots, e_m$ be the edge list in the order of increasing weight. So e_1 is the first edge chosen by Kruskal's algorithm.
- Let T_{opt} be an MST of G. By definition, the total weight of T_{opt} is the minimum.

Greedy Choice Property for Kruskal's Algorithm

- Let $e_1, e_2, \ldots, e_m$ be the edge list in the order of increasing weight. So e_1 is the first edge chosen by Kruskal's algorithm.
- Let T_{opt} be an MST of G. By definition, the total weight of T_{opt} is the minimum.
- We want to show T_{opt} contains e_1.

Greedy Choice Property for Kruskal's Algorithm

- Let $e_1, e_2, \ldots, e_m$ be the edge list in the order of increasing weight. So e_1 is the first edge chosen by Kruskal's algorithm.
- Let T_{opt} be an MST of G. By definition, the total weight of T_{opt} is the minimum.
- We want to show T_{opt} contains e_1.
- But this is not always possible. Recall that the MST of G is not unique.

Greedy Choice Property for Kruskal's Algorithm

- Let $e_1, e_2, \ldots, e_m$ be the edge list in the order of increasing weight. So e_1 is the first edge chosen by Kruskal's algorithm.
- Let T_{opt} be an MST of G. By definition, the total weight of T_{opt} is the minimum.
- We want to show T_{opt} contains e_1.
- But this is not always possible. Recall that the MST of G is not unique.
- So we will do this: Starting from T_{opt}, we change T_{opt}, without increasing the weight in the process, to another MST T' that contains e_1.

Greedy Choice Property for Kruskal's Algorithm

- Let $e_1, e_2, \ldots, e_m$ be the edge list in the order of increasing weight. So e_1 is the first edge chosen by Kruskal's algorithm.
- Let T_{opt} be an MST of G. By definition, the total weight of T_{opt} is the minimum.
- We want to show T_{opt} contains e_1.
- But this is not always possible. Recall that the MST of G is not unique.
- So we will do this: Starting from T_{opt}, we change T_{opt}, without increasing the weight in the process, to another MST T' that contains e_1.
- If T_{opt} contains e_1, then we are done (lucky!)

Greedy Choice Property for Kruskal's Algorithm

- Suppose T_{opt} does not contain e_1.

Greedy Choice Property for Kruskal's Algorithm

- Suppose T_{opt} does not contain e_1.
- Consider the graph $H = T_{opt} \cup \{e_1\}$.

Greedy Choice Property for Kruskal's Algorithm

- Suppose T_{opt} does not contain e_1.
- Consider the graph $H = T_{opt} \cup \{e_1\}$.
- H contains a cycle C. Let $e_i \neq e_1$ be another edge on C.
- Let $T' = T_{opt} - \{e_i\} \cup \{e_1\}$.

Greedy Choice Property for Kruskal's Algorithm

- Suppose T_{opt} does not contain e_1.
- Consider the graph $H = T_{opt} \cup \{e_1\}$.
- H contains a cycle C. Let $e_i \neq e_1$ be another edge on C.
- Let $T' = T_{opt} - \{e_i\} \cup \{e_1\}$.
- Then T' is a spanning tree of G.

Greedy Choice Property for Kruskal's Algorithm

- Suppose T_{opt} does not contain e_1.
- Consider the graph $H = T_{opt} \cup \{e_1\}$.
- H contains a cycle C. Let $e_i \neq e_1$ be another edge on C.
- Let $T' = T_{opt} - \{e_i\} \cup \{e_1\}$.
- Then T' is a spanning tree of G.
- Since e_1 is the edge with the smallest weight, $w(e_1) \leq w(e_i)$.

Greedy Choice Property for Kruskal's Algorithm

- Suppose T_{opt} does not contain e_1.
- Consider the graph $H = T_{opt} \cup \{e_1\}$.
- H contains a cycle C. Let $e_i \neq e_1$ be another edge on C.
- Let $T' = T_{opt} - \{e_i\} \cup \{e_1\}$.
- Then T' is a spanning tree of G.
- Since e_1 is the edge with the smallest weight, $w(e_1) \leq w(e_i)$.
- Hence $w(T') = w(T_{opt}) - w(e_i) + w(e_1) \leq w(T_{opt})$.

Greedy Choice Property for Kruskal's Algorithm

- Suppose T_{opt} does not contain e_1.
- Consider the graph $H = T_{opt} \cup \{e_1\}$.
- H contains a cycle C. Let $e_i \neq e_1$ be another edge on C.
- Let $T' = T_{opt} - \{e_i\} \cup \{e_1\}$.
- Then T' is a spanning tree of G.
- Since e_1 is the edge with the smallest weight, $w(e_1) \leq w(e_i)$.
- Hence $w(T') = w(T_{opt}) - w(e_i) + w(e_1) \leq w(T_{opt})$.
- But T_{opt} is a MST!

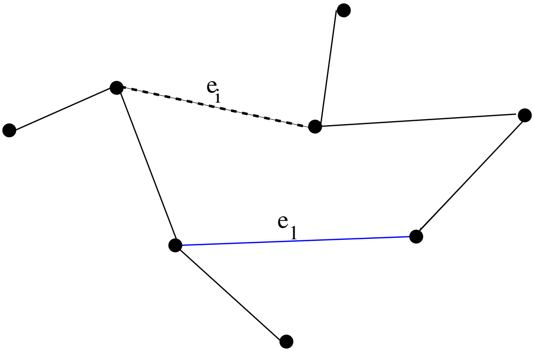
Greedy Choice Property for Kruskal's Algorithm

- Suppose T_{opt} does not contain e_1.
- Consider the graph $H = T_{opt} \cup \{e_1\}$.
- H contains a cycle C. Let $e_i \neq e_1$ be another edge on C.
- Let $T' = T_{opt} - \{e_i\} \cup \{e_1\}$.
- Then T' is a spanning tree of G.
- Since e_1 is the edge with the smallest weight, $w(e_1) \leq w(e_i)$.
- Hence $w(T') = w(T_{opt}) - w(e_i) + w(e_1) \leq w(T_{opt})$.
- But T_{opt} is a MST!
- So we must have $w(e_i) = w(e_1)$ and $w(T_{opt}) = w(T')$. In other words, both T_{opt} and T' are MSTs of G.

