CS 445
Analysis of algorithms 2
Greedy algorithms
Éric Schost
eschost@uwo.ca
Huffman encoding
(jpeg, MP3, ... )
Data and encodings

The need for encoding techniques:

- Computer see data as 0’s and 1’s.
- Texts, sounds, images are not given like this (letters / numbers)

We use the translation table

\[ A = 00, \ B = 01, \ C = 10, \ D = 11 \]

to turn \( AABACDAABBBABADABAACA \) into

\[ 00000100101100001010001001100010000100. \]

The table is called a code; the 00, 01, … are codewords. Two constraints:

- make it short,
- make it easy to decode.
Making it short and easy

Recipe:

• To make it short: the most common letters should have shorter encodings.

• To make it easy: no codeword should be prefix of another one.

Example:

• $A = 0, B = 00, \ldots$ bad, since you don’t know how to decode 00.

• $A = 0, B = 10, C = 110, D = 111$ is fine and gives the shorter encoding

$$001001101110100100110111010001100.$$
Tree representation

Using the rule

\[
0 \leftrightarrow \text{left} \quad 1 \leftrightarrow \text{right},
\]

you can turn your code into a binary tree.

Example with \(A = 0, B = 10, C = 110, D = 111\):

Conversely, a binary tree (with leaves labelled by \(A, B, \ldots\)) will give you a code. So how to build the tree?
Building the tree

We build the tree **bottom up**.

- Start by putting all letters, with their frequencies (number of times they appear in the message) into a set `ToDo`.

  \[
  \begin{array}{c}
  A \\
  B \\
  C \\
  D \\
  \end{array}
  \]

- Extend the frequency to trees: the frequency of a tree is the sum of the frequencies of the letters in it.

- Progressively build the tree by putting letters together. Always join the two trees with the lowest frequencies.
Using frequencies

Each letter has a frequency (an integer). Letters with large frequency should stay up in the tree.

• Extend the frequency to trees: the frequency of a tree is the sum of the frequencies of the letters in it.

• Always join the two trees with the lowest frequencies.
Example (frequencies in red)
Example (frequencies in red)
Example (frequencies in red)
Example (frequencies in red)
Optimality

Let

- $n_A$ be the frequency of $A =$ number of times $A$ appears in the message.
- $\ell_A$ be the length of the codeword for $A$.

Same for $n_B, \ldots$ and $\ell_B, \ldots$

Prop. The previous greedy algorithm computes a code which minimizes

$$n_A \ell_A + \cdots + n_Z \ell_Z$$

(assuming our letters are $A, \ldots, Z$), which is the length of the encoded message.
Today’s problems
Combinatorial optimization

Today: you are given a finite set $E$, and a weight (cost) function $w : E \to \mathbb{N}$.

$(E$ are some suitcases to put in the trunk of the car$)$

How to find a subset $F$ of $E$ such that

- $\sum_{f \in F} w(f)$ is maximal
  
  (maximizing the sum tells you how many suitcases you can put)

- subject to some condition on $F$.
  
  (with the condition that all suitcases fit in the trunk)
Greedy algorithms

Greedy algorithm:

- Sort the elements of $E$ by decreasing weight:

$$E = (e_1, \ldots, e_n) \text{ with } w(e_1) \geq \cdots \geq w(e_n).$$

When some weights are equal, find one (smart) way to order them.

- Initialize $F = \emptyset$.

- For $i = 1, \ldots, n$, add $i$ to $F$ if $F \cup \{e_i\}$ still satisfies the condition.

Features.

- Simple: don’t think globally, just try to put in as much stuff as you can!
- No guarantee to get the optimal.
- Usually, it is hard to prove correctness, and easy to prove un-correctness.
A resource allocation problem
A first try

A car rental company has the following requests for a given day:

\( R_1 \): 2pm to 8pm

\( R_2 \): 3pm to 4pm

\( R_3 \): 5pm to 6pm

You want to find a set of requests that is satisfiable and has maximal length. Here, the answer is \( F = (1) \), that stands for \( R_1 \).

Greedy algorithm:

- Weight function: \( w(R_i) = \text{end}(R_i) - \text{start}(R_i) = \text{length}(R_i) \).
- Sort the requests \( R_1, \ldots, R_n \) by decreasing length.
- Initialize \( F = \emptyset \).
- For \( i = 1, \ldots, n \), add \( i \) to \( F \) if the request \( R_i \) does not overlap with \( F \).

