Topological Ordering Problem

**Input:** a directed acyclic graph (DAG) \( G = (V, E) \)

**Output:** 1-to-1 function \( \pi : V \rightarrow \{1, 2, 3 \ldots, n\} \), so that

- if \((u, v) \in E\) then \( \pi(u) < \pi(v) \)
Topological Ordering Problem

**Input:** a directed acyclic graph (DAG) $G = (V, E)$

**Output:** 1-to-1 function $\pi : V \rightarrow \{1, 2, 3 \cdots, n\}$, so that
- if $(u, v) \in E$ then $\pi(u) < \pi(v)$
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
**Topological Ordering**

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

![Graph Diagram]
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
**Topological Ordering**

- **Algorithm**: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?
Topological Ordering

Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

A:
- Use linked-lists of outgoing edges
- Maintain the in-degree $d_v$ of vertices
- Maintain a queue (or stack) of vertices $v$ with $d_v = 0$
**topological-sort**($G$)

1: let $d_v \leftarrow 0$ for every $v \in V$
2: **for** every $v \in V$ **do**
3:  **for** every $u$ such that $(v, u) \in E$ **do**
4:  $d_u \leftarrow d_u + 1$
5:  $S \leftarrow \{v : d_v = 0\}$, $i \leftarrow 0$
6:  **while** $S \neq \emptyset$ **do**
7:   $v \leftarrow$ arbitrary vertex in $S$, $S \leftarrow S \setminus \{v\}$
8:   $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
9:   **for** every $u$ such that $(v, u) \in E$ **do**
10:    $d_u \leftarrow d_u - 1$
11:   **if** $d_u = 0$ **then** add $u$ to $S$
12: **if** $i < n$ **then** output “not a DAG”

- $S$ can be represented using a queue or a stack
- Running time = $O(n + m)$
### $S$ as a Queue or a Stack

<table>
<thead>
<tr>
<th>DS</th>
<th>Queue</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialization</strong></td>
<td>head $\leftarrow 1$, tail $\leftarrow 0$</td>
<td>top $\leftarrow 0$</td>
</tr>
<tr>
<td><strong>Non-Empty?</strong></td>
<td>head $\leq$ tail</td>
<td>top $&gt; 0$</td>
</tr>
<tr>
<td><strong>Add</strong>(v)</td>
<td>tail $\leftarrow$ tail + 1</td>
<td>top $\leftarrow$ top + 1</td>
</tr>
<tr>
<td></td>
<td>$S$[tail] $\leftarrow$ v</td>
<td>$S$[top] $\leftarrow$ v</td>
</tr>
<tr>
<td><strong>Retrieve</strong> v</td>
<td>v $\leftarrow$ S[head]</td>
<td>v $\leftarrow$ S[top]</td>
</tr>
<tr>
<td></td>
<td>head $\leftarrow$ head + 1</td>
<td>top $\leftarrow$ top − 1</td>
</tr>
</tbody>
</table>
Example

![Diagram of a graph with nodes a, b, c, d, e, f, g and edges connecting them.

A queue labeled as follows:

- Head
- Tail

Queue:

- a

<table>
<thead>
<tr>
<th>degree</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

queue:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>tail</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>head</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

degree

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

A graph with nodes labeled $b$, $c$, $d$, $e$, $f$, and $g$. The edges connect $b$ to $e$, $c$ to $d$, $d$ to $e$, $c$ to $g$, and $f$ to $g$.

A queue with the elements $a$, $b$, $c$, $d$, $e$, $f$, and $g$. The current head is at the front, and the tail is at the back.

A table showing the degree of each node:

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

Queue:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

degree
Example

\[ a \quad b \quad c \quad d \quad f \quad g \]

\[ e \rightarrow b \rightarrow d \rightarrow e \quad c \rightarrow g \rightarrow f \rightarrow g \]

queue:

\[
\begin{array}{cccc}
\text{head} & a & b & c \\
\text{tail} & & & \\
\end{array}
\]

\[
\begin{array}{ccccccc}
\text{degree} & a & b & c & d & e & f & g \\
0 & 0 & 0 & 1 & 2 & 1 & 3 \\
\end{array}
\]
Example

queue:
\[
\begin{array}{c}
\text{tail} \\
\text{head}
\end{array}
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

degree

- c
- d
- f
Example

- **Queue:**
  - Components: a, b, c, d, e, f, g

- **Degree:**
  - Values: 0, 0, 0, 1, 1, 1, 3

- **Graph Diagram:**
  - Nodes: e, d, f, c, g
  - Edges: e→d, d→c, c→f, f→g, g→e

- **Queue Structure:**
  - Head and Tail
  - Arrangement: a, b, c (sorted order)
Example

