Algorithmic Insights II - Greedy and Dynamic Programming

K. Subramani

Lane Department of Computer Science and Electrical Engineering
West Virginia University

February 16 and February 23, 2015
1. Review of concepts
Outline

1. Review of concepts
2. The Greedy Approach
Outline

1. Review of concepts
2. The Greedy Approach
3. Dynamic Programming
Review of concepts
The Greedy Approach
Dynamic Programming

Algorithmic Insights

Computational Complexity
Review

Algorithmic Insights
Algorithmic Insights

1. Recursion.
Algorithmic Insights

1. Recursion.
2. Divide and Conquer.
Review of concepts

The Greedy Approach

Dynamic Programming

Algorithmic Insights

1. Recursion.
2. Divide and Conquer.
3. Greedy.

Computational Complexity
Algorithmic Insights

1. Recursion.
2. Divide and Conquer.
3. Greedy.
Review

Algorithmic Insights

1. Recursion.
2. Divide and Conquer.
3. Greedy.
5. Iterative approaches (Rewriting).
Review of concepts
The Greedy Approach
Dynamic Programming

Algorithmic Insights

1. Recursion.
2. Divide and Conquer.
3. Greedy.
5. Iterative approaches (Rewriting).
6. Transformations and reductions.
The Greedy Approach

Main Idea

1. Formulate a greedy criterion (usually a simple one).
2. Start with an empty solution set, which must be feasible.
3. Prove that the greedy choice is always safe. (Usually involves an exchange argument).
4. Add items one at a time to the current feasible solution, using the greedy criterion.
5. Terminate when all items have been considered or a maximum feasible subset has been reached.

Algorithmic Insights

Computational Complexity
The Greedy Approach

Main Idea

1. Formulate a greedy criterion (usually a simple one).
2. Start with an empty solution set, which must be feasible.
3. Prove that the greedy choice is always safe. (Usually involves an exchange argument).
4. Add items one at a time to the current feasible solution, using the greedy criterion.
5. Terminate when all items have been considered or a maximum feasible subset has been reached.

Algorithmic Insights
Computational Complexity
Main Idea

1. Formulate a greedy criterion
The Greedy Approach

Main Idea

1. Formulate a greedy criterion (usually a simple one).
The Greedy Approach

Main Idea

1. Formulate a greedy criterion (usually a simple one).
2. Start with an empty solution set, which must be feasible.
Main Idea

1. Formulate a greedy criterion (usually a simple one).
2. Start with an empty solution set, which must be feasible.
3. Prove that the greedy choice is always safe.
The Greedy Approach

Main Idea

1. Formulate a greedy criterion (usually a simple one).
2. Start with an empty solution set, which must be feasible.
3. Prove that the greedy choice is always safe. (Usually involves an exchange argument).
The Greedy Approach

Main Idea

1. Formulate a greedy criterion (usually a simple one).
2. Start with an empty solution set, which must be feasible.
3. Prove that the greedy choice is always safe. (Usually involves an exchange argument).
4. Add items one at a time to the current feasible solution, using the greedy criterion.
The Greedy Approach

Main Idea

1. Formulate a greedy criterion (usually a simple one).
2. Start with an empty solution set, which must be feasible.
3. Prove that the greedy choice is always safe. (Usually involves an exchange argument).
4. Add items one at a time to the current feasible solution, using the greedy criterion.
5. Terminate when all items have been considered or a maximum feasible subset has been reached.
The file storage problem

You are given \( n \) files \( F_1, F_2, \ldots, F_n \), which have to be stored on tape.

File \( F_i \) has length \( l_i \), i.e., it has \( l_i \) records.

The cost of accessing a file is equal to its position on the tape. Thus, the cost of accessing the \( k \)th file is:

\[
\sum_{i=1}^{k} l_i.
\]

Assuming that each file is equally likely to be accessed, the expected cost of accessing a random file is:

\[
E[\text{cost}] = \frac{1}{n} \cdot \sum_{i=1}^{n} l_i.
\]

Different orders of file storage give rise to different expected costs.

In what order should the files be stored, so that the expected cost is minimized?
The file storage problem

Problem

You are given \( n \) files \( F_1, F_2, \ldots, F_n \), which have to be stored on tape. File \( F_i \) has length \( l_i \), i.e., it has \( l_i \) records. The cost of accessing a file is equal to its position on the tape. Thus, the cost of accessing the \( k \)th file is:

\[
\sum_{i=1}^{k} l_i
\]

Assuming that each file is equally likely to be accessed, the expected cost of accessing a random file is:

\[
E[\text{cost}] = \frac{1}{n} \cdot \sum_{i=1}^{n} l_i
\]

Different orders of file storage give rise to different expected costs. In what order should the files be stored, so that the expected cost is minimized?
The file storage problem

You are given $n$ files $F_1, F_2 \ldots, F_n$, which have to be stored on tape.
You are given $n$ files $F_1, F_2, \ldots, F_n$, which have to be stored on tape. File $F_i$ has length $l_i$, i.e., it has $l_i$ records.
The file storage problem

Problem

1. You are given $n$ files $F_1, F_2, \ldots, F_n$, which have to be stored on tape.
2. File $F_i$ has length $l_i$, i.e., it has $l_i$ records.
3. The cost of accessing a file is equal to its position on the tape.
The file storage problem

1. You are given \( n \) files \( F_1, F_2, \ldots, F_n \), which have to be stored on tape.
2. File \( F_i \) has length \( l_i \), i.e., it has \( l_i \) records.
3. The cost of accessing a file is equal to its position on the tape. Thus, the cost of accessing the \( k^{th} \) file is:
You are given \( n \) files \( F_1, F_2 \ldots, F_n \), which have to be stored on tape.

File \( F_i \) has length \( l_i \), i.e., it has \( l_i \) records.

The cost of accessing a file is equal to its position on the tape. Thus, the cost of accessing the \( k^{th} \) file is: \( \sum_{i=1}^{k} l_i \).
You are given \( n \) files \( F_1, F_2, \ldots, F_n \), which have to be stored on tape.

File \( F_i \) has length \( l_i \), i.e., it has \( l_i \) records.

The cost of accessing a file is equal to its position on the tape. Thus, the cost of accessing the \( k^{th} \) file is: \( \sum_{i=1}^{k} l_i \).

Assuming that each file is equally likely to be accessed,
The file storage problem

You are given \( n \) files \( F_1, F_2 \ldots, F_n \), which have to be stored on tape.

1. File \( F_i \) has length \( l_i \), i.e., it has \( l_i \) records.

2. The cost of accessing a file is equal to its position on the tape. Thus, the cost of accessing the \( k^{th} \) file is: \( \sum_{i=1}^{k} l_i \).

3. Assuming that each file is equally likely to be accessed, the expected cost of accessing a random file is: \( E[\text{cost}] = \).
The file storage problem

**Problem**

1. You are given $n$ files $F_1, F_2, \ldots, F_n$, which have to be stored on tape.
2. File $F_i$ has length $l_i$, i.e., it has $l_i$ records.
3. The cost of accessing a file is equal to its position on the tape. Thus, the cost of accessing the $k^{th}$ file is: $\sum_{i=1}^{k} l_i$.
4. Assuming that each file is equally likely to be accessed, the expected cost of accessing a random file is: $E[\text{cost}] = \frac{1}{n} \cdot \sum_{i=1}^{n} \sum_{j=1}^{i} l_j$. 
The file storage problem

Problem

1. You are given \( n \) files \( F_1, F_2 \ldots, F_n \), which have to be stored on tape.
2. File \( F_i \) has length \( l_i \), i.e., it has \( l_i \) records.
3. The cost of accessing a file is equal to its position on the tape. Thus, the cost of accessing the \( k^{th} \) file is: \( \sum_{i=1}^{k} l_i \).
4. Assuming that each file is equally likely to be accessed, the expected cost of accessing a random file is: \( E[\text{cost}] = \frac{1}{n} \cdot \sum_{i=1}^{n} \sum_{j=1}^{i} l_j \).
5. Different orders of file storage give rise to different expected costs.
The file storage problem

1. You are given $n$ files $F_1, F_2, \ldots, F_n$, which have to be stored on tape.
2. File $F_i$ has length $l_i$, i.e., it has $l_i$ records.
3. The cost of accessing a file is equal to its position on the tape. Thus, the cost of accessing the $k^{th}$ file is: $\sum_{i=1}^{k} l_i$.
4. Assuming that each file is equally likely to be accessed, the expected cost of accessing a random file is: $E[\text{cost}] = \frac{1}{n} \cdot \sum_{i=1}^{n} \sum_{j=1}^{i} l_j$.
5. Different orders of file storage give rise to different expected costs.
6. In what order should the files be stored, so that the expected cost is minimized?
The files should be stored in increasing order of length on the tape, i.e., if $l_i \leq l_j$, then $F_i$ must precede $F_j$ on the tape.

**Proof**

1. Assume that there exists an optimal solution in which the files on the tape are not in increasing order of length.
2. So there must be files $F_i$ and $F_j$ such that $l_i < l_j$, but $F_i$ is stored after $F_j$.
3. Without loss of generality, we assume that $F_i$ and $F_j$ are adjacent files. (Why can we assume this?)
4. Switch these two files! The expected cost decreases by: $(l_j - l_i) n$.
5. Thus, a non-ordered organization cannot be optimal.
The files should be stored in increasing order of length on the tape, i.e., if \( l_i \leq l_j \), then \( F_i \) must precede \( F_j \) on the tape.

**Proof**

1. Assume that there exists an optimal solution in which the files on the tape are not in increasing order of length.
2. So there must be files \( F_i \) and \( F_j \) such that \( l_i < l_j \), but \( F_i \) is stored after \( F_j \).
3. Without loss of generality, we assume that \( F_i \) and \( F_j \) are adjacent files. (Why can we assume this?)
4. Switch these two files! The expected cost decreases by: \( (l_j - l_i) \).
5. Thus, a non-ordered organization cannot be optimal.
Solution

1. The files should be stored in increasing order of length on the tape,
The files should be stored in increasing order of length on the tape, i.e., if $l_i \leq l_j$, then $F_i$ must precede $F_j$ on the tape.
Solution

1. The files should be stored in increasing order of length on the tape, i.e., if \( l_i \leq l_j \), then \( F_i \) must precede \( F_j \) on the tape.

Proof
Solution

1. The files should be stored in increasing order of length on the tape, i.e., if $l_i \leq l_j$, then $F_i$ must precede $F_j$ on the tape.

Proof

1. Assume that there exists an optimal solution in which the files on the tape are not in increasing order of length.
Solution

1. The files should be stored in increasing order of length on the tape, i.e., if $l_i \leq l_j$, then $F_i$ must precede $F_j$ on the tape.

Proof

1. Assume that there exists an optimal solution in which the files on the tape are not in increasing order of length.

2. So there must be files $F_i$ and $F_j$ such that $l_i < l_j$, but $F_i$ is stored after $F_j$. 
File storage

Solution

1. The files should be stored in increasing order of length on the tape, i.e., if $l_i \leq l_j$, then $F_i$ must precede $F_j$ on the tape.

Proof

1. Assume that there exists an optimal solution in which the files on the tape are not in increasing order of length.
2. So there must be files $F_i$ and $F_j$ such that $l_i < l_j$, but $F_i$ is stored after $F_j$.
3. Without loss of generality, we assume that $F_i$ and $F_j$ are adjacent files.
Solution

1. The files should be stored in increasing order of length on the tape, i.e., if $l_i \leq l_j$, then $F_i$ must precede $F_j$ on the tape.

Proof

1. Assume that there exists an optimal solution in which the files on the tape are not in increasing order of length.
2. So there must be files $F_i$ and $F_j$ such that $l_i < l_j$, but $F_i$ is stored after $F_j$.
3. Without loss of generality, we assume that $F_i$ and $F_j$ are adjacent files. (Why can we assume this?)
File storage

Solution

1. The files should be stored in increasing order of length on the tape, i.e., if $l_i \leq l_j$, then $F_i$ must precede $F_j$ on the tape.

Proof

1. Assume that there exists an optimal solution in which the files on the tape are not in increasing order of length.
2. So there must be files $F_i$ and $F_j$ such that $l_i < l_j$, but $F_i$ is stored after $F_j$.
3. Without loss of generality, we assume that $F_i$ and $F_j$ are adjacent files. (Why can we assume this?)
4. Switch these two files!
Solution

1. The files should be stored in increasing order of length on the tape, i.e., if $l_i \leq l_j$, then $F_i$ must precede $F_j$ on the tape.

Proof

1. Assume that there exists an optimal solution in which the files on the tape are not in increasing order of length.
2. So there must be files $F_i$ and $F_j$ such that $l_i < l_j$, but $F_i$ is stored after $F_j$.
3. Without loss of generality, we assume that $F_i$ and $F_j$ are adjacent files. (Why can we assume this?)
4. Switch these two files! The expected cost decreases by:
Solution

1. The files should be stored in increasing order of length on the tape, i.e., if $l_i \leq l_j$, then $F_i$ must precede $F_j$ on the tape.