Greedy Choice Property for Kruskal's Algorithm

- Suppose T_{opt} does not contain e_1.
- Consider the graph $H = T_{opt} \cup \{e_1\}$.
- H contains a cycle C. Let $e_i \neq e_1$ be another edge on C.
- Let $T' = T_{opt} - \{e_i\} \cup \{e_1\}$.
- Then T' is a spanning tree of G.
- Since e_1 is the edge with the smallest weight, $w(e_1) \leq w(e_i)$.
- Hence $w(T') = w(T_{opt}) - w(e_i) + w(e_1) \leq w(T_{opt})$.
- But T_{opt} is a MST!
- So we must have $w(e_i) = w(e_1)$ and $w(T_{opt}) = w(T')$. In other words, both T_{opt} and T' are MSTs of G.
- This is what we want to show: There is an MST that contains e_1. So when Kruskal's algorithm includes e_1 into T, we are not making a mistake.

Greedy Choice Property for Kruskal's Algorithm



Correctness Proof of Kruskal's Algorithm

- The proof is by induction.

Correctness Proof of Kruskal's Algorithm

- The proof is by induction.
- Kruskal's algorithm selects the lightest edge $e_1 = (x, y)$.

Correctness Proof of Kruskal's Algorithm

- The proof is by induction.
- Kruskal's algorithm selects the lightest edge $e_1 = (x, y)$.
- By Greedy Choice Property, there exists an optimal MST of G that contains e_1.

Correctness Proof of Kruskal's Algorithm

- The proof is by induction.
- Kruskal's algorithm selects the lightest edge $e_1 = (x, y)$.
- By Greedy Choice Property, there exists an optimal MST of G that contains e_1.
- By induction hypothesis, Kruskal's algorithm construct a MST T' in the graph $G' = ((V - \{x, y\} \cup \{z\}), E')$ which is obtained from G by merging the two end vertices x, y of e_1.

Correctness Proof of Kruskal's Algorithm

- The proof is by induction.
- Kruskal's algorithm selects the lightest edge $e_1 = (x, y)$.
- By Greedy Choice Property, there exists an optimal MST of G that contains e_1.
- By induction hypothesis, Kruskal's algorithm construct a MST T' in the graph $G' = ((V - \{x, y\} \cup \{z\}), E')$ which is obtained from G by merging the two end vertices x, y of e_1.
- By the Optimal Substructure Property of MST, $T = T' \cup \{e_1\}$ is a MST of G.

Correctness Proof of Kruskal's Algorithm

- The proof is by induction.
- Kruskal's algorithm selects the lightest edge $e_1 = (x, y)$.
- By Greedy Choice Property, there exists an optimal MST of G that contains e_1.
- By induction hypothesis, Kruskal's algorithm construct a MST T' in the graph $G' = ((V - \{x, y\} \cup \{z\}), E')$ which is obtained from G by merging the two end vertices x, y of e_1.
- By the Optimal Substructure Property of MST, $T = T' \cup \{e_1\}$ is a MST of G.
- This T is the tree constructed by Kruskal's algorithm. Hence, Kruskal's algorithm indeed returns a MST.

Outline

We mentioned that some seemingly intuitive greedy strategies do not really work. Here is an example.

0/1 Knapsack Problem

We mentioned that some seemingly intuitive greedy strategies do not really work. Here is an example.

0/1 Knapsack Problem

Input: n item$_i$ $(1 \leq i \leq n)$. Each item$_i$ has an integer weight $w[i] \geq 0$ and a profit $p[i] \geq 0$.
A knapsack with an integer capacity K.
Find: A subset of items so that the total weight of the selected items is at most K, and the total profit is maximized.

0/1 Knapsack Problem

We mentioned that some seemingly intuitive greedy strategies do not really work. Here is an example.

0/1 Knapsack Problem

Input: n item$_i$ $(1 \le i \le n)$. Each item$_i$ has an integer weight $w[i] \ge 0$ and a profit $p[i] \ge 0$.
A knapsack with an integer capacity K.
Find: A subset of items so that the total weight of the selected items is at most K, and the total profit is maximized.

There are several greedy strategies that seem reasonable. But none of them works.

0/1 Knapsack Problem

Greedy Strategy 1

Since the goal is to maximize the profit without exceeding the capacity, we fill the items in the order of increasing weights. Namely:

- Sort the items by increasing item weight: $w[1] \leq w[2] \leq \cdots$.
- Fill the knapsack in the order item$_1$, item$_2$, ... until no more items can be put into the knapsack without exceeding the capacity.

0/1 Knapsack Problem

Greedy Strategy 1

Since the goal is to maximize the profit without exceeding the capacity, we fill the items in the order of increasing weights. Namely:

- Sort the items by increasing item weight: $w[1] \leq w[2] \leq \cdots$.
- Fill the knapsack in the order item$_1$, item$_2$, ... until no more items can be put into the knapsack without exceeding the capacity.

Counter Example:

$n = 2$, $w[1] = 2$, $w[2] = 4$, $p[1] = 2$, $p[2] = 3$, $K = 4$.

- This strategy puts item$_1$ into the knapsack with total profit 2.
- The optimal solution: put item$_2$ into the knapsack with total profit 3.

0/1 Knapsack Problem

- For this greedy strategy, we can still show the Optimal Substructure Property holds:
  - if S is an optimal solution, that contains the item$_1$, for the original input,

- For this greedy strategy, we can still show the Optimal Substructure Property holds:
  - if S is an optimal solution, that contains the item$_1$, for the original input,
  - then $S - \{\text{item}_1\}$ is an optimal solution for the input consisting of item$_2$, item$_3, \cdots$, item$_n$ and the knapsack with capacity $K - w[1]$.

0/1 Knapsack Problem

- For this greedy strategy, we can still show the Optimal Substructure Property holds:
  - if S is an optimal solution, that contains the item$_1$, for the original input,
  - then $S - \{\text{item}_1\}$ is an optimal solution for the input consisting of item$_2$, item$_3, \cdots$, item$_n$ and the knapsack with capacity $K - w[1]$.
- However, we cannot prove the Greedy Choice Property: We are not able to show there is an optimal solution that contains the item$_1$ (the lightest item).

0/1 Knapsack Problem

- For this greedy strategy, we can still show the Optimal Substructure Property holds:
  - if S is an optimal solution, that contains the item$_1$, for the original input,
  - then $S - \{\text{item}_1\}$ is an optimal solution for the input consisting of item$_2$, item$_3$, $\cdots$, item$_n$ and the knapsack with capacity $K - w[1]$.
- However, we cannot prove the Greedy Choice Property: We are not able to show there is an optimal solution that contains the item$_1$ (the lightest item).
- Without this property, there is no guarantee this strategy would work. (As the counter example has shown, it doesn't work.)