Correct algorithm?
A similar example

A car rental company has the following requests for a given day:

\[ R_1: \text{2pm to 8pm} \]

\[ R_2: \text{3pm to 4pm} \]

\[ R_3: \text{5pm to 6pm} \]

You want to find a set of requests that is satisfiable and has maximal cardinality.
Here, the answer is \( T = (2, 3) \), that stands for \( R_2, R_3 \).

Greedy algorithm:

- Weight function: \( w(R_i) = 1 \).
- Sort the requests \( R_1, \ldots, R_n \) by increasing end time.
- Initialize \( T = \emptyset \).
- For \( i = 1, \ldots, n \), add \( i \) to \( T \) if the request \( R_i \) does not overlap with \( T \).
Validity of the greedy algorithm

Let \( T = (x_1 < \cdots < x_p) \) be the output of the algorithm.

Let \( S = (y_1 < \cdots < y_q) \) be any satisfiable choice of requests.

Proof that \( p \geq q \).

- \( p \geq 1 \), of course.

- By definition of \( x_1 \), \( \text{end}(x_1) \leq \text{end}(y_1) \). So we can replace \( y_1 \) by \( x_1 \) in \( S \). We get \( S_1 = (x_1 < y_2 < \cdots < y_q) \), which is still satisfiable.

- Suppose \( q \geq 2 \). Since \( (x_1, y_2) \) is satisfiable, the algorithm didn’t stop at \( x_1 \). So \( p \geq 2 \).

- By definition of \( x_2 \), \( \text{end}(x_2) \leq \text{end}(y_2) \). So we can replace \( y_2 \) by \( x_2 \) in \( S_1 \). We get \( S_2 = (x_1 < x_2 < y_3 < \cdots < y_q) \), which is still satisfiable.

- Suppose \( q \geq 3 \). Since \( (x_1, x_2, y_3) \) is satisfiable, the algorithm didn’t stop at \( x_1, x_2 \). So \( p \geq 3 \).

- \( \ldots \)
What’s the conclusion?

In the previous examples, for

- a *same* set $E$ (the requests) and
- a *same* condition (no overlap),

the greedy algorithm may or may not work, depending on the weight function.

There are cases where the greedy algorithms will *always* work, no matter what the weight function is.

We will now see an example of that, computing maximum spanning trees.
Spanning trees on a graph
Graphs

Def. An **oriented** graph consists in a finite set of nodes (vertices) and a finite set of **oriented edges** that connect the nodes.

![Diagram of an oriented graph]

Def. A **symmetric** (non-oriented) graph consists in a finite set of nodes (vertices) and a finite set of **non-oriented edges** that connect the nodes.

![Diagram of a symmetric graph]

For the moment, we consider **symmetric** graphs. Most of the time, they are **connected** (all pairs of nodes can be connected).
Subgraphs

Given a graph $G$, we want to be able to describe some graphs it contains.

A subgraph of $G$ is obtained by selecting

- some of the vertices in $G$,
- and some of the edges that connect them.
Subgraphs

Given a graph $G$, we want to be able to describe some graphs it contains.

A subgraph of $G$ is obtained by selecting

- some of the vertices in $G$,
- and some of the edges that connect them.
Subgraphs

Given a graph $G$, we want to be able to describe some graphs it contains.

A subgraph of $G$ is obtained by selecting

- some of the vertices in $G$,
- and some of the edges that connect them.
Spanning trees

**Def.** A tree is a symmetric graph where any two vertices are connected by exactly one path. There is no loop.

A spanning tree is a way to connect all nodes in $G$, without loops.

**Def.** A spanning tree on a connected graph $G$ is a subgraph of $G$ which
- has exactly the same vertices as $G$,
- and is a tree.
Minimum / Maximum spanning tree

Suppose now that the edges of $G$ are weighted. How to find the spanning tree with minimal weight? or with maximal weight?

Applications in the design of networks, connections, ... 

• power / telephone lines between sites,
• wire connections on chip, ...

Problem. Prove that if you can solve maximum spanning tree problems, you can also solve minimum spanning trees problems.

Warning. The weights should stay $\geq 0$. 

Maximum spanning tree

Greedy algorithm:

- Sort the edges by decreasing weight, so that you can write them

  \[ e_1, \ldots, e_n \text{ with } w(e_1) \geq \cdots \geq w(e_n) \]

- Initialize \( E_T = \emptyset \)

- For \( i = 1, \ldots, n \), add the edge \( e_i \) to \( E_T \) if it creates no cycle in \( T \).