Proof

1. Assume that there exists an optimal solution in which the files on the tape are not in increasing order of length.
2. So there must be files $F_i$ and $F_j$ such that $l_i < l_j$, but $F_i$ is stored after $F_j$.
3. Without loss of generality, we assume that $F_i$ and $F_j$ are adjacent files. (Why can we assume this?)
4. Switch these two files! The expected cost decreases by: $\frac{(l_j - l_i)}{n}$.
Solution

1. The files should be stored in increasing order of length on the tape, i.e., if $l_i \leq l_j$, then $F_i$ must precede $F_j$ on the tape.

Proof

1. Assume that there exists an optimal solution in which the files on the tape are not in increasing order of length.
2. So there must be files $F_i$ and $F_j$ such that $l_i < l_j$, but $F_i$ is stored after $F_j$.
3. Without loss of generality, we assume that $F_i$ and $F_j$ are adjacent files. (Why can we assume this?)
4. Switch these two files! The expected cost decreases by: $\frac{(l_j - l_i)}{n}$.
5. Thus, a non-ordered organization cannot be optimal.
The Minimum Spanning Tree problem

Problem
Given an edge-weighted, undirected graph $G = \langle V, E, c \rangle$, find a spanning tree of minimum weight.

Greedy Approach
1: Order the edges in $E$ in ascending order of weight.
2: W.l.o.g. assume that $c(e_1) \leq c(e_2) \leq \ldots c(e_m)$.
3: $T \rightarrow \emptyset$.
4: for $(i = 1$ to $m)$ do
5: if $(T \cup \{e_i\}$ does not have a cycle)
6: $T \rightarrow (T \cup \{e_i\}$.
7: end if
8: end for

Algorithm 3.1: Kruskal's algorithm

Algorithmic Insights
Computational Complexity
The Minimum Spanning Tree problem

Problem

Given an edge-weighted, undirected graph $G = \langle V, E, c \rangle$, find a spanning tree of minimum weight.

Greedy Approach

1: Order the edges in $E$ in ascending order of weight.
2: W.l.o.g. assume that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$.
3: $T \rightarrow \emptyset$.
4: for ($i = 1$ to $m$) do
5:  if ($T \cup \{e_i\}$ does not have a cycle) then
6:      $T \rightarrow (T \cup \{e_i\})$.
7:  end if
8: end for

Algorithm 3.2: Kruskal's algorithm
The Minimum Spanning Tree problem

Problem

Given an edge-weighted, undirected graph $G = (V, E, c)$,
The Minimum Spanning Tree problem

Problem

Given an edge-weighted, undirected graph $G = \langle V, E, c \rangle$, find a spanning tree of minimum weight.
The Minimum Spanning Tree problem

**Problem**
Given an edge-weighted, undirected graph $G = \langle V, E, c \rangle$, find a spanning tree of minimum weight.

**Greedy Approach**

1: Order the edges in $E$ in ascending order of weight.
2: W.l.o.g. assume that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$.
3: $T \rightarrow \emptyset$.
4: for ($i = 1$ to $m$) do
5: if ($T \cup \{e_i\}$ does not have a cycle)
6: $T \rightarrow (T \cup \{e_i\})$.
7: end if
8: end for

Algorithm 3.5: Kruskal’s algorithm
The Minimum Spanning Tree problem

Problem

Given an edge-weighted, undirected graph $G = \langle V, E, c \rangle$, find a spanning tree of minimum weight.

Greedy Approach

1: Order the edges in $E$ in ascending order of weight.
The Minimum Spanning Tree problem

**Problem**

Given an edge-weighted, undirected graph $G = \langle V, E, c \rangle$, find a spanning tree of minimum weight.

**Greedy Approach**

1. Order the edges in $E$ in ascending order of weight.
2. W.l.o.g. assume that $c(e_1) \leq c(e_2) \leq \ldots c(e_m)$. 
The Minimum Spanning Tree problem

Problem

Given an edge-weighted, undirected graph $G = \langle V, E, c \rangle$, find a spanning tree of minimum weight.

Greedy Approach

1: Order the edges in $E$ in ascending order of weight.
2: W.l.o.g. assume that $c(e_1) \leq c(e_2) \leq \ldots c(e_m)$.
3: $T \rightarrow \emptyset$. 
The Minimum Spanning Tree problem

Problem
Given an edge-weighted, undirected graph $G = \langle V, E, c \rangle$, find a spanning tree of minimum weight.

Greedy Approach

1: Order the edges in $E$ in ascending order of weight.
2: W.l.o.g. assume that $c(e_1) \leq c(e_2) \leq \ldots c(e_m)$.
3: $T \rightarrow \emptyset$.
4: for ($i = 1$ to $m$) do
The Minimum Spanning Tree problem

Problem

Given an edge-weighted, undirected graph $G = \langle V, E, c \rangle$, find a spanning tree of minimum weight.

Greedy Approach

1: Order the edges in $E$ in ascending order of weight.
2: W.l.o.g. assume that $c(e_1) \leq c(e_2) \leq \ldots c(e_m)$.
3: $T \rightarrow \emptyset$.
4: for ($i = 1$ to $m$) do
5: if $((T \cup \{e_i\})$ does not have a cycle) then
The Minimum Spanning Tree problem

**Problem**

Given an edge-weighted, undirected graph $G = \langle V, E, c \rangle$, find a spanning tree of minimum weight.

**Greedy Approach**

1: Order the edges in $E$ in ascending order of weight.
2: W.l.o.g. assume that $c(e_1) \leq c(e_2) \leq \ldots c(e_m)$.
3: $T \rightarrow \emptyset$.
4: for ($i = 1$ to $m$) do
5:  
6:     if (($T \cup \{e_i\})$ does not have a cycle) then
7:         $T \rightarrow (T \cup \{e_i\})$. 
The Minimum Spanning Tree problem

## Problem

Given an edge-weighted, undirected graph \( G = \langle V, E, c \rangle \), find a spanning tree of minimum weight.

## Greedy Approach

1. Order the edges in \( E \) in ascending order of weight.
2. W.l.o.g. assume that \( c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m) \).
3. \( T \rightarrow \emptyset \).
4. for \( i = 1 \) to \( m \) do
5. if \( (T \cup \{e_i\}) \) does not have a cycle then
6. \( T \rightarrow (T \cup \{e_i\}) \).
7. end if
The Minimum Spanning Tree problem

**Problem**

Given an edge-weighted, undirected graph $G = \langle V, E, c \rangle$, find a spanning tree of minimum weight.

**Greedy Approach**

1. Order the edges in $E$ in ascending order of weight.
2. W.l.o.g. assume that $c(e_1) \leq c(e_2) \leq \ldots c(e_m)$.
3. $T \rightarrow \emptyset$.
4. for ($i = 1$ to $m$) do
5.    if ($(T \cup \{e_i\})$ does not have a cycle) then
6.      $T \rightarrow (T \cup \{e_i\})$.
7.    end if
8. end for

**Algorithm 3.13:** Kruskal's algorithm
Proof of Kruskal

Definition
A cut in an undirected graph is any partition of the vertices into two disjoint subsets.

Any cut determines a cut-set.

Theorem
Let C denote a cut-set corresponding to some cut in an undirected graph G.
There is an MST of G, which includes the lightest edge in C.
Proof of Kruskal

Definition

A cut in an undirected graph is any partition of the vertices into two disjoint subsets. Any cut determines a cut-set.

Theorem
Let $C$ denote a cut-set corresponding to some cut in an undirected graph $G$. There is an MST of $G$ which includes the lightest edge in $C$. 

Algorithmic Insights

Computational Complexity
Definition

A cut in an undirected graph is any partition of the vertices into two disjoint subsets.
Definition

A cut in an undirected graph is any partition of the vertices into two disjoint subsets. Any cut determines a cut-set.
Proof of Kruskal

Definition

A cut in an undirected graph is any partition of the vertices into two disjoint subsets. Any cut determines a cut-set.

Theorem

[Theorem content is not visible in the image]
Definition

A cut in an undirected graph is any partition of the vertices into two disjoint subsets. Any cut determines a cut-set.

Theorem

Let $C$ denote a cut-set corresponding to some cut in an undirected graph $G$. 
Proof of Kruskal

Definition

A cut in an undirected graph is any partition of the vertices into two disjoint subsets. Any cut determines a cut-set.

Theorem

Let $C$ denote a cut-set corresponding to some cut in an undirected graph $G$. There is an MST of $G$, which includes the lightest edge in $C$. 
The fractional knapsack problem

You are given $n$ objects $o_i, i = 1, 2, ..., n$.

Object $o_i$ has weight $w_i$ and profit $p_i$.

You are given a knapsack of capacity $W$.

You are permitted to choose a fraction of an object. Pack the objects into the knapsack, so as to maximize profit, without violating the capacity constraint.

Greedy Algorithm

1: W.l.o.g. assume that $p_1 w_1 \geq p_2 w_2 \geq ... \geq p_n w_n$.

2: for ($i = 1$ to $n$) do

3: Pack as much of object $o_i$ as you can in the knapsack.

4: end for
The fractional knapsack problem

You are given $n$ objects $o_i$, $i = 1, 2, ..., n$.

Object $o_i$ has weight $w_i$ and profit $p_i$.

You are given a knapsack of capacity $W$.

You are permitted to choose a fraction of an object. Pack the objects into the knapsack, so as to maximize profit, without violating the capacity constraint.

Greedy Algorithm

1: W.l.o.g. assume that $p_1 w_1 \geq p_2 w_2 ... \geq p_n w_n$.

2: for ($i = 1$ to $n$) do

3: Pack as much of object $o_i$ as you can in the knapsack.

4: end for

Algorithmic Insights

Computational Complexity
The fractional knapsack problem

The Problem

1. You are given \( n \) objects \( o_i, i = 1, 2, \ldots, n \).
The fractional knapsack problem

The Problem

1. You are given $n$ objects $o_i$, $i = 1, 2, \ldots, n$.
2. Object $o_i$ has weight $w_i$ and profit $p_i$.
The fractional knapsack problem

The Problem

1. You are given $n$ objects $o_i$, $i = 1, 2, \ldots, n$.
2. Object $o_i$ has weight $w_i$ and profit $p_i$.
3. You are given a knapsack of capacity $W$. 

Greedy Algorithm

1. W.l.o.g. assume that $p_1 w_1 \geq p_2 w_2 \ldots \geq p_n w_n$.
2. for $i = 1$ to $n$
   3. Pack as much of object $o_i$ as you can in the knapsack.
4. end for
The fractional knapsack problem

The Problem

1. You are given \( n \) objects \( o_i, i = 1, 2, \ldots, n \).
2. Object \( o_i \) has weight \( w_i \) and profit \( p_i \).
3. You are given a knapsack of capacity \( W \).
4. You are permitted to choose a fraction of an object.
The fractional knapsack problem

The Problem

1. You are given $n$ objects $o_i$, $i = 1, 2, \ldots, n$.
2. Object $o_i$ has weight $w_i$ and profit $p_i$.
3. You are given a knapsack of capacity $W$.
4. You are permitted to choose a fraction of an object.

Pack the objects into the knapsack, so as to maximize profit, without violating the capacity constraint.
The fractional knapsack problem

**The Problem**

1. You are given \( n \) objects \( o_i, \ i = 1, 2, \ldots, n \).
2. Object \( o_i \) has weight \( w_i \) and profit \( p_i \).
3. You are given a knapsack of capacity \( W \).
4. You are permitted to choose a fraction of an object.

Pack the objects into the knapsack, so as to maximize profit, without violating the capacity constraint.

**Greedy Algorithm**

1. W.l.o.g. assume that \( p_1 w_1 \geq p_2 w_2 \ldots \geq p_n w_n \).
2. for \( i = 1 \) to \( n \) do
3.   Pack as much of object \( o_i \) as you can in the knapsack.
4. end for
The fractional knapsack problem

**The Problem**

1. You are given \( n \) objects \( o_i, i = 1, 2, \ldots, n \).
2. Object \( o_i \) has weight \( w_i \) and profit \( p_i \).
3. You are given a knapsack of capacity \( W \).
4. You are permitted to choose a fraction of an object.

Pack the objects into the knapsack, so as to maximize profit, without violating the capacity constraint.

**Greedy Algorithm**

1. W.l.o.g. assume that \( \frac{p_1}{w_1} \geq \frac{p_2}{w_2} \ldots \geq \frac{p_n}{w_n} \).
The fractional knapsack problem

The Problem

1. You are given $n$ objects $o_i$, $i = 1, 2, \ldots, n$.
2. Object $o_i$ has weight $w_i$ and profit $p_i$.
3. You are given a knapsack of capacity $W$.
4. You are permitted to choose a fraction of an object.

Pack the objects into the knapsack, so as to maximize profit, without violating the capacity constraint.

Greedy Algorithm

1. W.l.o.g. assume that $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \ldots \geq \frac{p_n}{w_n}$.
2. for $(i = 1 \text{ to } n)$ do
The fractional knapsack problem

The Problem

1. You are given \( n \) objects \( o_i, i = 1, 2, \ldots, n \).
2. Object \( o_i \) has weight \( w_i \) and profit \( p_i \).
3. You are given a knapsack of capacity \( W \).
4. You are permitted to choose a fraction of an object.

Pack the objects into the knapsack, so as to maximize profit, without violating the capacity constraint.