0/1 Knapsack Problem

Greedy Strategy 2

Since the goal is to maximize the profit without exceeding the capacity, we fill the items in the order of decreasing profits. Namely:

- Sort the items by decreasing item profit: $p[1] \geq p[2] \geq \cdots$.
- Fill the knapsack in the order item$_1$, item$_2$, ... until no more items can be put into the knapsack without exceeding the capacity.

0/1 Knapsack Problem

Greedy Strategy 2

Since the goal is to maximize the profit without exceeding the capacity, we fill the items in the order of decreasing profits. Namely:

- Sort the items by decreasing item profit: $p[1] \geq p[2] \geq \cdots$.
- Fill the knapsack in the order item$_1$, item$_2$, ... until no more items can be put into the knapsack without exceeding the capacity.

Counter Example:

$n = 3$, $p[1] = 3$, $p[2] = 2$, $p[3] = 2$, $w[1] = 3$, $w[2] = 2$, $w[3] = 2$, $K = 4$.

- This strategy puts item$_1$ into the knapsack with total profit 3.
- The optimal solution: put item$_2$ and item$_3$ into the knapsack with total profit 4.

0/1 Knapsack Problem

Greedy Strategy 3

Since the goal is to maximize the profit without exceeding the capacity, we fill the items in the order of decreasing unit profit. Namely:

- Sort the items by decreasing item unit profit: $\frac{p[1]}{w[1]} \geq \frac{p[2]}{w[2]} \geq \frac{p[3]}{w[1]} \cdots$
- Fill the knapsack in the order item$_1$, item$_2$, ... until no more items can be put into the knapsack without exceeding the capacity.

0/1 Knapsack Problem

Greedy Strategy 3

Since the goal is to maximize the profit without exceeding the capacity, we fill the items in the order of decreasing unit profit. Namely:

- Sort the items by decreasing item unit profit: $\frac{p[1]}{w[1]} \geq \frac{p[2]}{w[2]} \geq \frac{p[3]}{w[1]} \cdots$
- Fill the knapsack in the order item$_1$, item$_2$, ... until no more items can be put into the knapsack without exceeding the capacity.

Counter Example:

$n = 2$, $w[1] = 2$, $w[2] = 4$, $p[1] = 2$, $p[2] = 3$, $K = 4$.

- We have: $\frac{p[1]}{w[1]} = \frac{2}{2} = 1 \geq \frac{p[2]}{w[2]} = \frac{3}{4}$.
- This strategy puts item$_1$ into knapsack with total profit 2.
- The optimal solution: put item$_2$ into knapsack with total profit 3.

Fractional Knapsack Problem

Fractional Knapsack Problem

Input: n item$_i$ $(1 \leq i \leq n)$. Each item$_i$ has an integer weight $w[i] \geq 0$ and a profit $p[i] \geq 0$.
A knapsack with an integer capacity K.
Find: A subset of items to put into the knapsack. We can select a fraction of an item. The goal is the same: the total weight of the selected items is at most K, and the total profit is maximized.

Fractional Knapsack Problem

Fractional Knapsack Problem

Input: n item$_i$ $(1 \leq i \leq n)$. Each item$_i$ has an integer weight $w[i] \geq 0$ and a profit $p[i] \geq 0$.
A knapsack with an integer capacity K.
Find: A subset of items to put into the knapsack. We can select a fraction of an item. The goal is the same: the total weight of the selected items is at most K, and the total profit is maximized.

Mathematical description of Fractional Knapsack Problem

Input: $2n+1$ integers $p[1], p[2], \cdots, p[n], \;\; w[1], w[2], \cdots, w[n], \;\; K$
Find: a vector $(x_1, x_2, \ldots, x_n)$ such that:

- $0 \leq x_i \leq 1$ for $1 \leq i \leq n$
- $\sum_{i=1}^{n} x_i \cdot w[i] \leq K$
- $\sum_{i=1}^{n} x_i \cdot p[i]$ is maximized.

Fractional Knapsack Problem

- Although the Fractional Knapsack Problem looks very similar to the 0/1 Knapsack Problem, it is much much easier.

Fractional Knapsack Problem

- Although the Fractional Knapsack Problem looks very similar to the 0/1 Knapsack Problem, it is much much easier.
- The Greedy Strategy 3 works.

Fractional Knapsack Problem

- Although the Fractional Knapsack Problem looks very similar to the 0/1 Knapsack Problem, it is much much easier.
- The Greedy Strategy 3 works.

Greedy-Fractional-Knapsack

Sort the items by decreasing unit profit: $\frac{p[1]}{w[1]} \geq \frac{p[2]}{w[2]} \geq \frac{p[3]}{w[3]} \cdots$
$i = 1$
while $K > 0$ **do**
    if $K > w[i]$ **then**
        $x_i = 1$ **and** $K = K - w[i]$
    else
        $x_i = K/w[i]$ **and** $K = 0$
    end if
    $i = i + 1$
end while

Fractional Knapsack Problem

- Although the Fractional Knapsack Problem looks very similar to the 0/1 Knapsack Problem, it is much much easier.
- The Greedy Strategy 3 works.

Greedy-Fractional-Knapsack

```
Sort the items by decreasing unit profit: p[1]/w[1] ≥ p[2]/w[2] ≥ p[3]/w[3] ···
i = 1
while K > 0 do
    if K > w[i] then
        x_i = 1 and K = K − w[i]
    else
        x_i = K/w[i] and K = 0
    end if
    i = i + 1
end while
```

It can be shown the Greedy Choice Property holds in this case.

Outline

Activity Selection Problem

Activity Selection Problem

- A set $S = \{1, 2, \ldots, n\}$ of activities.

Activity Selection Problem

Activity Selection Problem

- A set $S = \{1, 2, \ldots, n\}$ of activities.
- Each activity i has a staring time s_i and a finishing time f_i ($s_i \leq f_i$).

Activity Selection Problem

- A set $S = \{1, 2, \dots, n\}$ of activities.
- Each activity i has a staring time s_i and a finishing time f_i ($s_i \le f_i$).
- Two activities i and j are compatible if the interval $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

Activity Selection Problem

Activity Selection Problem

- A set $S = \{1, 2, \ldots, n\}$ of activities.
- Each activity i has a staring time s_i and a finishing time f_i ($s_i \leq f_i$).
- Two activities i and j are compatible if the interval $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- Goal: Select a subset $A \subseteq S$ of mutually compatible activities so that $|A|$ is maximized.

Activity Selection Problem

Activity Selection Problem

- A set $S = \{1, 2, \dots, n\}$ of activities.
- Each activity i has a staring time s_i and a finishing time f_i ($s_i \leq f_i$).
- Two activities i and j are compatible if the interval $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- Goal: Select a subset $A \subseteq S$ of mutually compatible activities so that $|A|$ is maximized.