**Theorem** For any choice of a weight function \( w \), the greedy algorithm computes the maximum spanning tree.
Step 1: cardinality of a spanning tree

Prop. Let $T$ be a tree with $n$ nodes. Then $T$ has $n - 1$ edges.

Proof. Let $p$ be the number of edges. True when $T$ has 1 node, because $p = 0$.

Then, we do an induction, by cutting one leaf and its connecting edge.

This reduces the number of leaves and the number of edges by 1, and we still have a tree, so $(p - 1) = (n - 1) - 1$ and thus $p = n - 1$.

Corollary. Let $G$ be a symmetric, connected graph with $n$ nodes. Then any spanning tree on $G$ has $p = n - 1$ edges.
Step 2: augmentation of sets without loops

Prop. Let $G$ be a connected graph, and let $E$ be a subset of the edges of $G$. If $E$ has no loop and $|E| < p$, then one can find an edge $e$ not in $E$ such that $E \cup \{e\}$ still has no loop.

Proof. Let $S$ be the vertices contained in $E$. Then $G' = (S, E)$ is a subgraph of $G$.

Case 1: $G'$ is connected (it is a tree).

Then, $|S| = |E| + 1 < p + 1 = n$, so there is a vertex $v$ of $G$ not in $S$. Take for $x$ any edge containing $v$.

Case 2: $G'$ is not connected.

Take for $x$ any edge on a path that connects two components.
Validity of the greedy algorithm, part 1

Let $T = (x_1 > \cdots > x_r)$ be the output of the algorithm.

**Prop.** $T$ is a spanning tree, so $r = p$.

**Proof.** Of course, $T$ has no loop.

Suppose $T$ is not a spanning tree. Then, there exists an edge $e$ not in $T$, such that $T \cup \{e\}$ still has no loop.

**Case 1:** $e > x_1$. Impossible, since $x_1$ is the element with the largest weight.

**Case 2:** $x_i > e > x_{i+1}$. Impossible: at the moment we inserted $x_{i+1}$, we decided not to include $e$. This means that $e$ created a loop with $x_1, \ldots, x_i$.

**Case 3:** $x_r > e$. Impossible: we would have included it in $T$, since there is no loop in $T \cup \{e\}$. 
Step 3: exchanging edges

Prop. Let $T$ and $S$ be two spanning trees, and let $x$ be an edge in $T$ but not in $S$. Then there exists an edge $y$ in $S$ but not in $T$ such that $S - y + x$ is still a spanning tree.

Proof. Adding the edge $x$ to $S$.
Prop. Let $T$ and $S$ be two spanning trees, and let $x$ be an edge in $T$ but not in $S$. Then there exists an edge $y$ in $S$ but not in $T$ such that $S - y + x$ is still a spanning tree.

Proof. This creates a loop.
Step 3: exchanging edges

Prop. Let $T$ and $S$ be two spanning trees, and let $x$ be an edge in $T$ but not in $S$. Then there exists an edge $y$ in $S$ but not in $T$ such that $S - y + x$ is still a spanning tree.

Proof. One of the edges on the loop is not in $T$.

So we can remove it!
Validity of the greedy algorithm, part 2

Let $T = (x_1 > \cdots > x_p)$ be the tree given by the algorithm.

Let $S = (y_1 > \cdots > y_p)$ be any spanning tree.

Proof that $w(T) \geq w(S)$.

1. For all $i$, $x_1 \geq y_i$.
2. There exists $y_i$ in $S$ such that $S' = S - \{y_i\} + \{x_1\}$ is still a spanning tree. Then $w(S') \geq w(S)$, and $S'$ looks like $(x_1 > y_2' > \cdots > y_p')$.
3. For all $i$, $(x_1, y_i')$ is no cycle, so $x_2 \geq y_i'$.
4. There there exists $y_i'$ in $S$ such that $S'' = S' - \{y_i'\} + \{x_2\}$ is still a spanning tree. Then $w(S'') \geq w(S')$, and $S''$ looks like $(x_1 > x_2 > y_3'' > \cdots > y_p'')$.
5. For all $i$, $(x_1, x_2, y_i'')$ has no cycle, so $x_3 \geq y_i''$.
6. …
Conclusion

Summary: what are the properties we used?

- all spanning trees have the same size;
- the exchange property;
- a criterion: a set of edges is contained in a spanning tree if and only if it has no loops.

This was enough for us to prove the validity of the greedy algorithm.