Greedy Algorithm

1. W.l.o.g. assume that \( \frac{p_1}{w_1} \geq \frac{p_2}{w_2} \ldots \geq \frac{p_n}{w_n} \).
2. for \( i = 1 \) to \( n \) do
3. Pack as much of object \( o_i \) as you can in the knapsack.
The fractional knapsack problem

The Problem

1. You are given \( n \) objects \( o_i, i = 1, 2, \ldots, n \).
2. Object \( o_i \) has weight \( w_i \) and profit \( p_i \).
3. You are given a knapsack of capacity \( W \).
4. You are permitted to choose a fraction of an object.

Pack the objects into the knapsack, so as to maximize profit, without violating the capacity constraint.

Greedy Algorithm

1. W.l.o.g. assume that \( \frac{p_1}{w_1} \geq \frac{p_2}{w_2} \ldots \geq \frac{p_n}{w_n} \).
2. for \( (i = 1 \text{ to } n) \) do
3. Pack as much of object \( o_i \) as you can in the knapsack.
4. end for
Correctness

Note that the greedy solution will have structure \( \langle 1, 1, \alpha, 0, \ldots \rangle \).

Assume that there exists an optimal solution which is superior to the greedy solution.

Let \( k \) be the first index at which the optimal solution differs from the greedy solution.

Let \( \alpha_k \) and \( \alpha'_k \) denote the fractions of the greedy and optimal solutions respectively.

Observe that \( \alpha_k \) must be greater than \( \alpha'_k \).

Use an exchange argument.
Proof of correctness

Note that the greedy solution will have structure \langle 1, 1, \alpha, 0, \ldots \rangle.

Assume that there exists an optimal solution which is superior to the greedy solution.

Let \( k \) be the first index at which the optimal solution differs from the greedy solution.

Let \( \alpha_k \) and \( \alpha'_k \) denote the fractions of the greedy and optimal solutions respectively.

Observe that \( \alpha_k \) must be greater than \( \alpha'_k \).

Use an exchange argument.
Proof of correctness

1. Note that the greedy solution will have structure $\langle 1, 1, \alpha, 0, \ldots \rangle$. 
Correctness

Proof of correctness

1. Note that the greedy solution will have structure $\langle 1, 1, \alpha, 0, \ldots \rangle$.
2. Assume that there exists an optimal solution which is superior to the greedy solution.
Correctness

Proof of correctness

1. Note that the greedy solution will have structure $\langle 1, 1, \alpha, 0, \ldots \rangle$.
2. Assume that there exists an optimal solution which is superior to the greedy solution.
3. Let $k$ be the first index at which the optimal solution differs from the greedy solution.
Correctness

Proof of correctness

1. Note that the greedy solution will have structure $\langle 1, 1, \alpha, 0, \ldots \rangle$.
2. Assume that there exists an optimal solution which is superior to the greedy solution.
3. Let $k$ be the first index at which the optimal solution differs from the greedy solution.
4. Let $\alpha_k$ and $\alpha'_k$ denote the fractions of the greedy and optimal solutions respectively.
Correctness

**Proof of correctness**

1. Note that the greedy solution will have structure \( \langle 1, 1, \alpha, 0, \ldots \rangle \).
2. Assume that there exists an optimal solution which is superior to the greedy solution.
3. Let \( k \) be the first index at which the optimal solution differs from the greedy solution.
4. Let \( \alpha_k \) and \( \alpha'_k \) denote the fractions of the greedy and optimal solutions respectively.
5. Observe that \( \alpha_k \) must be greater than \( \alpha'_k \).
Proof of correctness

1. Note that the greedy solution will have structure \( \langle 1, 1, \alpha, 0, \ldots \rangle \).
2. Assume that there exists an optimal solution which is superior to the greedy solution.
3. Let \( k \) be the first index at which the optimal solution differs from the greedy solution.
4. Let \( \alpha_k \) and \( \alpha'_k \) denote the fractions of the greedy and optimal solutions respectively.
5. Observe that \( \alpha_k \) must be greater than \( \alpha'_k \).
6. Use an exchange argument.
You are given \( n \) unit time jobs, \( J_i, i=1,2,...,n \).

Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).

If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.

Greedy Algorithm

1: Order the jobs in descending order of profit.
2: Assume that \( p_1 \geq p_2 \geq ... \geq p_n \).
3: Let \( S = \emptyset \).
4: for \( i = 1 \) to \( n \) do
5: if \( (S \cup \{J_i\} \) is feasible) then
6: \( S \rightarrow S \cup \{J_i\} \).
7: end if
8: end for
You are given \( n \) unit time jobs, \( J_i, i = 1, 2, \ldots, n \).

Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).

If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.

**Greedy Algorithm**

1. Order the jobs in descending order of profit.
2. Assume that \( p_1 \geq p_2 \ldots \geq p_n \).
3. Let \( S = \emptyset \).
4. for \( (i = 1 \text{ to } n) \) do
   5. if \( (S \cup \{J_i\} \text{ is feasible}) \) then
      6. \( S \rightarrow S \cup \{J_i\} \).
   7. end if
   8. end for
## The problem

1. You are given $n$ unit time jobs, $J_i$, $i = 1, 2, \ldots, n$. 

### Algorithm 3.16: Job scheduling
Scheduling with profits and deadlines

The problem

1. You are given \( n \) unit time jobs, \( J_i, i = 1, 2, \ldots, n \).
2. Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).
Scheduling with profits and deadlines

The problem

1. You are given \( n \) unit time jobs, \( J_i, i = 1, 2, \ldots, n \).
2. Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).
3. If a job commences execution after its deadline, its profit is 0.
Scheduling with profits and deadlines

The problem

1. You are given $n$ unit time jobs, $J_i$, $i = 1, 2, \ldots, n$.
2. Job $J_i$ has a deadline $d_i$ and a profit $p_i$.
3. If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.
Scheduling with profits and deadlines

The problem

1. You are given \( n \) unit time jobs, \( J_i, i = 1, 2, \ldots, n \).
2. Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).
3. If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.

Greedy Algorithm
The problem

1. You are given \( n \) unit time jobs, \( J_i, i = 1, 2, \ldots, n \).
2. Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).
3. If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.

Greedy Algorithm

1. Order the jobs in descending order of profit.
Scheduling with profits and deadlines

The problem

1. You are given \( n \) unit time jobs, \( J_i, i = 1, 2, \ldots, n \).
2. Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).
3. If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.

Greedy Algorithm

1. Order the jobs in descending order of profit.
2. Assume that \( p_1 \geq p_2 \ldots \geq p_n \).
Scheduling with profits and deadlines

The problem

1. You are given $n$ unit time jobs, $J_i$, $i = 1, 2, \ldots, n$.
2. Job $J_i$ has a deadline $d_i$ and a profit $p_i$.
3. If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.

Greedy Algorithm

1. Order the jobs in descending order of profit.
2. Assume that $p_1 \geq p_2 \ldots \geq p_n$.
3. Let $S = \emptyset$. 
Scheduling with profits and deadlines

The problem

1. You are given \( n \) unit time jobs, \( J_i, i = 1, 2, \ldots, n \).
2. Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).
3. If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.

Greedy Algorithm

1. Order the jobs in descending order of profit.
2. Assume that \( p_1 \geq p_2 \ldots \geq p_n \).
3. Let \( S = \emptyset \).
4. for \( (i = 1 \text{ to } n) \) do

Algorithm 3.24: Job scheduling
The problem

1. You are given \( n \) unit time jobs, \( J_i, i = 1, 2, \ldots, n \).
2. Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).
3. If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.

Greedy Algorithm

1. Order the jobs in descending order of profit.
2. Assume that \( p_1 \geq p_2 \ldots \geq p_n \).
3. Let \( S = \emptyset \).
4. for \( (i = 1 \text{ to } n) \) do
5. \hspace{1em} if \( (S \cup \{J_i\} \) is feasible) then
Scheduling with profits and deadlines

The problem

1. You are given \( n \) unit time jobs, \( J_i, i = 1, 2, \ldots, n \).
2. Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).
3. If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.

Greedy Algorithm

1. Order the jobs in descending order of profit.
2. Assume that \( p_1 \geq p_2 \ldots \geq p_n \).
3. Let \( S = \emptyset \).
4. for \( (i = 1 \text{ to } n) \) do
   5. if (\( S \cup \{J_i\} \) is feasible) then
   6. \( S \rightarrow S \cup \{J_i\} \).
The problem

1. You are given \( n \) unit time jobs, \( J_i, i = 1, 2, \ldots, n \).
2. Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).
3. If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.

Greedy Algorithm

1. Order the jobs in descending order of profit.
2. Assume that \( p_1 \geq p_2 \ldots \geq p_n \).
3. Let \( S = \emptyset \).
4. for \((i = 1 \text{ to } n)\) do
5. \hspace{1em} if \((S \cup \{J_i\} \text{ is feasible})\) then
6. \hspace{2em} \(S \rightarrow S \cup \{J_i\}\).
7. \hspace{1em} end if

Algorithm 3.27: Job scheduling
Scheduling with profits and deadlines

The problem

1. You are given \( n \) unit time jobs, \( J_i, i = 1, 2, \ldots, n \).
2. Job \( J_i \) has a deadline \( d_i \) and a profit \( p_i \).
3. If a job commences execution after its deadline, its profit is 0.

Schedule the jobs so as to maximize profit.

Greedy Algorithm

1. Order the jobs in descending order of profit.
2. Assume that \( p_1 \geq p_2 \ldots \geq p_n \).
3. Let \( S = \emptyset \).
4. for \( i = 1 \) to \( n \) do
5.   if \( (S \cup \{J_i\}) \) is feasible then
6.     \( S \rightarrow S \cup \{J_i\} \).
7.   end if
8. end for

Algorithm 3.28: Job scheduling
Correctness

Theorem

A set of jobs $S$ is feasible if and only if the sequence obtained by ordering the jobs according to nondecreasing deadlines is feasible.

Proof of correctness

Exchange argument.
Correctness

Theorem

A set of jobs $S$ is feasible if and only if the sequence obtained by ordering the jobs according to nondecreasing deadlines is feasible.

Proof of correctness

Exchange argument.
Correctness

Theorem

A set of jobs $S$ is feasible if and only if the sequence obtained by ordering the jobs according to nondecreasing deadlines is feasible.
Correctness

Theorem

A set of jobs $S$ is feasible if and only if the sequence obtained by ordering the jobs according to nondecreasing deadlines is feasible.

Proof of correctness
Correctness

**Theorem**

A set of jobs $S$ is feasible if and only if the sequence obtained by ordering the jobs according to nondecreasing deadlines is feasible.

**Proof of correctness**

Exchange argument.
The process scheduling problem

You are given a collection of processes \( P_i, i = 1, 2, \ldots, n \).

Associated with process \( P_i \) is its start time \( s_i \) and finish time \( f_i \).

Process \( P_i \) must start at \( s_i \) and is guaranteed to finish at \( f_i \).

Any machine can execute only one process at a time.

Processes \( P_i \) and \( P_j \) are said to be non-conflicting if \( f_i \leq s_j \) or \( f_j \leq s_i \).

Two processes cannot be scheduled on the same machine if they conflict.

Schedule all the processes, while minimizing the number of machines used.
The process scheduling problem

You are given a collection of processes $P_i, i = 1, 2, ..., n$.

Associated with process $P_i$ is its start time $s_i$ and finish time $f_i$.

Process $P_i$ must start at $s_i$ and is guaranteed to finish at $f_i$.

Any machine can execute only one process at a time.

Processes $P_i$ and $P_j$ are said to be non-conflicting if $f_i \leq s_j$ or $f_j \leq s_i$.

Two processes cannot be scheduled on the same machine if they conflict.

Schedule all the processes, while minimizing the number of machines used.
The process scheduling problem

The Problem

You are given a collection of processes $P_i$, $i = 1, 2, \ldots, n$. 

Associated with process $P_i$ is its start time $s_i$ and finish time $f_i$. Process $P_i$ must start at $s_i$ and is guaranteed to finish at $f_i$. Any machine can execute only one process at a time.

Processes $P_i$ and $P_j$ are said to be non-conflicting if $f_i \leq s_j$ or $f_j \leq s_i$. Two processes cannot be scheduled on the same machine if they conflict.
The process scheduling problem

The Problem

1. You are given a collection of processes $P_i, i = 1, 2, \ldots, n$.
2. Associated with process $P_i$ is its start time $s_i$ and finish time $f_i$. 

Algorithmic Insights

Computational Complexity
The process scheduling problem

1. You are given a collection of processes $P_i, i = 1, 2, \ldots, n$.
2. Associated with process $P_i$ is its start time $s_i$ and finish time $f_i$.
3. Process $P_i$ must start at $s_i$ and is guaranteed to finish at $f_i$. 