Application

- Consider a single CPU computer. It can run only one job at any time.

Activity Selection Problem

Activity Selection Problem

- A set $S = \{1, 2, \ldots, n\}$ of activities.
- Each activity i has a staring time s_i and a finishing time f_i ($s_i \leq f_i$).
- Two activities i and j are compatible if the interval $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- Goal: Select a subset $A \subseteq S$ of mutually compatible activities so that $|A|$ is maximized.

Application

- Consider a single CPU computer. It can run only one job at any time.
- Each activity i is a job to be run on the CPU that must start at time s_i and finish at time f_i.

Activity Selection Problem

- A set $S = \{1, 2, \ldots, n\}$ of activities.
- Each activity i has a staring time s_i and a finishing time f_i ($s_i \leq f_i$).
- Two activities i and j are compatible if the interval $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- Goal: Select a subset $A \subseteq S$ of mutually compatible activities so that $|A|$ is maximized.

Application

- Consider a single CPU computer. It can run only one job at any time.
- Each activity i is a job to be run on the CPU that must start at time s_i and finish at time f_i.
- How to select a maximum subset A of jobs to run on CPU?

Greedy Algorithm for Activity Selection Problem

Greedy Strategy

At any moment t, select the activity i with the smallest finish time f_i.

Greedy Algorithm for Activity Selection Problem

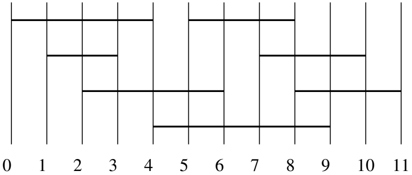
Greedy Strategy

At any moment t, select the activity i with the smallest finish time f_i.

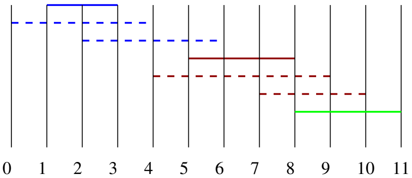
Greedy-Activity-Selection

Sort the activities by increasing finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$
$A = \{1\}$ (A is the set of activities to be selected.)
$j = 1$ (j is the current activity being considered.)
for $i = 2$ **to** n **do**
    if $s_i \geq f_j$ **then**
        $A = A \cup \{i\}$
        $j = i$
    end if
end for
return A

Example



Input

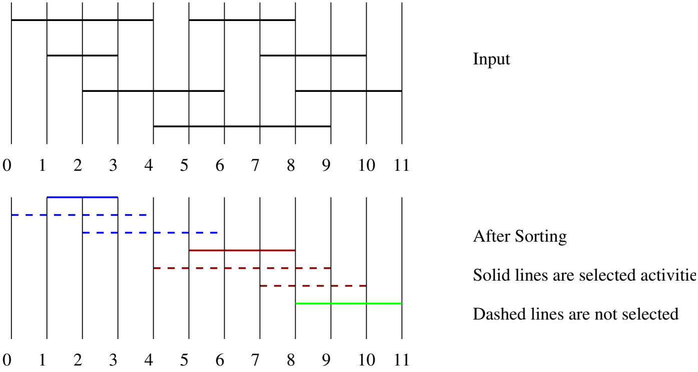


After Sorting

Solid lines are selected activitie

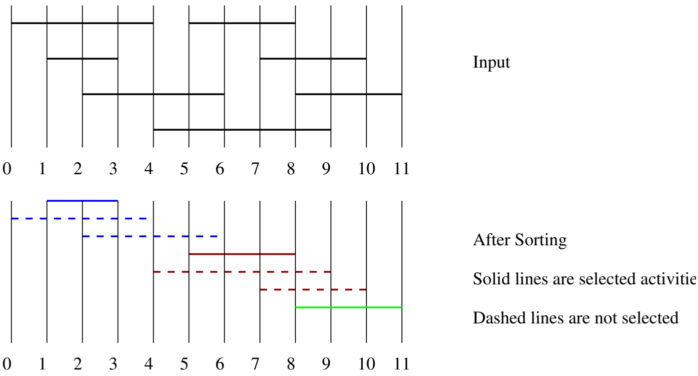
Dashed lines are not selected

Example



- $[1,3)$ is the first interval selected. The dashed intervals $[0,4)$ and $[2,6)$ are killed because they are not compatible with $[1,3)$.

Example



- $[1, 3)$ is the first interval selected. The dashed intervals $[0, 4)$ and $[2, 6)$ are killed because they are not compatible with $[1, 3)$.
- This problem is also called the interval scheduling problem.

Proof of Correctness

- Let $S = \{1, 2, \ldots, n\}$ be the set of activities to be selected. Assume $f_1 \le f_2 \le \cdots f_n$.

Proof of Correctness

- Let $S = \{1, 2, \ldots, n\}$ be the set of activities to be selected. Assume $f_1 \leq f_2 \leq \cdots f_n$.
- Let O be an optimal solution. Namely O is a subset of mutually compatible activities and $|O|$ is maximum.

Proof of Correctness

- Let $S = \{1, 2, \ldots, n\}$ be the set of activities to be selected. Assume $f_1 \leq f_2 \leq \cdots f_n$.
- Let O be an optimal solution. Namely O is a subset of mutually compatible activities and $|O|$ is maximum.
- Let X be the output from the Greedy algorithm. We always have $1 \in X$.

Proof of Correctness

- Let $S = \{1, 2, \ldots, n\}$ be the set of activities to be selected. Assume $f_1 \leq f_2 \leq \cdots f_n$.
- Let O be an optimal solution. Namely O is a subset of mutually compatible activities and $|O|$ is maximum.
- Let X be the output from the Greedy algorithm. We always have $1 \in X$.
- We want to show $|O| = |X|$. We will do this by induction on n.

Greedy Choice Property

- The activity 1 is selected by the greedy algorithm. We need to show there is an optimal solution that contains the activity 1.

Proof of Correctness

- Let $S = \{1, 2, \ldots, n\}$ be the set of activities to be selected. Assume $f_1 \leq f_2 \leq \cdots f_n$.
- Let O be an optimal solution. Namely O is a subset of mutually compatible activities and $|O|$ is maximum.
- Let X be the output from the Greedy algorithm. We always have $1 \in X$.
- We want to show $|O| = |X|$. We will do this by induction on n.

Greedy Choice Property

- The activity 1 is selected by the greedy algorithm. We need to show there is an optimal solution that contains the activity 1.
- If the optimal solution O contains 1, we are done.