Next step: we abstract these important features, to get ... matroids.
What about the complexity?

Assumptions:

• Cost model: all operations on integers take time $1$
• The graph is given by an array $[\text{edge}_i, \text{weight}_i]$.
• We have $v$ vertices and $e$ edges, with $e \leq v^2$.

Cost analysis

• Sorting the edges takes time $O(e \log(e)) = O(e \log(v))$.
• Then, each we examine a new edge, we need to check if it creates no cycle.

Naive check: each time, search through all the graph, $O(v)$.

Better: maintain a representation of all the sets-of-vertices that already have been connected.

Union-Find: $O(\alpha(v)) \ll O(\log(v))$.

Total: $O(e \log(v))$. 
Similar examples
Semi-matching

Consider an array of integers $w_{i,j} \geq 0$ (call them \textit{weights})

$$W = \begin{bmatrix}
4 & 6 & 4 & 5 \\
3 & 8 & 1 & 6 \\
2 & 9 & 2 & 10 \\
1 & 2 & 3 & 18
\end{bmatrix}.$$ 

\textbf{Semi-matching:} picking one element per row.

\textbf{Goal:} semi-matching of maximal weight.

(rows columns)

(the edges are weighted with the $w_{i,j}$)
Semi-matching

Consider an array of integers $w_{i,j} \geq 0$ (call them *weights*)

$$W = \begin{bmatrix}
4 & 6 & 4 & 5 \\
3 & 8 & 1 & 6 \\
2 & 9 & 2 & 10 \\
1 & 2 & 3 & 18
\end{bmatrix}.$$ 

**Semi-matching**: picking one element per row.

**Goal**: semi-matching of maximal weight.

(the edges are weighted with the $w_{i,j}$)
Similarities with the spanning tree problem

Basic correspondence

- Semi-matchings similar to spanning trees.
- Partial semi-matchings (when you haven’t picked one element for each row yet) similar to sets of edges without loops.

More subtle...

- All semi-matchings have the same size.
- The exchange property between semi-matchings is still here.

Conclusion: the greedy algorithm still works.
Another resource scheduling problem

You are given a list $T$ of tasks $t_1, \ldots, t_n$ to do. Each task takes time 1 and has:

- a date $d(t_i)$ (the deadline),
- a reward $w(t_i) \geq 0$ that you get if you can finish $t_i$ before (or on) the deadline.

Example:

<table>
<thead>
<tr>
<th>task</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>deadline</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>reward</td>
<td>30</td>
<td>20</td>
<td>50</td>
<td>60</td>
<td>10</td>
</tr>
</tbody>
</table>

Question: what tasks should you choose to get the maximal reward?

$\implies$ finding $\max_S \sum_{s \in S} w(s)$, for $S$ a satisfiable subset of $T$.

Answer: 3, 4, 5.
Similarities with the spanning tree problem

Basic correspondence

• Maximal satisfiable sets of constraints similar to spanning trees.
• Satisfiable sets of constraints similar to sets of edges without loops.

More subtle...

• All maximal sets of constraints have the same size.
• The exchange property between maximal sets of constraints is still here.

Conclusion: the greedy algorithm still works.
A problem of geometry

You are given vectors in the plane (not all in the same direction). Your goal:

• find a subset of vectors with maximal total length

• that are independent (you cannot express any of them as a combination of the other ones).
A problem of geometry

You are given vectors in the plane (not all in the same direction). Your goal:

- find a subset of vectors with maximal total length
- that are independent (you cannot express any of them as a combination of the other ones).
Similarities with the spanning tree problem

Basic correspondence

- Maximal sets of independent vectors similar to spanning trees.
- Sets of independent vectors similar to sets of edges without loops.

More subtle...

- All maximal sets of independent vectors have the same size.
- The exchange property between maximal sets of independent vectors is still here.

Conclusion: the greedy algorithm still works.
Matroids
General case

Recall the problem: you are given a finite set $E$, and a weight (cost) function $w : E \rightarrow \mathbb{N}$.

You are to find a subset $F$ of $E$ such that

- $\sum_{f \in F} w(f)$ is maximal
- subject to some independence condition on $F$.

Terminology

- Let $I$ be all subsets of $E$ that satisfy the condition (later on, we call them independents).
- Let $B$ be all maximal elements of $I$, in which you cannot add any element without breaking the condition (later on, we call them bases).
Summary of examples

Spanning trees

• $E$: edges in a graph.

• condition: having no loop.