### Algorithmic Insights

#### Computational Complexity
The process scheduling problem

The Problem

1. You are given a collection of processes $P_i$, $i = 1, 2, \ldots, n$.
2. Associated with process $P_i$ is its start time $s_i$ and finish time $f_i$.
3. Process $P_i$ must start at $s_i$ and is guaranteed to finish at $f_i$.
4. Any machine can execute only one process at a time.
**The process scheduling problem**

<table>
<thead>
<tr>
<th>The Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. You are given a collection of processes $P_i$, $i = 1, 2, \ldots, n$.</td>
</tr>
<tr>
<td>2. Associated with process $P_i$ is its start time $s_i$ and finish time $f_i$.</td>
</tr>
<tr>
<td>3. Process $P_i$ must start at $s_i$ and is guaranteed to finish at $f_i$.</td>
</tr>
<tr>
<td>4. Any machine can execute only one process at a time.</td>
</tr>
<tr>
<td>5. Processes $P_i$ and $P_j$ are said to be non-conflicting if $f_i \leq s_j$ or $f_j \leq s_i$.</td>
</tr>
</tbody>
</table>
The process scheduling problem

The Problem

1. You are given a collection of processes $P_i, i = 1, 2, \ldots, n$.
2. Associated with process $P_i$ is its start time $s_i$ and finish time $f_i$.
3. Process $P_i$ must start at $s_i$ and is guaranteed to finish at $f_i$.
4. Any machine can execute only one process at a time.
5. Processes $P_i$ and $P_j$ are said to be non-conflicting if $f_i \leq s_j$ or $f_j \leq s_i$.
6. Two processes cannot be scheduled on the same machine if they conflict.
### The Problem

1. You are given a collection of processes \( P_i, i = 1, 2, \ldots, n \).
2. Associated with process \( P_i \) is its start time \( s_i \) and finish time \( f_i \).
3. Process \( P_i \) must start at \( s_i \) and is guaranteed to finish at \( f_i \).
4. Any machine can execute only one process at a time.
5. Processes \( P_i \) and \( P_j \) are said to be non-conflicting if \( f_i \leq s_j \) or \( f_j \leq s_i \).
6. Two processes cannot be scheduled on the same machine if they conflict.

Schedule all the processes, while minimizing the number of machines used.
The Greedy Approach

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots \leq s_n$.
3: for $(i = 1$ to $n)$ do
4: Assign $P_i$ to the first available machine.
5: if (no machine is available) then
6: Assign it to a new machine.
7: end if
8: end for

Correctness

Assume that the greedy approach requires $k$ machines, but that the optimal solution requires $(k - 1)$ machines.

Let process $P_I$ be the first process assigned to machine $k$ in the greedy approach.

Clearly, $P_i$ conflicts with all the processes on the first $(k - 1)$ machines.

But these processes also conflict with each other!
The Greedy Algorithm

The Greedy Approach

1. Order the processes in non-decreasing order of start time.
2. W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots \leq s_n$.
3. for $i = 1$ to $n$
   4. Assign $P_i$ to the first available machine.
   5. if (no machine is available)
      6. Assign it to a new machine.
      7. end if
   8. end for

Correctness

Assume that the greedy approach requires $k$ machines, but that the optimal solution requires $(k - 1)$ machines.

Let process $P_I$ be the first process assigned to machine $k$ in the greedy approach. Clearly, $P_i$ conflicts with all the processes on the first $(k - 1)$ machines. But these processes also conflict with each other!
Review of concepts
The Greedy Approach
Dynamic Programming

The Greedy Algorithm

The Greedy Approach

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots \leq s_n$.
3: for (i = 1 to n) do
4: Assign $P_i$ to the first available machine.
5: if (no machine is available) then
6: Assign it to a new machine.
7: end if
8: end for

Correctness
Assume that the greedy approach requires $k$ machines, but that the optimal solution requires $(k - 1)$ machines.
Let process $P_I$ be the first process assigned to machine $k$ in the greedy approach. Clearly, $P_i$ conflicts with all the processes on the first $(k - 1)$ machines. But these processes also conflict with each other!

Algorithmic Insights
Computational Complexity
**The Greedy Algorithm**

<table>
<thead>
<tr>
<th>The Greedy Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Order the processes in non-decreasing order of start time.</td>
</tr>
</tbody>
</table>
The Greedy Approach

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that \( s_1 \leq s_2 \leq \ldots s_n \).
The Greedy Algorithm

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots s_n$.
3: \textbf{for} $(i = 1 \text{ to } n) \textbf{ do}$
The Greedy Algorithm

The Greedy Approach

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots s_n$.
3: for $(i = 1 \text{ to } n)$ do
4: Assign $P_i$ to the first available machine.

Correctness
Assume that the greedy approach requires $k$ machines, but that the optimal solution requires $(k - 1)$ machines.
Let process $P_i$ be the first process assigned to machine $k$ in the greedy approach.
Clearly, $P_i$ conflicts with all the processes on the first $(k - 1)$ machines.
But these processes also conflict with each other!
The Greedy Algorithm

The Greedy Approach

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots s_n$.
3: for ($i = 1$ to $n$) do
4: Assign $P_i$ to the first available machine.
5: if (no machine is available) then

Correctness
Assume that the greedy approach requires $k$ machines, but that the optimal solution requires ($k - 1$) machines.

Let process $P_i$ be the first process assigned to machine $k$ in the greedy approach.
Clearly, $P_i$ conflicts with all the processes on the fist ($k - 1$) machines.
But these processes also conflict with each other!
The Greedy Algorithm

The Greedy Approach

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots s_n$.
3: \textbf{for (} $i = 1$ \textbf{to} $n$ \textbf{)} \textbf{do}
4: \hspace{1em} Assign $P_i$ to the first available machine.
5: \hspace{1em} \textbf{if (} no machine is available \textbf{)} \textbf{then}
6: \hspace{2em} Assign it to a new machine.

Correctness
Assume that the greedy approach requires $k$ machines, but that the optimal solution requires $(k-1)$ machines.

Let process $P_i$ be the first process assigned to machine $k$ in the greedy approach. Clearly, $P_i$ conflicts with all the processes on the first $(k-1)$ machines. But these processes also conflict with each other!
The Greedy Algorithm

The Greedy Approach

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots s_n$.
3: **for** $(i = 1 \text{ to } n)$ **do**
4: Assign $P_i$ to the first available machine.
5: **if** (no machine is available) **then**
6: Assign it to a new machine.
7: **end if**
The Greedy Algorithm

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots s_n$.
3: \textbf{for} ($i = 1$ \textbf{to} $n$) \textbf{do}
4: \hspace{1em} Assign $P_i$ to the first available machine.
5: \hspace{1em} \textbf{if} (no machine is available) \textbf{then}
6: \hspace{2em} Assign it to a new machine.
7: \hspace{1em} \textbf{end if}
8: \textbf{end for}
The Greedy Algorithm

The Greedy Approach

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that \( s_1 \leq s_2 \leq \ldots \leq s_n \).
3: \textbf{for } (i = 1 \textbf{ to } n) \textbf{ do}
4: \hspace{1em} Assign \( P_i \) to the first available machine.
5: \hspace{1em} \textbf{if } (no machine is available) \textbf{ then}
6: \hspace{2em} Assign it to a new machine.
7: \hspace{1em} \textbf{end if}
8: \hspace{1em} \textbf{end for}

Correctness
The Greedy Algorithm

The Greedy Approach

1. Order the processes in non-decreasing order of start time.
2. W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots s_n$.
3. for $(i = 1$ to $n)$ do
4. Assign $P_i$ to the first available machine.
5. if (no machine is available) then
6. Assign it to a new machine.
7. end if
8. end for

Correctness

Assume that the greedy approach requires $k$ machines, but that the optimal solution requires $(k – 1)$ machines.
The Greedy Algorithm

The Greedy Approach

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots s_n$.
3: for $(i = 1$ to $n)$ do
4: Assign $P_i$ to the first available machine.
5: if (no machine is available) then
6: Assign it to a new machine.
7: end if
8: end for

Correctness

Assume that the greedy approach requires $k$ machines, but that the optimal solution requires $(k - 1)$ machines.

Let process $P_i$ be the first process assigned to machine $k$ in the greedy approach.
The Greedy Algorithm

The Greedy Approach

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that \( s_1 \leq s_2 \leq \ldots s_n \).
3: for \( i = 1 \) to \( n \) do
   4: Assign \( P_i \) to the first available machine.
   5: if (no machine is available) then
      6: Assign it to a new machine.
   7: end if
8: end for

Correctness

Assume that the greedy approach requires \( k \) machines, but that the optimal solution requires \( (k - 1) \) machines.

Let process \( P_i \) be the first process assigned to machine \( k \) in the greedy approach.

Clearly, \( P_i \) conflicts with all the processes on the fist \( (k - 1) \) machines.
The Greedy Algorithm

The Greedy Approach

1: Order the processes in non-decreasing order of start time.
2: W.l.o.g. assume that $s_1 \leq s_2 \leq \ldots s_n$.
3: for $(i = 1$ to $n)$ do
4: Assign $P_i$ to the first available machine.
5: if (no machine is available) then
6: Assign it to a new machine.
7: end if
8: end for

Correctness

Assume that the greedy approach requires $k$ machines, but that the optimal solution requires $(k - 1)$ machines.

Let process $P_i$ be the first process assigned to machine $k$ in the greedy approach.

Clearly, $P_i$ conflicts with all the processes on the first $(k - 1)$ machines.

But these processes also conflict with each other!
The minimum weight matroid problem

**Definition**
A matroid \( M \) is a finite set \( E(M) \) together with a subset \( I(M) \) of \( 2^{E(M)} \) that satisfies the following properties:

1. \( \emptyset \in I(M) \).
2. \( Y \in I(M) \land X \subseteq Y \Rightarrow X \in I(M) \).
3. \( X, Y \in I(M) \land |Y| > |X| \Rightarrow \exists e \in Y \setminus X, \text{ such that } X \cup \{e\} \in I(M) \).

The above axioms are called independence axioms.

A maximal independent set is said to be a basis.

**The Problem**
1. Let \( E(M) = \{s_1, s_2, \ldots, s_n\} \).
2. Let \( w_i \) denote the weight of \( s_i \).

Find a basis of minimum weight.
The minimum weight matroid problem

Definition

A matroid $\mathcal{M}$ is a finite set $E(\mathcal{M})$ together with a subset $I(\mathcal{M})$ of $2^{E(\mathcal{M})}$ that satisfies the following properties:

1. $\emptyset \in I(\mathcal{M})$.
2. $(Y \in I(\mathcal{M}) \land X \subseteq Y) \Rightarrow X \in I(\mathcal{M})$.
3. $(X, Y \in I(\mathcal{M}) \land |Y| > |X|) \Rightarrow \exists e \in Y \setminus X, such that $X \cup \{e\} \in I(\mathcal{M})$.

The above axioms are called independence axioms. A maximal independent set is said to be a basis.

The Problem

1. Let $E(\mathcal{M}) = \{s_1, s_2, \ldots, s_n\}$.
2. Let $w_i$ denote the weight of $s_i$.

Find a basis of minimum weight.
The minimum weight matroid problem

Definition

A matroid $M$ is a finite set $E(M)$ together with a subset $I(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in I(M)$.
2. If $Y \in I(M)$ and $X \subseteq Y$, then $X \in I(M)$.
3. If $X, Y \in I(M)$ and $|Y| > |X|$, then there exists an $e \in Y \setminus X$ such that $X \cup \{e\} \in I(M)$.

The above axioms are called independence axioms. A maximal independent set is said to be a basis.
The minimum weight matroid problem

Definition

A matroid $M$ is a finite set $E(M)$ together with a subset $\mathcal{I}(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in \mathcal{I}(M)$.
The minimum weight matroid problem

Definition

A matroid $M$ is a finite set $E(M)$ together with a subset $I(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in I(M)$.
2. $(Y \in I(M) \wedge |Y| > |X|) \Rightarrow \exists e \in Y \setminus X, \text{ such that } X \cup \{e\} \in I(M)$.

The above axioms are called independence axioms.

A maximal independent set is said to be a basis.
The minimum weight matroid problem

Definition

A matroid \( M \) is a finite set \( E(M) \) together with a subset \( \mathcal{I}(M) \) of \( 2^{E(M)} \) that satisfies the following properties:

1. \( \emptyset \in \mathcal{I}(M) \).
2. \( (Y \in \mathcal{I}(M) \land (X \subseteq Y)) \)

Computational Complexity
Definition

A matroid $M$ is a finite set $E(M)$ together with a subset $\mathcal{I}(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in \mathcal{I}(M)$.
2. $(Y \in \mathcal{I}(M) \wedge (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M))$. 

The above axioms are called independence axioms.

The Problem

1. Let $E(M) = \{s_1, s_2, \ldots, s_n\}$.
2. Let $w_i$ denote the weight of $s_i$.

Find a basis of minimum weight.
The minimum weight matroid problem

Definition

A matroid $M$ is a finite set $E(M)$ together with a subset $I(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in I(M)$.
2. $(Y \in I(M) \land (X \subseteq Y) \Rightarrow X \in I(M)$.
3. $(X, Y \in I(M))$

Algorithmic Insights

Computational Complexity
The minimum weight matroid problem

**Definition**

A matroid $M$ is a finite set $E(M)$ together with a subset $\mathcal{I}(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in \mathcal{I}(M)$.
2. $(Y \in \mathcal{I}(M) \land (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M))$.
3. $(X, Y \in \mathcal{I}(M) \land (|Y| > |X|))$
Definition

A matroid $M$ is a finite set $E(M)$ together with a subset $I(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in I(M)$.
2. $(Y \in I(M) \land (X \subseteq Y) \Rightarrow X \in I(M)$.
3. $(X, Y \in I(M) \land (|Y| > |X|) \Rightarrow \exists e \in Y \setminus X, \text{ such that } X \cup \{e\} \in I(M)$.