Proof of Correctness

- Let $S = \{1, 2, \ldots, n\}$ be the set of activities to be selected. Assume $f_1 \leq f_2 \leq \cdots f_n$.
- Let O be an optimal solution. Namely O is a subset of mutually compatible activities and $|O|$ is maximum.
- Let X be the output from the Greedy algorithm. We always have $1 \in X$.
- We want to show $|O| = |X|$. We will do this by induction on n.

Greedy Choice Property

- The activity 1 is selected by the greedy algorithm. We need to show there is an optimal solution that contains the activity 1.
- If the optimal solution O contains 1, we are done.
- If not, let k be the first activity in O. Let $O' = O - \{k\} \cup \{1\}$.

Proof of Correctness

- Let $S = \{1, 2, \ldots, n\}$ be the set of activities to be selected. Assume $f_1 \leq f_2 \leq \cdots f_n$.
- Let O be an optimal solution. Namely O is a subset of mutually compatible activities and $|O|$ is maximum.
- Let X be the output from the Greedy algorithm. We always have $1 \in X$.
- We want to show $|O| = |X|$. We will do this by induction on n.

Greedy Choice Property

- The activity 1 is selected by the greedy algorithm. We need to show there is an optimal solution that contains the activity 1.
- If the optimal solution O contains 1, we are done.
- If not, let k be the first activity in O. Let $O' = O - \{k\} \cup \{1\}$.
- Since $f_1 \leq f_k$, all activities in O' are still mutually compatible.

Proof of Correctness

- Let $S = \{1, 2, \ldots, n\}$ be the set of activities to be selected. Assume $f_1 \leq f_2 \leq \cdots f_n$.
- Let O be an optimal solution. Namely O is a subset of mutually compatible activities and $|O|$ is maximum.
- Let X be the output from the Greedy algorithm. We always have $1 \in X$.
- We want to show $|O| = |X|$. We will do this by induction on n.

Greedy Choice Property

- The activity 1 is selected by the greedy algorithm. We need to show there is an optimal solution that contains the activity 1.
- If the optimal solution O contains 1, we are done.
- If not, let k be the first activity in O. Let $O' = O - \{k\} \cup \{1\}$.
- Since $f_1 \leq f_k$, all activities in O' are still mutually compatible.
- Clearly $|O| = |O'|$. So O' is an optimal solution containing 1.

Proof of Correctness

By the Greedy Choice Property, we may assume the optimal solution O contains the job 1.

Proof of Correctness

By the Greedy Choice Property, we may assume the optimal solution O contains the job 1.

Optimal Substructure Property

- Let $S_1 = \{i \in S \mid s_i \geq f_1\}$. ($S_1$ is the set of jobs that are compatible with job 1. Or equivalently, the set of jobs that are not killed by job 1.)

Proof of Correctness

By the Greedy Choice Property, we may assume the optimal solution O contains the job 1.

Optimal Substructure Property

- Let $S_1 = \{i \in S \mid s_i \geq f_1\}$. ($S_1$ is the set of jobs that are compatible with job 1. Or equivalently, the set of jobs that are not killed by job 1.)
- Let $O_1 = O - \{1\}$.

Proof of Correctness

By the Greedy Choice Property, we may assume the optimal solution O contains the job 1.

Optimal Substructure Property

- Let $S_1 = \{i \in S \mid s_i \geq f_1\}$. ($S_1$ is the set of jobs that are compatible with job 1. Or equivalently, the set of jobs that are not killed by job 1.)
- Let $O_1 = O - \{1\}$.
- Claim: O_1 is an optimal solution of the job set S_1.

By the Greedy Choice Property, we may assume the optimal solution O contains the job 1.

Optimal Substructure Property

- Let $S_1 = \{i \in S \mid s_i \geq f_1\}$. ($S_1$ is the set of jobs that are compatible with job 1. Or equivalently, the set of jobs that are not killed by job 1.)
- Let $O_1 = O - \{1\}$.
- Claim: O_1 is an optimal solution of the job set S_1.
    - If this is not true, let O_1' be an optimal solution set of S_1. Since O_1 is not optimal, we have $|O_1'| > |O_1|$.

Proof of Correctness

By the Greedy Choice Property, we may assume the optimal solution O contains the job 1.

Optimal Substructure Property

- Let $S_1 = \{i \in S \mid s_i \geq f_1\}$. ($S_1$ is the set of jobs that are compatible with job 1. Or equivalently, the set of jobs that are not killed by job 1.)
- Let $O_1 = O - \{1\}$.
- Claim: O_1 is an optimal solution of the job set S_1.
  - If this is not true, let O'_1 be an optimal solution set of S_1. Since O_1 is not optimal, we have $|O'_1| > |O_1|$.
  - Let $O' = O'_1 \cup \{1\}$. Then O' is a set of mutually compatible jobs in S, and $|O'| = |O'_1| + 1 > |O_1| + 1 = |O|$.

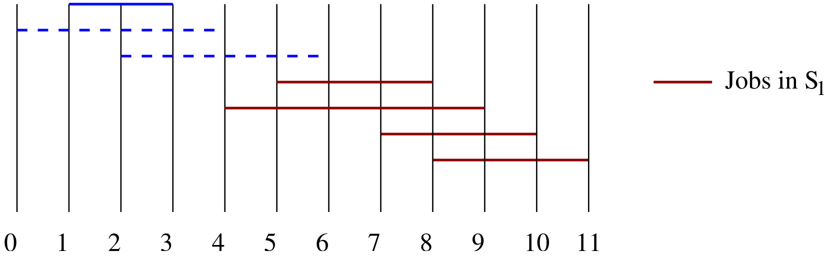
Proof of Correctness

By the Greedy Choice Property, we may assume the optimal solution O contains the job 1.

Optimal Substructure Property

- Let $S_1 = \{i \in S \mid s_i \geq f_1\}$. ($S_1$ is the set of jobs that are compatible with job 1. Or equivalently, the set of jobs that are not killed by job 1.)
- Let $O_1 = O - \{1\}$.
- Claim: O_1 is an optimal solution of the job set S_1.
  - If this is not true, let O'_1 be an optimal solution set of S_1. Since O_1 is not optimal, we have $|O'_1| > |O_1|$.
  - Let $O' = O'_1 \cup \{1\}$. Then O' is a set of mutually compatible jobs in S, and $|O'| = |O'_1| + 1 > |O_1| + 1 = |O|$.
  - But O is an optimal solution. This is a contradiction.
- Hence the claim is true.

Proof of Correctness



Proof of Correctness

- Since the Optimal Substructure and Greedy Choice properties are true, we can prove the correctness of the greedy algorithm by induction.