• maximal sets: spanning trees.

Semi-matchings

• $E$: entries of an array.

• condition: no two elements the same row.

• maximal sets: semi-matchings.
Summary of examples

Tasks

- **E**: tasks.
- **condition**: being satisfiable.
- **maximal sets**: satisfiable family of tasks that cannot be augmented.

Vectors (in \( n \) dimensions)

- **E**: vectors.
- **condition**: no relation.
- **maximal sets**: families of \( n \) vectors with no relation.
Matroid

**Def.** Suppose that $B$ satisfies the following property:

**Exchange:** let $B, B'$ be in $B$ and let $x$ be in $B$ but not in $B'$. Then there exists $y$ in $B'$ but not in $B$ such that $B' - y + x$ is still in $B$.

Then $(E, B)$ is a matroid.

**Terminology.** When this is the case,

- the elements of $B$ are the bases;
- the elements of $I$ are the independents.
Summary of examples

Spanning trees

• **Independents**: families of edges without a loop.
• **Bases**: spanning trees.

Semi-matchings

• **Independents**: families with no two elements on the same row.
• **Bases**: semi-matchings.

Tasks

• **Independents**: satisfiable family of tasks.
• **Bases**: satisfiable family of tasks that cannot be augmented.

Vectors (in $n$ dimension)

• **Independents**: families of $\leq n$ vectors without relations.
• **Bases**: families of $n$ vectors without relations.
Properties

Prop. In a matroid, all bases have the same cardinality.

We saw that for spanning trees!

Proof. Let’s check that (for instance) $B = (x_1, x_2, x_3)$ and $C = (y_1, y_2)$, with all $x_i, y_j$ distinct, gives a problem.

1. $x_1$ is in $B$ but not in $C$, so there exist an element $y$ in $C$ but not in $B$ such that $C - \{y\} \cup \{x_1\}$ is a basis.
   
   Let’s say $y = y_1$. So $C' = (x_1, y_2)$ is a basis.

2. $x_2$ is in $B$ but not in $C'$, so there exist an element $y'$ in $C$ but not in $B$ such that $C' - \{y'\} \cup \{x_2\}$ is a basis.
   
   We cannot have $y' = x_1$, so $y' = y_2$. So $C'' = (x_1, x_2)$ is a basis.

3. $x_3$ is in $B$ but not in $C''$ … problem!
The greedy algorithm

Let $w$ be a weight function. We want to find $S$ such that

$$\sum_{s \in S} w(s)$$

is maximal, with $S$ independent.

Remark: it is the same thing as asking for $S$ such that

$$\sum_{s \in S} w(s)$$

is maximal, with $S$ a basis.

Greedy algorithm:

- Sort the elements of $E$ by decreasing weight:

  $$E = (e_1, \ldots, e_n) \quad \text{with} \quad w(e_1) \geq \cdots \geq w(e_n).$$

- Initialize $F = \emptyset$.

- For $i = 1, \ldots, n$, add $i$ to $F$ if $F \cup \{e_i\}$ is still independent.
The greedy algorithm

Let $w$ be a weight function. We want to find $S$ such that

$$\sum_{s \in S} w(s)$$

is maximal, with $S$ independent.

Remark: it is the same thing as asking for $S$ such that

$$\sum_{s \in S} w(s)$$

is maximal, with $S$ a basis.

**Theorem:**

- For any choice of $w$, the greedy algorithm finds the optimal.

**Proof:**

- Exactly the same as for spanning trees!
Why abstraction is good

1. Suppose that you have identified the families $\mathbf{B}$ and $\mathbf{I}$ such that:
   – you think that $\mathbf{B}$ are the bases of a matroid,
   – and you think that $\mathbf{I}$ are the independents of this matroid,

2. **Basic plan** to prove that the greedy algorithm works: does $\mathbf{B}$ satisfy the exchange property?
   
   Maybe not so easy.

3. **Plan B**: there are some general criteria on $\mathbf{I}$ that will work too.

   Abstraction is good because it gives general results.
How to prove you have a matroid

Prop. Suppose that:

1. If $I$ is in $\mathbf{I}$, all its subsets are.

2. For all $I, J$ in $\mathbf{I}$, if $|J| = |I| + 1$, then there exists $y$ in $J$, but not in $I$, such that $I \cup \{y\}$ is still in $\mathbf{I}$.

Then, you have a matroid.

Concretely. When trying to see if the greedy algorithm will work, you can

- try to identify what the bases should be, and prove the exchange property,
- or try the identify what the independents should be, and prove the previous property.