Queue:

```
a b c
```

Degree:

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
```
Example

Example of a graph with nodes and edges:

- Nodes: e, d, f, g
- Edges: e → d, d → f, f → g, e → g

Queue:

```
  head
    a  b  c  d  f  
  tail
```

Degree Table:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

queue:

```
| a | b | c | d | f |
```

head

tail

degree

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
```
Example

Queue:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

degree:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Example

queue:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>尾</td>
<td>头</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

degree

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Example

queue:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

tail

head

degree

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Example

queue:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
<th>e</th>
</tr>
</thead>
</table>

\[
\text{degree} \quad 0 & 0 & 0 & 0 & 0 & 0 & 1
\]

\[\text{head}\]

\[\text{tail}\]
Example

queue: \[ \begin{array}{cccccc}
    a & b & c & d & f & e \\
\end{array} \]

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
  & a & b & c & d & e & f & g \\
\hline
degree & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{tabular}
Example

queue: \[ a \ b \ c \ d \ f \ e \]

g

degree

<table>
<thead>
<tr>
<th>degree</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

queue: \[ a \quad b \quad c \quad d \quad f \quad e \quad g \]

g

g

degree

\[ \begin{array}{cccccc}
\text{degree} & a & b & c & d & e & f & g \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]
Example

queue: \begin{array}{cccccccc}
  a & b & c & d & f & e & g \\
\end{array}

\begin{array}{cccccccc}
  \text{degree} & a & b & c & d & e & f & g \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
   - Applications: Word Ladder
**Def.** Word: A string formed by letters.

**Def.** Adjacency words: Word $A$ and $B$ are adjacent if they differ in exactly one letter.

e.g. word and work; tell and tall; askbe and askee.
Def. Word Ladder: Players start with one word and, in a series of steps, change or transform that word into another word.
Def. Word Ladder: Players start with one word and, in a series of steps, change or transform that word into another word.

- The objective is to make the change in the smallest number of steps, with each step involving changing a single letter of the word to create a new valid word.
**Word Ladder Problem**

**Input:** Two words \( S \) and \( T \), a list of words \( A = \{W_1, W_2, \ldots, W_k\} \).

**Output:** “The smallest word ladder” if we can change \( S \) to \( T \) by moving between adjacency words in \( A \cup \{S, T\} \); Otherwise, “No word ladder”.

Example:

- \( S = “a \ e \ f \ g \ h” \), \( T = “d \ l \ m \ i \ h” \)
- \( W_1 = “a \ e \ f \ i \ h” \), \( W_2 = “a \ e \ m \ g \ h” \), \( W_3 = “d \ l \ f \ i \ h” \)
  - \( W_4 = “s \ e \ f \ i \ h” \), \( W_5 = “a \ d \ f \ g \ h” \), \( W_6 = “d \ e \ m \ i \ h” \)
  - \( W_7 = “d \ e \ f \ i \ h” \), \( W_8 = “d \ e \ m \ g \ h” \), \( W_9 = “s \ e \ m \ i \ h” \)
Example:

- $S = \text{“a e f g h”}$, $T = \text{“d l m i h”}$
- $W_1 = \text{“a e f i h”}$, $W_2 = \text{“a e m g h”}$, $W_3 = \text{“d l f i h”}$
- $W_4 = \text{“s e f i h”}$, $W_5 = \text{“a d f g h”}$, $W_6 = \text{“d e m i h”}$
- $W_7 = \text{“d e f i h”}$, $W_8 = \text{“d e m g h”}$, $W_9 = \text{“s e m i h”}$

- Each vertex corresponds to a word.
- Two vertices are adjacent if the corresponding words are adjacent.
Each vertex corresponds to a word.

Two vertices are adjacent if the corresponding words are adjacent.

Hints: Given vertex $v$, check its nearest neighbor.
Def. In an optimization problem, our goal is to find a valid solution with the minimum cost (or maximum value).
Def. In an *optimization problem*, our goal is to find a valid solution with the minimum cost (or maximum value).

**Trivial Algorithm for an Optimization Problem**
Enumerate all valid solutions, compare them and output the best one.
**Def.** In an *optimization problem*, our goal of is to find a valid solution with the minimum cost (or maximum value).