Proof of Correctness

- Since the Optimal Substructure and Greedy Choice properties are true, we can prove the correctness of the greedy algorithm by induction.
- Greedy algorithm picks the job 1 in its solution.

Proof of Correctness

- Since the Optimal Substructure and Greedy Choice properties are true, we can prove the correctness of the greedy algorithm by induction.
- Greedy algorithm picks the job 1 in its solution.
- By the Greedy Choice property, there is an optimal solution that also contains the job 1. So this selection needs not be reversed.

Proof of Correctness

- Since the Optimal Substructure and Greedy Choice properties are true, we can prove the correctness of the greedy algorithm by induction.
- Greedy algorithm picks the job 1 in its solution.
- By the Greedy Choice property, there is an optimal solution that also contains the job 1. So this selection needs not be reversed.
- The greedy algorithm delete all jobs that are incompatible with job 1. The remaining jobs is the set S_1 in the proof of Optimal Substructure property.

Proof of Correctness

- Since the Optimal Substructure and Greedy Choice properties are true, we can prove the correctness of the greedy algorithm by induction.
- Greedy algorithm picks the job 1 in its solution.
- By the Greedy Choice property, there is an optimal solution that also contains the job 1. So this selection needs not be reversed.
- The greedy algorithm delete all jobs that are incompatible with job 1. The remaining jobs is the set S_1 in the proof of Optimal Substructure property.
- By induction hypothesis, Greedy algorithm will output an optimal solution X_1 for S_1.

Proof of Correctness

- Since the Optimal Substructure and Greedy Choice properties are true, we can prove the correctness of the greedy algorithm by induction.
- Greedy algorithm picks the job 1 in its solution.
- By the Greedy Choice property, there is an optimal solution that also contains the job 1. So this selection needs not be reversed.
- The greedy algorithm delete all jobs that are incompatible with job 1. The remaining jobs is the set S_1 in the proof of Optimal Substructure property.
- By induction hypothesis, Greedy algorithm will output an optimal solution X_1 for S_1.
- By the Optimal Substructure property, $X = X_1 \cup \{1\}$ is an optimal solution of the original job set S.

Proof of Correctness

- Since the Optimal Substructure and Greedy Choice properties are true, we can prove the correctness of the greedy algorithm by induction.
- Greedy algorithm picks the job 1 in its solution.
- By the Greedy Choice property, there is an optimal solution that also contains the job 1. So this selection needs not be reversed.
- The greedy algorithm delete all jobs that are incompatible with job 1. The remaining jobs is the set S_1 in the proof of Optimal Substructure property.
- By induction hypothesis, Greedy algorithm will output an optimal solution X_1 for S_1.
- By the Optimal Substructure property, $X = X_1 \cup \{1\}$ is an optimal solution of the original job set S.
- X is the output from Greedy algorithm. So the algorithm is correct.

Proof of Correctness

- Since the Optimal Substructure and Greedy Choice properties are true, we can prove the correctness of the greedy algorithm by induction.
- Greedy algorithm picks the job 1 in its solution.
- By the Greedy Choice property, there is an optimal solution that also contains the job 1. So this selection needs not be reversed.
- The greedy algorithm delete all jobs that are incompatible with job 1. The remaining jobs is the set S_1 in the proof of Optimal Substructure property.
- By induction hypothesis, Greedy algorithm will output an optimal solution X_1 for S_1.
- By the Optimal Substructure property, $X = X_1 \cup \{1\}$ is an optimal solution of the original job set S.
- X is the output from Greedy algorithm. So the algorithm is correct.
- Runtime: Clearly $O(n \log n)$ (dominated by sorting).

Outline

Scheduling All Intervals

- Schedule all activities using as few resources as possible.

Scheduling All Intervals

- Schedule all activities using as few resources as possible.
- **Input:**
  - A set $\mathcal{R} = \{I_1, \ldots, I_n\}$ of n requests/activities.

Scheduling All Intervals

- Schedule all activities using as few resources as possible.
- **Input:**
  - A set $\mathcal{R} = \{I_1, \ldots, I_n\}$ of n requests/activities.
  - Each I_i has a start time s_i and finish time f_i. (So each I_i is represented by an interval $[s_i, f_i)$).

Scheduling All Intervals

- Schedule all activities using as few resources as possible.
- **Input:**
  - A set $\mathcal{R} = \{I_1, \ldots, I_n\}$ of n requests/activities.
  - Each I_i has a start time s_i and finish time f_i. (So each I_i is represented by an interval $[s_i, f_i)$).
- **Output:** A partition of $\mathcal{R}$ into as few subsets as possible, so that the intervals in each subset are mutually compatible. (Namely, they do not overlap.)

Scheduling All Intervals

Application

- Each request I_i is a job to be run on a CPU.

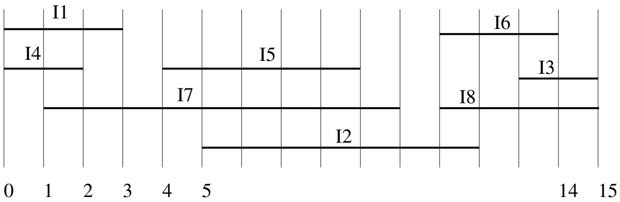
Scheduling All Intervals

Application

- Each request I_i is a job to be run on a CPU.
- If two intervals I_p and I_q overlap, they cannot run on the same CPU.

Application

- Each request I_i is a job to be run on a CPU.
- If two intervals I_p and I_q overlap, they cannot run on the same CPU.
- How to run all jobs using as few CPUs as possible?



Scheduling All Intervals

Another way to look at the problem:

Scheduling All Intervals

Another way to look at the problem:

- Color the intervals in $\mathcal{R}$ by different colors.

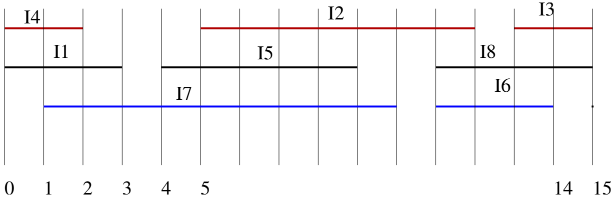
Scheduling All Intervals

Another way to look at the problem:

- Color the intervals in $\mathcal{R}$ by different colors.
- The intervals with the same color do not overlap.