The above axioms are called independence axioms. A maximal independent set is said to be a basis.

The Problem

1. Let $E(M) = \{s_1, s_2, \ldots, s_n\}$.
2. Let $w_i$ denote the weight of $s_i$.

Find a basis of minimum weight.

Algorithmic Insights

Computational Complexity
The minimum weight matroid problem

Definition

A matroid $M$ is a finite set $E(M)$ together with a subset $\mathcal{I}(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in \mathcal{I}(M)$.
2. $(Y \in \mathcal{I}(M) \land (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M))$.
3. $(X, Y \in \mathcal{I}(M) \land (|Y| > |X|) \Rightarrow \exists e \in Y \setminus X$, such that $X \cup \{e\} \in \mathcal{I}(M)$.

The above axioms are called independence axioms.
The minimum weight matroid problem

Definition

A matroid $M$ is a finite set $E(M)$ together with a subset $\mathcal{I}(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in \mathcal{I}(M)$.
2. $(Y \in \mathcal{I}(M) \land (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M))$.
3. $(X, Y \in \mathcal{I}(M) \land (|Y| > |X|) \Rightarrow \exists e \in Y \setminus X$, such that $X \cup \{e\} \in \mathcal{I}(M)$.

The above axioms are called independence axioms.

A maximal independent set is said to be a basis.
The minimum weight matroid problem

Definition

A matroid $M$ is a finite set $E(M)$ together with a subset $\mathcal{I}(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in \mathcal{I}(M)$.
2. $(Y \in \mathcal{I}(M) \land (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M))$.
3. $(X, Y \in \mathcal{I}(M) \land (|Y| > |X|) \Rightarrow \exists e \in Y \setminus X, \text{ such that } X \cup \{e\} \in \mathcal{I}(M)$.

The above axioms are called independence axioms.

A maximal independent set is said to be a basis.

The Problem
The minimum weight matroid problem

Definition

A matroid $M$ is a finite set $E(M)$ together with a subset $I(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in I(M)$.
2. $(Y \in I(M) \land (X \subseteq Y)) \Rightarrow X \in I(M)$.
3. $(X, Y \in I(M) \land (|Y| > |X|)) \Rightarrow \exists e \in Y \setminus X$, such that $X \cup \{e\} \in I(M)$.

The above axioms are called independence axioms.

A maximal independent set is said to be a basis.

The Problem

1. Let $E(M) = \{s_1, s_2, \ldots, s_n\}$. 
The minimum weight matroid problem

**Definition**

A matroid $M$ is a finite set $E(M)$ together with a subset $\mathcal{I}(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in \mathcal{I}(M)$.
2. $(Y \in \mathcal{I}(M) \land (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M))$.
3. $(X, Y \in \mathcal{I}(M) \land (|Y| > |X|) \Rightarrow \exists e \in Y \setminus X$, such that $X \cup \{e\} \in \mathcal{I}(M)$.

The above axioms are called independence axioms.

A maximal independent set is said to be a basis.

**The Problem**

1. Let $E(M) = \{s_1, s_2, \ldots, s_n\}$.
2. Let $w_i$ denote the weight of $s_i$. 

**Algorithmic Insights**

**Computational Complexity**
The minimum weight matroid problem

Definition

A matroid $M$ is a finite set $E(M)$ together with a subset $I(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in I(M)$.
2. $(Y \in I(M) \land (X \subseteq Y) \Rightarrow X \in I(M)$.
3. $(X, Y \in I(M) \land (|Y| > |X|) \Rightarrow \exists e \in Y \setminus X$, such that $X \cup \{e\} \in I(M)$.

The above axioms are called independence axioms.

A maximal independent set is said to be a basis.

The Problem

1. Let $E(M) = \{s_1, s_2, \ldots, s_n\}$.
2. Let $w_i$ denote the weight of $s_i$.

Find a basis of minimum weight.
The matroid lemma

Lemma

Let $S$ be a set where the family of independent sets forms a matroid. Suppose an independent set $F$ is contained in a minimum-weight basis. Let $v$ be one of the lightest elements of $S$ such that $F \cup \{v\}$ is also independent. Then $F \cup \{v\}$ is also contained in a minimum-weight basis.
The matroid lemma

**Lemma**

Let $S$ be a set where the family of independent sets forms a matroid. Suppose an independent set $F$ is contained in a minimum-weight basis. Let $v$ be one of the lightest elements of $S$ such that $F \cup \{v\}$ is also independent. Then $F \cup \{v\}$ is also contained in a minimum-weight basis.
The matroid lemma

Lemma

Let $S$ be a set where the family of independent sets forms a matroid.
The matroid lemma

Lemma

Let $S$ be a set where the family of independent sets forms a matroid. Suppose an independent set $F$ is contained in a minimum-weight basis.
Lemma

Let $S$ be a set where the family of independent sets forms a matroid. Suppose an independent set $F$ is contained in a minimum-weight basis. Let $v$ be one of the lightest elements of $S$ such that $F \cup \{v\}$ is also independent.
The matroid lemma

Lemma

Let $S$ be a set where the family of independent sets forms a matroid. Suppose an independent set $F$ is contained in a minimum-weight basis. Let $v$ be one of the lightest elements of $S$ such that $F \cup \{v\}$ is also independent. Then $F \cup \{v\}$ is also contained in a minimum-weight basis.
Dynamic Programming

Main ideas
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Algorithmic Insights
Computational Complexity
Dynamic Programming

Main ideas

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.
Main ideas

1. Characterize the structure of an optimal solution.
Main ideas

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
# Dynamic Programming

### Main ideas

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
Main ideas

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.
The Rod Cutting problem

Given a rod of \( n \) inches, and a table of prices \( p_1, p_2, \ldots, p_n \), determine the maximum revenue \( r_n \) obtainable by cutting up the rod and selling it into pieces.

How many possibilities?

Example

<table>
<thead>
<tr>
<th>Length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>Price</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
</tr>
</tbody>
</table>

Compute \( r_1, r_2, \ldots, r_6 \).
The Rod Cutting problem

The Problem

The Problem

The Problem
The Problem

Given a rod of $n$ inches, and a table of prices $p_i$, $i = 1, 2, \ldots, n$, determine the maximum revenue $r_n$ obtainable by cutting up the rod and selling it into pieces.
The Rod Cutting problem

The Problem

Given a rod of \( n \) inches, and a table of prices \( p_i, i = 1, 2, \ldots, n \), determine the maximum revenue \( r_n \) obtainable by cutting up the rod and selling it into pieces. How many possibilities?
The Rod Cutting problem

The Problem

Given a rod of \( n \) inches, and a table of prices \( p_i, \ i = 1, 2, \ldots, n \), determine the maximum revenue \( r_n \) obtainable by cutting up the rod and selling it into pieces. How many possibilities?

Example
The Rod Cutting problem

The Problem

Given a rod of \( n \) inches, and a table of prices \( p_i, i = 1, 2, \ldots, n \), determine the maximum revenue \( r_n \) obtainable by cutting up the rod and selling it into pieces. How many possibilities?

Example

<table>
<thead>
<tr>
<th>Length ( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price ( p_i )</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
</tr>
</tbody>
</table>
The Rod Cutting problem

The Problem

Given a rod of $n$ inches, and a table of prices $p_i$, $i = 1, 2, \ldots, n$, determine the maximum revenue $r_n$ obtainable by cutting up the rod and selling it into pieces. How many possibilities?

Example

<table>
<thead>
<tr>
<th>Length $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price $p_i$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
</tr>
</tbody>
</table>

Compute $r_i$, $i = 1, 2, \ldots 6$. 
Review of concepts

The Greedy Approach

Dynamic Programming

Optimal substructure property

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?) This is called the optimal substructure property. Hence, we can write,

$$ r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1) \quad (1) $$

Unlike Divide-and-Conquer, the subproblems could overlap. Recurrence (1) can be expressed more succinctly as:

$$ r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \quad (2) $$

$$ r_0 = 0 $$

Why are Recurrence (1) and Recurrence (2) equivalent?
Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. This is called the optimal substructure property. Hence, we can write,

\[ r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) \]  

Unlike Divide-and-Conquer, the subproblems could overlap. Recurrence (1) can be expressed more succinctly as:

\[ r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \]  

Why are Recurrence (1) and Recurrence (2) equivalent?
Optimal substructure property

Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally.

$$r_n = \max \left( p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1 \right)$$

Unlike Divide-and-Conquer, the subproblems could overlap. Recurrence (1) can be expressed more succinctly as:

$$r_n = \max_{1 \leq i \leq n} \left( p_i + r_{n-i} \right)$$

$$r_0 = 0$$

Why are Recurrence (1) and Recurrence (2) equivalent?
Optimal substructure property

Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)
Optimal substructure property

Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property.
Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,
Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,

\[ r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1). \]  

(1)
Optimal substructure property

Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,

\[ r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots r_{n-1} + r_1). \] (1)

Unlike Divide-and-Conquer, the subproblems could overlap.
Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,

\[ r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1). \]  

Unlike Divide-and-Conquer, the subproblems could overlap.

Recurrence (1) can be expressed more succinctly as:

\[ r_n = \max_{1 \leq i \leq n}(p_i + r_{n-i}). \]  

Why are Recurrence (1) and Recurrence (2) equivalent?
Optimal substructure property

Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,

\[ r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1). \] (1)

Unlike Divide-and-Conquer, the subproblems could overlap.

Recurrence (1) can be expressed more succinctly as:

\[ r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \] (2)

\[ r_0 = 0 \]
### Optimal substructure property

#### Recurrence

Observe that once the first cut is made, you get two independent subproblems which must be solved optimally. (Why?)

This is called the optimal substructure property. Hence, we can write,

\[ r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1). \]  

(1)

Unlike Divide-and-Conquer, the subproblems could overlap.

Recurrence (1) can be expressed more succinctly as:

\[ r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \]  

(2)

\[ r_0 = 0 \]

Why are Recurrence (1) and Recurrence (2) equivalent?
A recursive implementation

Algorithm 4.1: The recursive rod-cutting algorithm

Analysis

\[ T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{j=1}^{n} T(n-j), & \text{otherwise} 
\end{cases} \]
A recursive implementation

Recursive Algorithm

Function

\begin{align*}
CUT-ROD(p, n) =
& \begin{cases}
0, & \text{if } n = 0 \\
\max_{i=1}^{n} \left( p[i] + \text{CUT-ROD}(p, n-i) \right), & \text{otherwise}
\end{cases}
\end{align*}

Algorithm 4.2: The recursive rod-cutting algorithm

Analysis

\[ T(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1 + \sum_{j=1}^{n} T(n-j), & \text{otherwise} \end{cases} \]
A recursive implementation

Recursive Algorithm

Function $CUT-ROD(p, n)$

1: if $(n = 0)$ then
2: return $(0)$.
3: end if
4: $q = -\infty$.
5: for $(i = 1$ to $n)$ do
6: $q = \max(q, p[i] + CUT-ROD(p, n - i))$.
7: end for

Algorithm 4.3: The recursive rod-cutting algorithm

Analysis

$T(n) =$ \begin{cases} 1 & \text{if } n = 0 \\ 1 + \sum_{j=1}^{n} T(n-j) & \text{otherwise} \end{cases}$
A recursive implementation

Recursive Algorithm

Function \textsc{Cut-Rod}(p, n)

```plaintext
1: if (n = 0) then
2: return (0).
3: end if
4: q = −∞.
5: for (i = 1 to n) do
6: q = max(q, p[i] + \textsc{Cut-Rod}(p, n−i)).
7: end for
8: Algorithm 4.4: The recursive rod-cutting algorithm
9: Analysis
10: T(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1 + \sum_{j=1}^{n} T(n−j), & \text{otherwise} \end{cases}
```
Recursive Algorithm

**Function** \text{\textit{CUT-ROD}}(p, n)

1: \textbf{if } \textit{(n = 0)} \textbf{then}
A recursive implementation

### Recursive Algorithm

**Function** `CUT-ROD(p, n)`

1. if $(n = 0)$ then
2. return(0).
A recursive implementation

### Recursive Algorithm

**Function** `CUT-ROD(p, n)`

1. if \( n = 0 \) then
2.   return(0).
3. end if
A recursive implementation

Recursive Algorithm

Function `CUT-ROD(p, n)`

1: if \( n = 0 \) then
2:   return 0.
3: end if
4: \( q = -\infty \).
A recursive implementation

Recursive Algorithm

Function `CUT-ROD(p, n)`

1: if \( n = 0 \) then
2:    return 0.
3: end if
4: \( q = -\infty \).
5: for \( i = 1 \) to \( n \) do

Algorithm 4.9: The recursive rod-cutting algorithm

Analysis

\[ T(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1 + \sum_{j=1}^{n} T(n-j), & \text{otherwise} \end{cases} \]
A recursive implementation

**Recursive Algorithm**

**Function** \texttt{CUT-ROD}(p, n)

1: \textbf{if} \ (n = 0) \ \textbf{then}
2: \hspace{1em} \textbf{return}(0).
3: \textbf{end if}
4: \texttt{q} = -\infty.
5: \textbf{for} \ (i = 1 \ \textbf{to} \ n) \ \textbf{do}
6: \hspace{1em} q = \max(q, \ldots)}
A recursive implementation

Recursive Algorithm

Function $\text{CUT-ROD}(p, n)$

1: if $(n = 0)$ then
2: \hspace{1em} return(0).
3: end if
4: $q = -\infty$.
5: for $(i = 1$ to $n)$ do
6: \hspace{1em} $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$.