**Trivial Algorithm for an Optimization Problem**
Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in *exponential* time, as the number of potential solutions is often exponentially large.
**Def.** In an optimization problem, our goal is to find a valid solution with the minimum cost (or maximum value).

**Trivial Algorithm for an Optimization Problem**

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
Def. In an optimization problem, our goal is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- $f(n)$ is a polynomial if $f(n) = O(n^k)$ for some constant $k > 0$. 
Def. In an optimization problem, our goal is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem
Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- \( f(n) \) is a polynomial if \( f(n) = O(n^k) \) for some constant \( k > 0 \).
- convention: polynomial time = efficient
**Def.** In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

**Trivial Algorithm for an Optimization Problem**

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- $f(n)$ is a polynomial if $f(n) = O(n^k)$ for some constant $k > 0$.
- convention: polynomial time = efficient

**Goals of algorithm design**
Def. In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- \( f(n) \) is a polynomial if \( f(n) = O(n^k) \) for some constant \( k > 0 \).
- convention: polynomial time = efficient

Goals of algorithm design

1. Design efficient algorithms to solve problems
Def. In an optimization problem, our goal is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem
Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- $f(n)$ is a polynomial if $f(n) = O(n^k)$ for some constant $k > 0$.
- convention: polynomial time = efficient

Goals of algorithm design
1. Design efficient algorithms to solve problems
2. Design more efficient algorithms to solve problems
Common Paradigms for Algorithm Design

- Greedy Algorithms: shortest path problem
- Divide and Conquer: merge-sort, binary search
- Dynamic Programming: Fibonacci number
Greedy algorithm properties

Greedy algorithms are often for optimization problems. They often run in polynomial time due to their simplicity: easy to come up with, easy to analyze running time. Hard to see correctness. Mostly, it is not correct. E.g. \( \min f(x) \).
Greedy algorithm properties

- Greedy algorithms are often for optimization problems.
Greedy algorithm properties

- Greedy algorithms are often for optimization problems.
- They often run in polynomial time due to their simplicity: easy to come up with, easy to analyze running time.
Greedy algorithm properties

- Greedy algorithms are often for optimization problems.
- They often run in polynomial time due to their simplicity: easy to come up with, easy to analyze running time.
- Hard to see correctness. Mostly, it is not correct. E.g. \( \min f(x) \)
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

**Analysis of Greedy Algorithm**

- **Safety**: Prove that the reasonable strategy is “safe”
- **Self-reduce**: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

**Definition**

As a strategy is safe, there is always an optimal solution that agrees with the decision made according to the strategy.
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irreversible decision using a “reasonable” strategy

Analysis of Greedy Algorithm

- Safety: Prove that the reasonable strategy is “safe”
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem
**Greedy Algorithm**

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

**Analysis of Greedy Algorithm**

- **Safety**: Prove that the reasonable strategy is “safe” (key)
- **Self-reduce**: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (**usually easy**)
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a “reasonable” strategy

Analysis of Greedy Algorithm

- Safety: Prove that the reasonable strategy is “safe” (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Def. A strategy is safe: there is always an optimum solution that agrees with the decision made according to the strategy.
Outline

1. Toy Example: Box Packing
2. Interval Scheduling
3. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue
4. Data Compression and Huffman Code
5. Summary
Box Packing

**Input:** \( n \) boxes of capacities \( c_1, c_2, \cdots, c_n \)
\( m \) items of sizes \( s_1, s_2, \cdots, s_m \)

Can put **at most 1** item in a box

Item \( j \) can be put into box \( i \) if \( s_j \leq c_i \)

**Output:** A way to put as many items as possible in the boxes.
Box Packing

**Input:** $n$ boxes of capacities $c_1, c_2, \cdots, c_n$

$m$ items of sizes $s_1, s_2, \cdots, s_m$

Can put at most 1 item in a box

Item $j$ can be put into box $i$ if $s_j \leq c_i$

**Output:** A way to put as many items as possible in the boxes.

**Example:**

- Box capacities: 60, 40, 25, 15, 12
- Item sizes: 45, 42, 20, 19, 16
- Can put 3 items in boxes: 45 $\rightarrow$ 60, 20 $\rightarrow$ 40, 19 $\rightarrow$ 25
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a “reasonable” strategy
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a “reasonable” strategy

Designing a Reasonable Strategy for Box Packing

- Q: Take box 1. Which item should we put in box 1?
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a “reasonable” strategy

Designing a Reasonable Strategy for Box Packing

- Q: Take box 1. Which item should we put in box 1?
- A: The item of the largest size that can be put into the box.
Analysis of Greedy Algorithm