Another way to look at the problem:

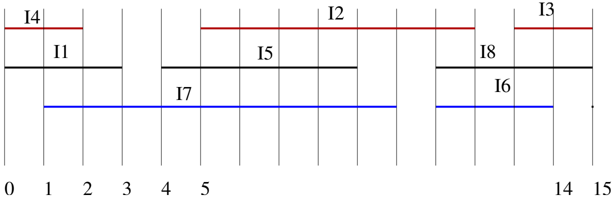
- Color the intervals in $\mathcal{R}$ by different colors.
- The intervals with the same color do not overlap.
- Using as few colors as possible.



Scheduling All Intervals

Another way to look at the problem:

- Color the intervals in $\mathcal{R}$ by different colors.
- The intervals with the same color do not overlap.
- Using as few colors as possible.



This problem is also known as Interval Graph Coloring Problem.

Scheduling All Intervals

Scheduling All Intervals

Graph Coloring

Let $G = (V, E)$ be an undirected graph.

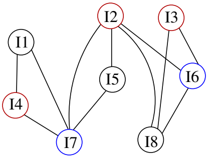
- A vertex coloring of G is an assignment of colors to the vertices of G so that no two vertices with the same color are adjacent to each other in G.

Scheduling All Intervals

Graph Coloring

Let $G = (V, E)$ be an undirected graph.

- A vertex coloring of G is an assignment of colors to the vertices of G so that no two vertices with the same color are adjacent to each other in G.
- Equivalently, a vertex coloring of G is a partition of V into vertex subsets so that no two vertices in the same subset are adjacent to each other.

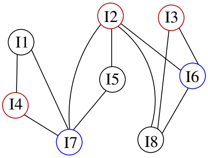


Scheduling All Intervals

Graph Coloring

Let $G = (V, E)$ be an undirected graph.

- A vertex coloring of G is an assignment of colors to the vertices of G so that no two vertices with the same color are adjacent to each other in G.
- Equivalently, a vertex coloring of G is a partition of V into vertex subsets so that no two vertices in the same subset are adjacent to each other.



A vertex coloring is also called just coloring of G. If G has a coloring with k colors, we say G is k-colorable.

Graph Coloring Problem

Input: An undirected graph $G = (V, E)$
Output: Find a vertex coloring of G using as few colors as possible.

Scheduling All Intervals

Graph Coloring Problem

Input: An undirected graph $G = (V, E)$
Output: Find a vertex coloring of G using as few colors as possible.

Chromatic Number

$$\chi(G) = \text{ the smallest } k \text{ such that } G \text{ is } k\text{-colorable}$$

Scheduling All Intervals

Graph Coloring Problem

Input: An undirected graph $G = (V, E)$
Output: Find a vertex coloring of G using as few colors as possible.

Chromatic Number

$$\chi(G) = \text{ the smallest } k \text{ such that } G \text{ is } k\text{-colorable}$$

- $\chi(G) = 1$ iff G has no edges.

Scheduling All Intervals

Graph Coloring Problem

Input: An undirected graph $G = (V, E)$
Output: Find a vertex coloring of G using as few colors as possible.

Chromatic Number

$$\chi(G) = \text{ the smallest } k \text{ such that } G \text{ is } k\text{-colorable}$$

- $\chi(G) = 1$ iff G has no edges.
- $\chi(G) = 2$ iff G is a bipartite graph with at least 1 edge.

Scheduling All Intervals

Graph Coloring Problem

Input: An undirected graph $G = (V, E)$
Output: Find a vertex coloring of G using as few colors as possible.

Chromatic Number

$$\chi(G) = \text{ the smallest } k \text{ such that } G \text{ is } k\text{-colorable}$$

- $\chi(G) = 1$ iff G has no edges.
- $\chi(G) = 2$ iff G is a bipartite graph with at least 1 edge.
- Graph Coloring is a very hard problem.

Scheduling All Intervals

Graph Coloring Problem

Input: An undirected graph $G = (V, E)$
Output: Find a vertex coloring of G using as few colors as possible.

Chromatic Number

$$\chi(G) = \text{ the smallest } k \text{ such that } G \text{ is } k\text{-colorable}$$

- $\chi(G) = 1$ iff G has no edges.
- $\chi(G) = 2$ iff G is a bipartite graph with at least 1 edge.
- Graph Coloring is a very hard problem.
- The problem can be solved in poly-time only for special graphs.

Four Color Theorem

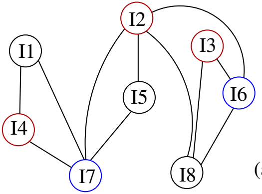
Every planar graph can be colored using at most 4 colors.

Scheduling All Intervals

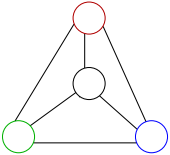
Four Color Theorem

Every planar graph can be colored using at most 4 colors.

G is a planar graph if it can be drawn on the plane so that no two edges cross.



(a)



(b)

Both graphs (a) and (b) are planar graphs. The graph (a) has a 3-coloring. The graph (b) requires 4 colors, because all 4 vertices are adjacent to each other, and hence each vertex must have a different color.

Scheduling All Intervals

Interval Graph

$G = (V, E)$ is called an interval graph if it can be represented as follows:

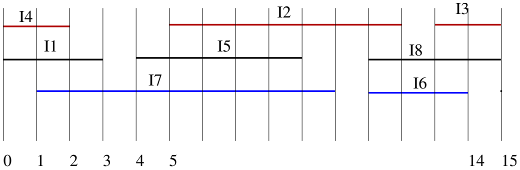
- Each vertex $p \in V$ represents an interval $[b_p, f_p)$.
- $(p, q) \in E$ if and only if the two intervals I_p and I_q overlap.

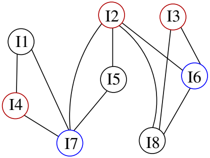
Scheduling All Intervals

Interval Graph

$G = (V, E)$ is called an interval graph if it can be represented as follows:

- Each vertex $p \in V$ represents an interval $[b_p, f_p)$.
- $(p, q) \in E$ if and only if the two intervals I_p and I_q overlap.





Scheduling All Intervals

- It is easy to see that the problem of scheduling all intervals is precisely the graph coloring problem for interval graphs.

Scheduling All Intervals

- It is easy to see that the problem of scheduling all intervals is precisely the graph coloring problem for interval graphs.
- We discuss a greedy algorithm for solving this problem.

Scheduling All Intervals

- It is easy to see that the problem of scheduling all intervals is precisely the graph coloring problem for interval graphs.
- We discuss a greedy algorithm for solving this problem.
- It is not easy to prove the greedy choice property for this greedy strategy.

Scheduling All Intervals

- It is easy to see that the problem of scheduling all intervals is precisely the graph coloring problem for interval graphs.
- We discuss a greedy algorithm for solving this problem.
- It is not easy to prove the greedy choice property for this greedy strategy.
- We show the correctness of the algorithm by other methods.