Algorithm 4.11: The recursive rod-cutting algorithm

Analysis

$T(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1 + \sum_{j=1}^{n} T(n-j), & \text{otherwise} \end{cases}$

Algorithmic Insights

Computational Complexity
A recursive implementation

### Recursive Algorithm

**Function** \( \text{CUT-ROD}(p, n) \)

1. if \((n = 0)\) then
2. \hspace{1em} return(0).
3. end if
4. \(q = -\infty.\)
5. for \((i = 1 \text{ to } n)\) do
6. \hspace{1em} \(q = \max(q, p[i] + \text{CUT-ROD}(p, n - i)).\)
7. end for

**Algorithm 4.12:** The recursive rod-cutting algorithm
A recursive implementation

Recursive Algorithm

Function $\text{CUT-ROD}(p, n)$
1: if $(n = 0)$ then
2: return(0).
3: end if
4: $q = -\infty$.
5: for $i = 1$ to $n$ do
6: $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$.
7: end for

Algorithm 4.13: The recursive rod-cutting algorithm

Analysis
A recursive implementation

Recursive Algorithm

```plaintext
Function CUT-ROD(p, n)
1: if (n = 0) then
2:    return(0).
3: end if
4: q = −∞.
5: for (i = 1 to n) do
6:    q = max(q, p[i] + CUT-ROD(p, n − i)).
7: end for
```

**Algorithm 4.14:** The recursive rod-cutting algorithm

Analysis

\[ T(n) = \]
A recursive implementation

Recursive Algorithm

Function $\text{CUT-ROD}(p, n)$

1: if $(n = 0)$ then
2:     return(0).
3: end if
4: $q = -\infty$.
5: for $(i = 1$ to $n)$ do
6:     $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$.
7: end for

Algorithm 4.15: The recursive rod-cutting algorithm

Analysis

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \end{cases}$$
A recursive implementation

**Recursive Algorithm**

```plaintext
Function CUT-ROD(p, n)
1: if (n = 0) then
2:   return(0).
3: end if
4: q = −∞.
5: for (i = 1 to n) do
6:   q = max(q, p[i] + CUT-ROD(p, n − i)).
7: end for
```

**Algorithm 4.16:** The recursive rod-cutting algorithm

**Analysis**

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{j=1}^{n} T(n - j), & \text{otherwise}
\end{cases}
\]
Analysis of the recursive algorithm

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise}
\end{cases}
\]

It is not hard to see that \( T(n) = 2^n \).
Analysis of the recursive algorithm

It is not hard to see that

\[ T(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise} \end{cases} \]

It follows that \( T(n) = 2^n \).
Analysis of the recursive algorithm

\[ T(n) = \left\{ \begin{array}{ll}
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise}
\end{array} \right. \]

It is not hard to see that \( T(n) = 2^n \).

Algorithmic Insights
Computational Complexity
Analysis of the recursive algorithm

Analysis (contd.)

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise}
\end{cases}
\]
Analysis of the recursive algorithm

\[ T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise}
\end{cases} \]
Analysis of the recursive algorithm

Analysis (contd.)

\[ T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise}
\end{cases} \]

It is not hard to see that \( T(n) = \)
Analysis of the recursive algorithm

Analysis (contd.)

\[ T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
1 + \sum_{k=0}^{n-1} T(k), & \text{otherwise}
\end{cases} \]

It is not hard to see that \( T(n) = 2^n \).
The Bottom-up Approach

The bottom-up algorithm

**Algorithm 4.17: Bottom-up rod-cutting**

1. Let \( r[0 \ldots n] \) be a new array.
2. \( r[0] = 0 \).
3. for \((j = 1 \text{ to } n)\) do
   4. \( q = -\infty \).
   5. for \((i = 1 \text{ to } j)\) do
      6. \( q = \max(q, p[i] + r[j - i]) \).
   7. \( r[j] = q \).
4. return \( r[n] \).
The bottom-up algorithm

1: Let \( r[0 \ldots n] \) be a new array.
2: \( r[0] = 0 \).
3: for \( j = 1 \) to \( n \)
   4: \( q = -\infty \).
   5: for \( i = 1 \) to \( j \)
      6: \( q = \max(q, p[i] + r[j-i]) \).
   7: \( r[j] = q \).
4: end for
5: return \( r[n] \).
The Bottom-up approach

**The bottom-up algorithm**

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>BOTTOM-ROD-CUT</strong>*(p, n)*</th>
</tr>
</thead>
</table>

```python
Let r[0...n] be a new array.

r[0] = 0.

for (j = 1 to n)

q = -∞.

for (i = 1 to j)

q = max(q, p[i] + r[j − i])

r[j] = q.

return r[n].
```
The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)
1: Let r[0 · · · n] be a new array.
Function \textsc{Bottom-Rod-Cut}(p, n)

1. Let $r[0 \cdots n]$ be a new array.
2. $r[0] = 0$. 

The bottom-up algorithm

\textbf{Function} \textsc{Bottom-Rod-Cut}(p, n)

1. Let $r[0 \cdots n]$ be a new array.
2. $r[0] = 0$. 

The Bottom-up approach

The bottom-up algorithm

**Function** `BOTTOM-ROD-CUT(p, n)`

1. Let `r[0 · · · n]` be a new array.
2. `r[0] = 0`.
3. `for (j = 1 to n) do`
The bottom-up algorithm

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let \( r[0 \ldots n] \) be a new array.
2: \( r[0] = 0 \).
3: **for** \( j = 1 \) to \( n \) **do**
4: \( q = -\infty \).
The Bottom-up approach

The bottom-up algorithm

Function $\text{BOTTOM-ROD-CUT}(p, n)$

1: Let $r[0 \cdots n]$ be a new array.
2: $r[0] = 0$.
3: for $(j = 1$ to $n)$ do
4: $q = -\infty$.
5: for $(i = 1$ to $j)$ do
The Bottom-up approach

The bottom-up algorithm

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let `r[0 · · · n]` be a new array.
2: `r[0] = 0`.
3: `for (j = 1 to n) do`
   4: `q = −∞`.
   5: `for (i = 1 to j) do`
   6: `q = max(q, p[i] + r[j − i]).`
The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)
1: Let r[0 · · · n] be a new array.
2: r[0] = 0.
3: for (j = 1 to n) do
4:   q = −∞.
5:   for (i = 1 to j) do
6:     q = max(q, p[i] + r[j − i]).
7: end for
8: r[j] = q.
9: end for
10: return r[n].
The bottom-up algorithm

<table>
<thead>
<tr>
<th>Function</th>
<th>BOTTOM-ROD-CUT(p, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>Let $r[0 \cdots n]$ be a new array.</td>
</tr>
<tr>
<td>2:</td>
<td>$r[0] = 0$.</td>
</tr>
<tr>
<td>3:</td>
<td>for ($j = 1$ to $n$) do</td>
</tr>
<tr>
<td>4:</td>
<td>$q = -\infty$.</td>
</tr>
<tr>
<td>5:</td>
<td>for ($i = 1$ to $j$) do</td>
</tr>
<tr>
<td>6:</td>
<td>$q = \max(q, p[i] + r[j - i])$.</td>
</tr>
<tr>
<td>7:</td>
<td>end for</td>
</tr>
<tr>
<td>8:</td>
<td>$r[j] = q$.</td>
</tr>
</tbody>
</table>
The Bottom-up approach

The bottom-up algorithm

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let $r[0 \cdots n]$ be a new array.
2: $r[0] = 0.$
3: **for** $(j = 1$ to $n)$ **do**
4: \hspace{1em} $q = -\infty.$
5: \hspace{1em} **for** $(i = 1$ to $j)$ **do**
6: \hspace{2em} $q = \max(q, p[i] + r[j - i]).$
7: \hspace{1em} **end for**
8: \hspace{1em} $r[j] = q.$
9: **end for**
The Bottom-up approach

The bottom-up algorithm

**Function** \textsc{Bottom-Rod-Cut}(p, n)

1: Let \(r[0 \cdots n]\) be a new array. 
2: \(r[0] = 0.\)
3: \textbf{for} \ (j = 1 \textbf{to} n) \ 	extbf{do}
4: \hspace{1em} \(q = -\infty.\)
5: \hspace{1em} \textbf{for} \ (i = 1 \textbf{to} j) \ 	extbf{do}
6: \hspace{2em} \(q = \max(q, p[i] + r[j-i]).\)
7: \hspace{1em} \textbf{end for}
8: \hspace{1em} \(r[j] = q.\)
9: \hspace{1em} \textbf{end for}
10: \textbf{return} \(r[n].\)

**Algorithm 4.29:** Bottom-up rod-cutting
Analyzing the bottom-up approach

The running time of the algorithm can be approximated by the number of times that Line (6) is executed.

Accordingly, \( T(n) = \begin{cases} 
0, & \text{if } n = 0 \\
\sum_{j=1}^{n-1} \sum_{i=1}^{j} 1, & \text{otherwise} 
\end{cases} \)

It is not hard to see that \( T(n) = \Theta(n^2) \).
Analyzing the bottom-up approach

Analysis

The running time of the algorithm can be approximated by the number of times that Line (6) is executed. Accordingly, 

$$T(n) = \begin{cases} 
0, & \text{if } n = 0 \\
\sum_{j=1}^{n} \sum_{i=1}^{j} 1, & \text{otherwise} 
\end{cases}$$

It is not hard to see that $$T(n) = \Theta(n^2).$$
Analyzing the bottom-up approach

Analysis

The running time of the algorithm can be approximated by the number of times that Line (6) is executed.
Analysis

The running time of the algorithm can be approximated by the number of times that Line (6) is executed.

Accordingly,

\[ T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} 1 \]
The running time of the algorithm can be approximated by the number of times that Line (6) is executed.

Accordingly,

$$T(n) = \begin{cases} 
0, & \text{if } n = 0 
\end{cases}$$
Analyzing the bottom-up approach

Analysis

The running time of the algorithm can be approximated by the number of times that Line (6) is executed.

Accordingly,

$$T(n) = \begin{cases} 0, & \text{if } n = 0 \\ \sum_{j=1}^{n} \sum_{i=1}^{j} 1, & \text{otherwise} \end{cases}$$
Analyzing the bottom-up approach

Analysis

The running time of the algorithm can be approximated by the number of times that Line (6) is executed.

Accordingly,

\[
T(n) = \begin{cases} 
0, & \text{if } n = 0 \\
\sum_{j=1}^{n} \sum_{i=1}^{j} 1, & \text{otherwise}
\end{cases}
\]

It is not hard to see that \( T(n) = \) \( \Theta(n^2) \).
Analyzing the bottom-up approach

Analysis

The running time of the algorithm can be approximated by the number of times that Line (6) is executed.

Accordingly,

\[ T(n) = \begin{cases} 
0, & \text{if } n = 0 \\
\sum_{j=1}^{n} \sum_{i=1}^{j} 1, & \text{otherwise} 
\end{cases} \]

It is not hard to see that \( T(n) = \Theta(n^2) \).
Reconstructing the Solution

Review of concepts
The Greedy Approach
Dynamic Programming

Algorithm 4.30: Bottom-up rod-cutting

1: Let \( r[0 \cdots n] \) and \( s[0 \cdots n] \) be new arrays.
2: \( r[0] = 0. \)
3: for \( (j = 1 \text{ to } n) \) do
4: \( q = -\infty. \)
5: for \( (i = 1 \text{ to } j) \) do
6: if \( (q < p[i] + r[j - i]) \) then
7: \( q = p[i] + r[j - i]. \)
8: \( s[j] = i. \) {The unsplittable left side is recorded.}
9: end if
10: end for
11: \( r[j] = q. \)
12: end for
13: return \( (r[n]). \)
Reconstructing the Solution

The bottom-up algorithm with solution

```
Function BOTTOM-ROD-CUTTING (p, n)
1: Let r[0··n] and s[0··n] be new arrays.
2: r[0] = 0.
3: for (j = 1 to n)
4:   q = −∞.
5:   for (i = 1 to j)
6:     if (q < p[i] + r[j−i])
7:       q = p[i] + r[j−i].
8:       s[j] = i. {The unsplittable left side is recorded.}
9:   end if
10: end for
11: r[j] = q.
12: end for
13: return (r[n]).
```

Algorithm 4.31: Bottom-up rod-cutting

Algorithmic Insights

Computational Complexity
Reconstructing the Solution

The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`
### The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1. Let $r[0 \cdots n]$ and $s[0 \cdots n]$ be new arrays.
Reconstructing the Solution

The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let \( r[0 \cdot n] \) and \( s[0 \cdot n] \) be new arrays.
2: \( r[0] = 0 \).
The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1. Let `r[0 · n]` and `s[0 · n]` be new arrays.
2. `r[0] = 0`.
3. **for** `(j = 1 to n)` **do**
The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1. Let $r[0 \cdot n]$ and $s[0 \cdot n]$ be new arrays.
2. $r[0] = 0$.
3. **for** $(j = 1$ to $n)$ **do**
4. $q = -\infty$.
Algorithmic Insights