- **Safety:** Prove that the reasonable strategy is “safe”
- **Self-reduce:** Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

Lemma
The strategy that put into box 1 the largest item it can hold is “safe”:
There is an optimum solution in which box 1 contains the largest item it can hold.
Intuition: putting the item gives us the easiest residual problem.

formal proof via exchanging argument:
Analysis of Greedy Algorithm

- Safety: Prove that the reasonable strategy is “safe”
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

**Lemma** The strategy that put into box 1 the largest item it can hold is “safe”: There is an optimum solution in which box 1 contains the largest item it can hold.
Analysis of Greedy Algorithm

- Safety: Prove that the reasonable strategy is “safe”
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

**Lemma** The strategy that put into box 1 the largest item it can hold is “safe”: There is an optimum solution in which box 1 contains the largest item it can hold.

- Intuition: putting the item gives us the easiest residual problem.
Analysis of Greedy Algorithm

- **Safety**: Prove that the reasonable strategy is “safe”
- **Self-reduce**: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

**Lemma** The strategy that put into box 1 the largest item it can hold is “safe”: There is an optimum solution in which box 1 contains the largest item it can hold.

- Intuition: putting the item gives us the easiest residual problem.
- Formal proof via exchanging argument:
Lemma  There is an optimum solution in which box 1 contains the largest item it can hold.
**Lemma**  There is an optimum solution in which box 1 contains the largest item it can hold.

**Proof.**

- Let $j =$ largest item that box 1 can hold.
Lemma  There is an optimum solution in which box 1 contains the largest item it can hold.

Proof.  
- Let \( j \) = largest item that box 1 can hold.  
- Take any optimum solution \( S \). If \( j \) is put into Box 1 in \( S \), done.
**Lemma** There is an optimum solution in which box 1 contains the largest item it can hold.

**Proof.**

- Let \( j \) = largest item that box 1 can hold.
- Take any optimum solution \( S \). If \( j \) is put into Box 1 in \( S \), done.
- Otherwise, assume this is what happens in \( S \):

\[
S:\quad \text{box 1} \quad \text{item } j \quad \text{…….} \quad \text{Box 1}
\]

\( S_0 \) is also an optimum solution. In \( S_0 \), \( j \) is put into Box 1.
**Lemma** There is an optimum solution in which box 1 contains the largest item it can hold.

**Proof.**

- Let \( j \) = largest item that box 1 can hold.
- Take any optimum solution \( S \). If \( j \) is put into Box 1 in \( S \), done.
- Otherwise, assume this is what happens in \( S \):
  
  \[ S' : \]
  
  ![Diagram showing box 1 with items](image)

- \( s_{j'} \leq s_j \), and swapping gives another solution \( S' \)
Lemma There is an optimum solution in which box 1 contains the largest item it can hold.

Proof.

- Let $j =$ largest item that box 1 can hold.
- Take any optimum solution $S$. If $j$ is put into Box 1 in $S$, done.
- Otherwise, assume this is what happens in $S$:

  $S'$:

  box 1

  

  item $j'$  item $j$

  $s_{j'} \leq s_j$, and swapping gives another solution $S'$

  $S'$ is also an optimum solution. In $S'$, $j$ is put into Box 1.
Notice that the exchanging operation is only for the sake of analysis; it is not a part of the algorithm.
Notice that the exchanging operation is only for the sake of analysis; it is not a part of the algorithm.

**Analysis of Greedy Algorithm**

- **Safety:** Prove that the reasonable strategy is “safe”
- **Self-reduce:** Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem
Notice that the exchanging operation is only for the sake of analysis; it is not a part of the algorithm.

Analysis of Greedy Algorithm

- **Safety**: Prove that the reasonable strategy is “safe”
- **Self-reduce**: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

- **Trivial**: we decided to put Item $j$ into Box 1, and the remaining instance is obtained by removing Item $j$ and Box 1.
**Generic Greedy Algorithm**

1. **while** the instance is non-trivial **do**
2. make the choice using the greedy strategy
3. reduce the instance

**Greedy Algorithm for Box Packing**

1. \( T \leftarrow \{1, 2, 3, \ldots, m\} \)
2. **for** \( i \leftarrow 1 \) to \( n \) **do**
3. \[ \text{if some item in } T \text{ can be put into box } i \text{ then} \]
4. \( j \leftarrow \) the largest item in \( T \) that can be put into box \( i \)
5. \( \text{print(“put item } j \text{ in box } i”) } \)
6. \( T \leftarrow T \setminus \{j\} \)