Scheduling All Intervals

- It is easy to see that the problem of scheduling all intervals is precisely the graph coloring problem for interval graphs.
- We discuss a greedy algorithm for solving this problem.
- It is not easy to prove the greedy choice property for this greedy strategy.
- We show the correctness of the algorithm by other methods.
- We use queues $Q_1, Q_2, \ldots$ to hold the subsets of intervals. (You can think that each Q_i is a CPU, and if an interval $I_p = [b_p, f_p)$ is put into Q_i, the job p is run on that CPU.)

Scheduling All Intervals

- It is easy to see that the problem of scheduling all intervals is precisely the graph coloring problem for interval graphs.
- We discuss a greedy algorithm for solving this problem.
- It is not easy to prove the greedy choice property for this greedy strategy.
- We show the correctness of the algorithm by other methods.
- We use queues $Q_1, Q_2, \ldots$ to hold the subsets of intervals. (You can think that each Q_i is a CPU, and if an interval $I_p = [b_p, f_p)$ is put into Q_i, the job p is run on that CPU.)
- Initially all queues are empty.

Scheduling All Intervals

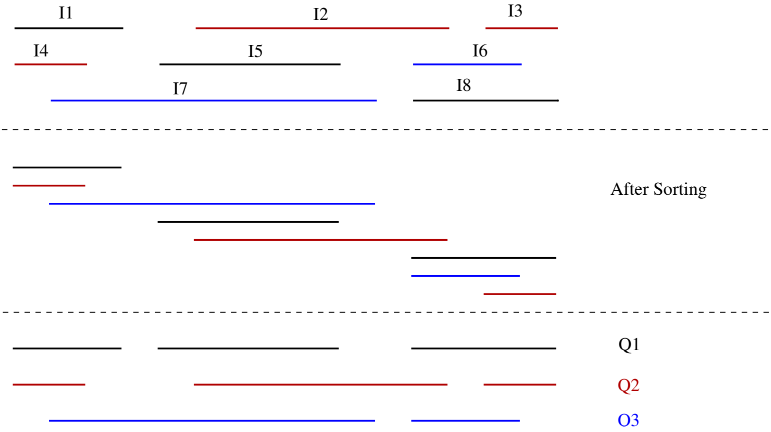
- It is easy to see that the problem of scheduling all intervals is precisely the graph coloring problem for interval graphs.
- We discuss a greedy algorithm for solving this problem.
- It is not easy to prove the greedy choice property for this greedy strategy.
- We show the correctness of the algorithm by other methods.
- We use queues $Q_1, Q_2, \ldots$ to hold the subsets of intervals. (You can think that each Q_i is a CPU, and if an interval $I_p = [b_p, f_p)$ is put into Q_i, the job p is run on that CPU.)
- Initially all queues are empty.
- When we consider an interval $[b_p, f_p)$ and a queue Q_i, we look at the last interval $[b_t, f_t)$ in Q_i. If $f_t \le b_p$, we say Q_i is available for $[b_p, f_p)$. (Meaning: the CPU Q_i has finished the last job assigned to it. So it is ready to run the job $[b_p, f_p)$.)

Scheduling All Intervals

Greedy-Schedule-All-Intervals

1. sort the intervals according to increasing b_p value: $b_1 \leq b_2 \leq \cdots \leq b_n$
2. $k = 0$ (k will be the number of queues we need.)
3. **for** $p = 1$ **to** n **do:**
4. look at $Q_1, Q_2, \ldots Q_k$, put $[b_p, f_p)$ into the first available Q_i.
5. if no current queue is available:
   - increase k by 1;
   - open a new empty queue;
   - put $[b_p, f_p)$ into this new queue.
6. **output** k and $Q_1, \ldots, Q_k$

Scheduling All Intervals



Scheduling All Intervals

Proof of correctness:

- We only put intervals into available queues. So each queue contains only non-overlapping intervals.

Proof of correctness:

- We only put intervals into available queues. So each queue contains only non-overlapping intervals.
- We need to show the algorithm uses minimum number of queues. (Namely, partition intervals into minimum number of subsets.)
  - If the input contains k mutually overlapping intervals, we must use at least k queues. (Because no two such intervals can be placed into the same queue.)

Scheduling All Intervals

Proof of correctness:

- We only put intervals into available queues. So each queue contains only non-overlapping intervals.
- We need to show the algorithm uses minimum number of queues. (Namely, partition intervals into minimum number of subsets.)
  - If the input contains k mutually overlapping intervals, we must use at least k queues. (Because no two such intervals can be placed into the same queue.)
  - When the algorithm opens a new empty queue Q_k for an interval $[b_p, f_p)$, none of the current queues $Q_1, \cdots, Q_{k-1}$ is available. This means that the last intervals in $Q_1, \cdots, Q_{k-1}$ all overlap with $[b_p, f_p)$. Hence the input contains k mutually overlapping intervals.

Proof of correctness:

- We only put intervals into available queues. So each queue contains only non-overlapping intervals.
- We need to show the algorithm uses minimum number of queues. (Namely, partition intervals into minimum number of subsets.)
  - If the input contains k mutually overlapping intervals, we must use at least k queues. (Because no two such intervals can be placed into the same queue.)
  - When the algorithm opens a new empty queue Q_k for an interval $[b_p, f_p)$, none of the current queues $Q_1, \cdots, Q_{k-1}$ is available. This means that the last intervals in $Q_1, \cdots, Q_{k-1}$ all overlap with $[b_p, f_p)$. Hence the input contains k mutually overlapping intervals.
  - The algorithm uses k queues. By the observation above, this is the smallest possible.

Runtime Analysis:

- Sorting takes $O(n \log n)$ time.
- The loop runs n times.
- The loop body scans $Q_1, \ldots, Q_k$ to find the first available queue. So it takes $O(k)$ time.
- Hence, the runtime is $\Theta(nk)$, (where k is the number of queues needed, or equivalently the chromatic number $\chi(G)$ of the input interval graph G.)

Scheduling All Intervals

Runtime Analysis:

- Sorting takes $O(n \log n)$ time.
- The loop runs n times.
- The loop body scans $Q_1, \ldots, Q_k$ to find the first available queue. So it takes $O(k)$ time.
- Hence, the runtime is $\Theta(nk)$, (where k is the number of queues needed, or equivalently the chromatic number $\chi(G)$ of the input interval graph G.)

In the worst case, k can be $\Theta(n)$. Hence, the worst case runtime is $\Theta(n^2)$.