Computational Complexity

The bottom-up algorithm with solution

Function **BOTTOM-ROD-CUT**(p, n)

1: Let \( r[0 \cdots n] \) and \( s[0 \cdots n] \) be new arrays.
2: \( r[0] = 0 \).
3: for \( j = 1 \) to \( n \) do
   4: \( q = -\infty \).
   5: for \( i = 1 \) to \( j \) do
   6: if \( q < p[i] + r[j - i] \) then
      7: \( q = p[i] + r[j - i] \).
      8: \( s[j] = i \). \{ The unsplittable left side is recorded. \}
   9: end if
10: end for
11: \( r[j] = q \).
12: end for
13: return \( r[n] \).
The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let `r[0 · · n]` and `s[0 · · n]` be new arrays.
2: `r[0] = 0`.
3: **for** `(j = 1` **to** `n) do`
4: `q = −∞`.
5: **for** `(i = 1` **to** `j) do`
6: **if** `(q < p[i] + r[j − i])` **then**
Reconstructing the Solution

The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let `r[0 · · n]` and `s[0 · · n]` be new arrays.
2: `r[0] = 0.`
3: **for** `(j = 1 to n) do`
4: `q = −∞.`
5: **for** `(i = 1 to j) do`
6: `if (q < p[i] + r[j − i]) then`
7: `q = p[i] + r[j − i].`

Algorithm 4.39: Bottom-up rod-cutting
The bottom-up algorithm with solution

**Function** \textsc{Bottom-Rod-Cut}(p, n)

1. Let \( r[0 \cdots n] \) and \( s[0 \cdots n] \) be new arrays.
2. \( r[0] = 0. \)
3. \textbf{for} \hspace{1em} (\( j = 1 \) to \( n \)) \textbf{do}
4. \hspace{1em} \( q = -\infty. \)
5. \hspace{1em} \textbf{for} \hspace{1em} (\( i = 1 \) to \( j \)) \textbf{do}
6. \hspace{2em} \textbf{if} \hspace{1em} (\( q < p[i] + r[j - i] \)) \textbf{then}
7. \hspace{3em} \( q = p[i] + r[j - i]. \)
8. \hspace{3em} \( s[j] = i. \) \{The unsplittable left side is recorded.\}
The bottom-up algorithm with solution

**Function** BOTTOM-ROD-CUT(p, n)

1: Let r[0 · · n] and s[0 · · n] be new arrays.
2: r[0] = 0.
3: for (j = 1 to n) do
4:   q = −∞.
5:   for (i = 1 to j) do
6:     if (q < p[i] + r[j − i]) then
7:       q = p[i] + r[j − i].
8:       s[j] = i. {The unsplittable left side is recorded.}
9:     end if
10: end for
11: r[j] = q.
12: end for
13: return (r[n]).
The bottom-up algorithm with solution

Function `BOTTOM-ROD-CUT(p, n)`

1: Let r[0 · · n] and s[0 · · n] be new arrays.
2: r[0] = 0.
3: for (j = 1 to n) do
4:     q = −∞.
5:     for (i = 1 to j) do
6:         if (q < p[i] + r[j − i]) then
7:             q = p[i] + r[j − i].
8:             s[j] = i. {The unsplittable left side is recorded.}
9:         end if
10:    end for
11: r[j] = q.
12: end for
13: return (r[n]).
The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let $r[0 \cdot n]$ and $s[0 \cdot n]$ be new arrays.
2: $r[0] = 0$.
3: for $(j = 1$ to $n)$ do
4:    $q = -\infty$.
5:    for $(i = 1$ to $j)$ do
6:       if $(q < p[i] + r[j - i])$ then
7:          $q = p[i] + r[j - i]$.
8:          $s[j] = i$. \{The unsplittable left side is recorded.\}
9:    end if
10: end for
11: $r[j] = q$. 
Reconstructing the Solution

The bottom-up algorithm with solution

Function `BOTTOM-ROD-CUT(p, n)`

1: Let $r[0 \cdot n]$ and $s[0 \cdot n]$ be new arrays.
2: $r[0] = 0$.
3: **for** $(j = 1$ to $n)$ **do**
4:     $q = -\infty$.
5:     **for** $(i = 1$ to $j)$ **do**
6:         if $(q < p[i] + r[j - i])$ **then**
7:             $q = p[i] + r[j - i]$.
8:             $s[j] = i$.  {The unsplittable left side is recorded.}
9:         **end if**
10:     **end for**
11: $r[j] = q$.
12: **end for**

13: **return** $(r[n])$.  

Algorithm 4.44: Bottom-up rod-cutting
The bottom-up algorithm with solution

**Function** `BOTTOM-ROD-CUT(p, n)`

1: Let $r[0 \cdots n]$ and $s[0 \cdots n]$ be new arrays.
2: $r[0] = 0$.
3: **for** $(j = 1$ to $n) \text{ do}$
4: $q = -\infty$.
5: **for** $(i = 1$ to $j) \text{ do}$
6: \hspace{1em} **if** $(q < p[i] + r[j - i]) \text{ then}$
7: \hspace{2em} $q = p[i] + r[j - i]$.
8: \hspace{2em} $s[j] = i$. \{The unsplittable left side is recorded.\}
9: \hspace{1em} **end if**
10: **end for**
11: $r[j] = q$.
12: **end for**
13: **return**$(r[n])$.

**Algorithm 4.45:** Bottom-up rod-cutting
Outputting the solution

Algorithm 4.46: Extracting the solution

```
function PRINT-SOLUTION(p, n)
1: while (n > 0) do
2: print s[n].
3: n = n - s[n].
4: end while
```

Algorithmic Insights

Computational Complexity
Outputting the solution

Printing the Solution

```
function PRINT-SOLUTION(p, n)
1: while (n > 0) do
2:   print s[n].
3:   n = n - s[n].
4: end while
```

Algorithm 4.47: Extracting the solution
Outputting the solution

**Function** `PRINT-SOLUTION(p, n)`

```
1: while (n > 0) do
2:   print s[n].
3:   n = n - s[n].
4: end while
```

Algorithm 4.48: Extracting the solution

**Algorithmic Insights**

**Computational Complexity**
Printing the Solution

**Function** PRINT-SOLUTION(p, n)

1: while (n > 0) do
**Function** PRINT-SOLUTION(p, n)

1: while \((n > 0)\) do
2:   print \(s[n]\).
Outputting the solution

Printing the Solution

Function PRINT-SOLUTION(p, n)
1: while (n > 0) do
2:    print s[n].
3:    n = n − s[n].
Outputting the solution

Printing the Solution

Function $\text{PRINT-SOLUTION}(p, n)$

1. while $(n > 0)$ do
2. print $s[n]$.
3. $n = n - s[n]$.
4. end while

Algorithm 4.52: Extracting the solution
The Matrix Chain Multiplication problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_i-1 \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.

2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders

$$T(n) = \begin{cases} 0, & \text{if } n = 0 \\ \sum_{k=1}^{n-1} T(k) \cdot T(n-k), & \text{otherwise} \end{cases}$$

Solving the recurrence gives the $n$th Catalan number whose growth is $\Omega(4^n n^{3/2})$.
You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_i \times d_i$, while minimizing the number of scalar multiplications.

Observe that, the total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.

The entries in the matrices do not affect the optimum solution.

Solving the recurrence gives the $n$th Catalan number whose growth is $\Omega(4^n n^{3/2})$. 

Algorithmic Insights

Computational Complexity
The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$. 

<table>
<thead>
<tr>
<th>Algorithmic Insights</th>
<th>Computational Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Review of concepts</td>
<td></td>
</tr>
<tr>
<td>The Greedy Approach</td>
<td></td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td></td>
</tr>
</tbody>
</table>
The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.
The Matrix Chain Multiplication problem

The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,
The Matrix Chain Multiplication problem

The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$. 

The Matrix Chain Multiplication problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.
The Matrix Chain Multiplication problem

The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders

Solving the recurrence gives the $n$th Catalan number whose growth is $\Omega(4^n n^{3/2})$. 
The Matrix Chain Multiplication problem

The Problem
You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders
The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders

$$T(n) =$$
The Matrix Chain Multiplication problem

The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders

$$T(n) = \begin{cases} 0, & \text{if } n = 0 \\ \sum_{k=1}^{n-1} T(k) \cdot T(n-k), & \text{otherwise} \end{cases}$$
The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders

$$T(n) = \begin{cases} 
0, & \text{if } n = 0 \\
\sum_{k=1}^{n-1} T(k) \cdot T(n - k), & \text{otherwise}
\end{cases}$$

Algorithmic Insights

Computational Complexity
The Matrix Chain Multiplication problem

The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$, while minimizing the number of scalar multiplications.

Observe that,

1. The total number of scalar multiplications when multiplying two matrices of dimensions $p \times q$ and $q \times r$ is $p \cdot q \cdot r$.
2. The entries in the matrices do not affect the optimum solution.

Cost of enumerating all the orders

$$T(n) = \begin{cases} 
0, & \text{if } n = 0 \\
\sum_{k=1}^{n-1} T(k) \cdot T(n - k), & \text{otherwise}
\end{cases}$$

Solving the recurrence gives the $n^{th}$ Catalan number whose growth is $\Omega\left(\frac{4^n}{n^2}\right)$. 

Algorithmic Insights  Computational Complexity
Optimality Substructure

If somebody gave you the first grouping, can the problem be simplified? Yes!
The two subproblems that result must be solved optimally. Why?

Therefore, the optimality substructure applies.

Let $m[i,j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots, A_j \rangle$.

$m[i,j] = \begin{cases} 0, & \text{if } j = i \\ \min_{i \leq k < j} (m[i,k] + m[k+1,j] + d_{i-1} \cdot d_k \cdot d_{j-1}), & \text{if } j > i. \end{cases}$
If somebody gave you the first grouping, can the problem be simplified? Yes! The two subproblems that result must be solved optimally. Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots, A_j \rangle$.

$$m[i, j] = \begin{cases} 0, & \text{if } j = i \\ \min_{i \leq k < j} (m[i, k] + m[k+1, j] + d_{i-1} \cdot d_k \cdot d_{j-1}), & \text{if } j > i. \end{cases}$$
If somebody gave you the first grouping, can the problem be simplified?
Optimality Substructure

Substructure

If somebody gave you the **first** grouping, can the problem be simplified?

Yes!
Substructure

If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally.
Optimality Substructure

Substructure

If somebody gave you the **first** grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)
If somebody gave you the **first** grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.
Substructure

If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots A_j \rangle$. 
Substructure

If somebody gave you the first grouping, can the problem be simplified?
Yes! The two subproblems that result must be solved optimally. (Why?)
Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots A_j \rangle$.

$m[i, j] = \begin{cases} 
0, & \text{if } j = i \\
\min_{i \leq k < j} (m[i, k] + m[k+1, j] + d_{i-1} \cdot d_k \cdot d_{j-1}), & \text{if } j > i.
\end{cases}$
Substructure

If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots A_j \rangle$.

$$
m[i, j] = \begin{cases} 
0, & \text{if } j = i \\
\min_{i \leq k < j} (m[i, k] + m[k+1, j] + d_{i-1} \cdot d_k \cdot d_{j-1}), & \text{if } j > i.
\end{cases}
$$
If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.

Let $m[i,j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots A_j \rangle$.

$$m[i,j] = \begin{cases} 0, & \text{if } j = i \\ m[i,k] + m[k+1,j] + d_i - 1 \cdot d_k \cdot d_j, & \text{if } j > i. \end{cases}$$
Substructure

If somebody gave you the first grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots A_j \rangle$.

\[
m[i, j] = \begin{cases} 
0, & \text{if } j = i \\
\min_{i \leq k < j} (m[i, k] + m[k + 1, j]) & \text{if } j > i
\end{cases}
\]
If somebody gave you the first grouping, can the problem be simplified?
Yes! The two subproblems that result must be solved optimally. (Why?)
Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots A_j \rangle$.

$$m[i, j] = \begin{cases} 
0, & \text{if } j = i \\
\min_{i \leq k < j} (m[i, k] + m[k + 1, j] + d_{i - 1} \cdot d_k \cdot d_j), & \text{if } j > i
\end{cases}$$
If somebody gave you the **first** grouping, can the problem be simplified?

Yes! The two subproblems that result must be solved optimally. (Why?)

Therefore, the optimality substructure applies.

Let $m[i, j]$ denote the optimal number of scalar multiplications to multiply the matrices $\langle A_i, A_{i+1}, \ldots A_j \rangle$.

\[
  m[i, j] = \begin{cases} 
  0, & \text{if } j = i \\
  \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + d_{i-1} \cdot d_k \cdot d_j), & \text{if } j > i.
  \end{cases}
\]
Resource analysis

For space usage, observe that we need an array $m[i,j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

Note: We have left out some details in the algorithm; such as extracting the optimal solution. The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i,j]$.

Example: Find the optimal parenthesization for the chain $\langle A_7 \times 10 \cdot B_{10} \times 3 \cdot C_3 \times 8 \cdot D_8 \times 4 \rangle$. 