If one of these strategies works, then you know that the greedy algorithm will succeed, for any choice of the weights.
A second scheduling problem
Another resource scheduling problem

You are given a list $T$ of tasks $t_1, \ldots, t_n$ to do. Each task takes time 1 and has:

- a date $d(t_i)$ (the deadline),
- a reward $w(t_i) \geq 0$ that you get if you can finish $t_i$ before (or on) the deadline.

Example:

<table>
<thead>
<tr>
<th>task</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>deadline</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>reward</td>
<td>30</td>
<td>20</td>
<td>50</td>
<td>60</td>
<td>10</td>
</tr>
</tbody>
</table>

Question: what tasks should you choose to get the maximal reward?

$\Rightarrow$ finding $\max_S \sum_{s \in S} w(s)$, for $S$ a satisfiable subset of $T$.

Answer: 3, 4, 5.
What should we do?

We are going to prove that the greedy algorithm solves the problem.

- Sort the tasks $T$ by decreasing weight:

  $$T = (t_1, \ldots, t_n) \text{ with } w(t_1) \geq \cdots \geq w(t_n).$$

  When some weights are equal, find one (smart) way to order them.

- Initialize $S = \emptyset$.

- For $i = 1, \ldots, n$, add $i$ to $S$ if $S \cup \{e_i\}$ is still satisfiable.

To prove this, we will prove that the sets $B \subset T$ such that $B$ is satisfiable are the independents of a matroid (this is the “plan B” strategy).

If we can do it, the main theorem ensures that the greedy algorithm gives us a satisfiable set with maximal weight.
How to prove that we have a matroid

Setup.

- Let $I$ and $J$ be two satisfiable sets of tasks.
- Suppose that $|J| = |I| + 1$.

To do: find one task $y$ in $J$, but not in $I$, such that $I \cup \{y\}$ is still satisfiable.
The counting function

To set of tasks $S$, we associate the **counting function** $N(S, \cdot)$:

$$N(S, i) \text{ is the number of tasks } t \text{ in } S \text{ with } d(t) \leq i.$$ 

For the previous example, with $S = (1, 2, 3, 4, 5)$ (all tasks)
The counting function

To set of tasks $S$, we associate the counting function $N(S, \cdot)$:

$$N(S, i) \text{ is the number of tasks } t \text{ in } S \text{ with } d(t) \leq i.$$ 

For the previous example, with $S = (3, 4, 5)$ (optimal choice)
The counting function

To set of tasks $S$, we associate the counting function $N(S, .)$:

$$N(S, i)$$ is the number of tasks $t$ in $S$ with $d(t) \leq i$.

For the previous example, with $S = (3, 4, 5)$ (optimal choice)

Prop. $S$ is satisfiable if and only if $N(S, i) \leq i$ for all $i$. 
The counting function

To set of tasks $S$, we associate the counting function $N(S, .)$:

$$N(S, i) \text{ is the number of tasks } t \text{ in } S \text{ with } d(t) \leq i.$$ 

For the previous example, with $S = (1, 2, 3, 4, 5)$ (all tasks)

Prop. $S$ is satisfiable if and only if $N(S, i) \leq i$ for all $i$. 
The counting function

To set of tasks $S$, we associate the counting function $N(S,\cdot)$:

$$N(S, i)$$ is the number of tasks $t$ in $S$ with $d(t) \leq i$.

For the previous example, with $S = (3, 4, 5)$ (optimal choice)

Prop. $S$ is satisfiable if and only if $N(S, i) \leq i$ for all $i$. 
Proof

Step 1  \( N(J, i) = |J| \) and \( N(I, i) = |I| \) for \( i \) large enough.

Because of the definition.

Step 2  Let \( i_0 \) be the largest element such that \( N(J, i) \leq N(I, i) \).

This \( i_0 \) exists because of Step 1.

Step 3  So for all \( i > i_0 \), we have \( N(I, i) < N(J, i) \leq i \).

Because \( J \) is satisfiable.

Step 4  There exists one element \( y \) in \( J \) of deadline \( i_0 + 1 \) which is not in \( I \).

Because \( N(J, .) \) jumps over \( N(I, .) \) at \( i_0 + 1 \).

Step 5  After adding this \( y \) to \( I \), \( I \) is still satisfiable.

Because adding \( y \) to \( I \) increases \( N(I, i) \) by 1 for \( i > i_0 \), and we had some slack.