Algorithmic Insights
Computational Complexity
Resource analysis

Analysis

For space usage, observe that we need an array \( m[i, j] \) and some variable space. Thus, space usage is \( \Theta(n^2) \).

For time, note that each entry requires \( O(n) \) time. Since there are \( \Theta(n^2) \) entries to be filled out, the time taken by our dynamic programming algorithm is \( O(n^3) \).

Can you show that the time required is \( \Theta(n^3) \)?

Note: We have left out some details in the algorithm; such as extracting the optimal solution. The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the \( k \) that is optimal for \( m[i, j] \).

Example: Find the optimal parenthesization for the chain \( \langle A_{7 \times 10} \cdot B_{10 \times 3} \cdot C_{3 \times 8} \cdot D_{8 \times 4} \rangle \).
For space usage, observe that we need an array $m[i,j]$ and some variable space. Thus, space usage is $\Theta(n^2)$. For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$. Can you show that the time required is $\Theta(n^3)$?

Note: We have left out some details in the algorithm; such as extracting the optimal solution. The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i,j]$. Example: Find the optimal parenthesization for the chain $\langle A_{7 \times 10} \cdot B_{10 \times 3} \cdot C_{3 \times 8} \cdot D_{8 \times 4} \rangle$. Algorithmic Insights: Computational Complexity.
Resource analysis

Analysis

1. For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.
Resource analysis

Analysis

1. For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

2. For time, note that each entry requires $O(n)$ time.

Note: We have left out some details in the algorithm; such as extracting the optimal solution. The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i, j]$. 

Example: Find the optimal parenthesization for the chain $\langle A_{7 \times 10} \cdot B_{10 \times 3} \cdot C_{3 \times 8} \cdot D_{8 \times 4} \rangle$. 

Computational Complexity
For space usage, observe that we need an array $m[i,j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.
Resource analysis

Analysis

1. For space usage, observe that we need an array \( m[i, j] \) and some variable space. Thus, space usage is \( \Theta(n^2) \).

2. For time, note that each entry requires \( O(n) \) time. Since there are \( \Theta(n^2) \) entries to be filled out, the time taken by our dynamic programming algorithm is \( O(n^3) \).

Can you show that the time required is \( \Theta(n^3) \)?
Resource analysis

Analysis

1. For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

2. For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

Note

*We have left out some details in the algorithm;*
For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

We have left out some details in the algorithm; such as extracting the optimal solution.
Analysis

1. For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

2. For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

Note

We have left out some details in the algorithm; such as extracting the optimal solution.

The technique for extracting the optimal solution is similar to the rod-cutting problem;
1. For space usage, observe that we need an array $m[i,j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

2. For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by our dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

Note

We have left out some details in the algorithm; such as extracting the optimal solution.

The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i,j]$.
Analysis

1. For space usage, observe that we need an array $m[i,j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

2. For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by out dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

Note

We have left out some details in the algorithm; such as extracting the optimal solution.

The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i,j]$. 

Example
Resource analysis

Analysis

1. For space usage, observe that we need an array $m[i, j]$ and some variable space. Thus, space usage is $\Theta(n^2)$.

2. For time, note that each entry requires $O(n)$ time. Since there are $\Theta(n^2)$ entries to be filled out, the time taken by out dynamic programming algorithm is $O(n^3)$.

Can you show that the time required is $\Theta(n^3)$?

Note

We have left out some details in the algorithm; such as extracting the optimal solution.

The technique for extracting the optimal solution is similar to the rod-cutting problem; keep track of the $k$ that is optimal for $m[i, j]$.

Example

Find the optimal parenthesization for the chain $\langle A_{7 \times 10} \cdot B_{10 \times 3} \cdot C_{3 \times 8} \cdot D_{8 \times 4} \rangle$. 
The Longest Common Subsequence problem

You are given two subsequences of characters $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ over an alphabet $\Sigma$.

A subsequence of a sequence is defined as a sequence whose characters occur in the same order as the original sequence (not necessarily contiguous).

Compute the longest common subsequence (LCS) of $X$ and $Y$.

We use $X_i$ to denote the string $\langle x_1, x_2, ..., x_i \rangle$.

Brute-Force Approach

Assuming $m < n$, $X$ has $2^m$ possible subsequences.
The problem

You are given two subsequences of characters $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ over an alphabet $\Sigma$.

A subsequence of a sequence is defined as a sequence whose characters occur in the same order as the original sequence (not necessarily contiguous).

Compute the longest common subsequence (LCS) of $X$ and $Y$.

We use $X_i$ to denote the string $\langle x_1, x_2, ..., x_i \rangle$.

Brute-Force Approach

Assuming $m < n$, $X$ has $2^m$ possible subsequences.
The problem

You are given two subsequences of characters $X = \langle x_1, x_2, \ldots x_m \rangle$ and $Y = \langle y_1, y_2, \ldots y_n \rangle$ over an alphabet $\Sigma$. 
The problem

1. You are given two subsequences of characters $X = \langle x_1, x_2, \ldots x_m \rangle$ and $Y = \langle y_1, y_2, \ldots y_n \rangle$ over an alphabet $\Sigma$.

2. A subsequence of a sequence is defined as a sequence whose characters occur in the same order as the original sequence.
You are given two subsequences of characters $X = \langle x_1, x_2, \ldots x_m \rangle$ and $Y = \langle y_1, y_2, \ldots y_n \rangle$ over an alphabet $\Sigma$.

A subsequence of a sequence is defined as a sequence whose characters occur in the same order as the original sequence (not necessarily contiguous).
The problem

1. You are given two subsequences of characters $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ over an alphabet $\Sigma$.

2. A subsequence of a sequence is defined as a sequence whose characters occur in the same order as the original sequence (not necessarily contiguous).

Compute the longest common subsequence (LCS) of $X$ and $Y$. 
The problem

1. You are given two subsequences of characters $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ over an alphabet $\Sigma$.

2. A subsequence of a sequence is defined as a sequence whose characters occur in the same order as the original sequence (not necessarily contiguous).

Compute the longest common subsequence (LCS) of $X$ and $Y$.

We use $X_i$ to denote the string $\langle x_1, x_2, \ldots, x_i \rangle$. 
The problem

1. You are given two subsequences of characters $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ over an alphabet $\Sigma$.

2. A subsequence of a sequence is defined as a sequence whose characters occur in the same order as the original sequence (not necessarily contiguous).

Compute the longest common subsequence (LCS) of $X$ and $Y$.

We use $X_i$ to denote the string $\langle x_1, x_2, \ldots, x_i \rangle$.
The problem

1. You are given two subsequences of characters $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ over an alphabet $\Sigma$.

2. A subsequence of a sequence is defined as a sequence whose characters occur in the same order as the original sequence (not necessarily contiguous).

Compute the longest common subsequence (LCS) of $X$ and $Y$.

We use $X_i$ to denote the string $\langle x_1, x_2, \ldots, x_i \rangle$.

Brute-Force Approach

Assuming $m < n$, $X$ has $2^m$ possible subsequences.
Optimal Substructure

Theorem

Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, ..., z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.

2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y_n = Y$.

3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X_m = X$ and $Y_{n-1}$.

Recursive solution

Let $c[i, j]$ denote the length of the LCS between $X_i$ and $Y_j$. Then,

$$c[i, j] = \begin{cases} 
0, & \text{if } i = 0 \text{ or } j = 0 \\
1 + \max(c[i-1, j-1], c[i, j-1]), & \text{if } x_i = y_j \\
\max(c[i-1, j], c[i, j-1]), & \text{otherwise}
\end{cases}$$
Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, ..., z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.

2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_m$ and $Y_n$.

3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X_m = X$ and $Y_n$.

Recursive solution

Let $c[i, j]$ denote the length of the LCS between $X_i$ and $Y_j$. Then,

$$c[i, j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1, & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]), & \text{otherwise} \end{cases}$$
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.
Theorem

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. If $x_m = y_n$,
Optimal Substructure

Theorem

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
Theorem

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y_{n-1}$. 

Algorithmic Insights

Computational Complexity
Theorem

Let \( X = \langle x_1, x_2, \ldots x_m \rangle \) and \( Y = \langle y_1, y_2, \ldots y_n \rangle \) be two sequences and let \( Z = \langle z_1, z_2, \ldots, z_k \rangle \) denote their LCS.

1. If \( x_m = y_n \), then \( z_k = x_m \) and \( Z_{k-1} \) is an LCS of \( X_{m-1} \) and \( Y_{n-1} \).
2. If \( x_m \neq y_n \), then \( z_k \neq x_m \) implies that
Optimal Substructure

Theorem

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y = Y_n$. 

Algorithmic Insights
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y = Y_n$.
3. If $x_m \neq y_n$,
Theorem

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y = Y_n$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y = Y_{n-1}$. 

Algorithmic Insights

Computational Complexity
Review of concepts

The Greedy Approach

Dynamic Programming

Optimal Substructure

**Theorem**

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y = Y_n$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X_m = X$ and $Y_{n-1}$.

**Algorithmic Insights**

**Computational Complexity**
Theorem

Let \( X = \langle x_1, x_2, \ldots, x_m \rangle \) and \( Y = \langle y_1, y_2, \ldots, y_n \rangle \) be two sequences and let \( Z = \langle z_1, z_2, \ldots, z_k \rangle \) denote their LCS.

1. If \( x_m = y_n \), then \( z_k = x_m \) and \( Z_{k-1} \) is an LCS of \( X_{m-1} \) and \( Y_{n-1} \).
2. If \( x_m \neq y_n \), then \( z_k \neq x_m \) implies that \( Z \) is an LCS of \( X_{m-1} \) and \( Y = Y_n \).
3. If \( x_m \neq y_n \), then \( z_k \neq y_n \) implies that \( Z \) is an LCS of \( X_m = X \) and \( Y_{n-1} \).

Recursive solution
Theorem

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y = Y_n$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X_m = X$ and $Y_{n-1}$.

Recursive solution

Let $c[i, j]$ denote the length of the LCS between $X_i$ and $Y_j$. 
Optimal Substructure

**Theorem**

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. **If** $x_m = y_n$, **then** $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. **If** $x_m \neq y_n$, **then** $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y = Y_n$.
3. **If** $x_m \neq y_n$, **then** $z_k \neq y_n$ implies that $Z$ is an LCS of $X_m = X$ and $Y_{n-1}$.

**Recursive solution**

Let $c[i, j]$ denote the length of the LCS between $X_i$ and $Y_j$. Then,

\[
c[i, j] = \begin{cases} 
0, & \text{if } i = 0 \text{ or } j = 0 \\
1 + c[i-1, j-1], & \text{if } x_i = y_j \\
\max(c[i-1, j], c[i, j-1]), & \text{otherwise} 
\end{cases}
\]
Theorem

Let $X = \langle x_1, x_2, \ldots , x_m \rangle$ and $Y = \langle y_1, y_2, \ldots , y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots , z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y = Y_n$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X_m = X$ and $Y_{n-1}$.

Recursive solution

Let $c[i, j]$ denote the length of the LCS between $X_i$ and $Y_j$. Then,

$$c[i, j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0; \\ c[i-1, j-1] + 1, & \text{if } x_i = y_j; \\ \max(c[i-1, j], c[i, j-1]), & \text{otherwise}. \end{cases}$$
Theorem

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y = Y_n$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X_m = X$ and $Y_{n-1}$.

Recursive solution

Let $c[i, j]$ denote the length of the LCS between $X_i$ and $Y_j$. Then,

$$c[i, j] = \begin{cases} 
0, & \text{if } i = 0 \text{ or } j = 0 \\
\max \left( c[i, j-1], c[i-1, j] \right) + 1, & \text{otherwise}
\end{cases}$$
Theorem

Let \( X = \langle x_1, x_2, \ldots, x_m \rangle \) and \( Y = \langle y_1, y_2, \ldots, y_n \rangle \) be two sequences and let \( Z = \langle z_1, z_2, \ldots, z_k \rangle \) denote their LCS.

1. If \( x_m = y_n \), then \( z_k = x_m \) and \( Z_{k-1} \) is an LCS of \( X_{m-1} \) and \( Y_{n-1} \).
2. If \( x_m \neq y_n \), then \( z_k \neq x_m \) implies that \( Z \) is an LCS of \( X_{m-1} \) and \( Y = Y_n \).
3. If \( x_m \neq y_n \), then \( z_k \neq y_n \) implies that \( Z \) is an LCS of \( X_m = X \) and \( Y_{n-1} \).

Recursive solution

Let \( c[i, j] \) denote the length of the LCS between \( X_i \) and \( Y_j \). Then,

\[
c[i, j] = \begin{cases} 
0, & \text{if } i = 0 \text{ or } j = 0 \\
c[i - 1, j - 1] + 1, & \text{otherwise}
\end{cases}
\]
Theorem

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y = Y_n$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X_m = X$ and $Y_{n-1}$.

Recursive solution

Let $c[i, j]$ denote the length of the LCS between $X_i$ and $Y_j$. Then,

$$c[i, j] = \begin{cases} 
0, & \text{if } i = 0 \text{ or } j = 0 \\
 c[i-1, j-1] + 1, & \text{if } x_i = y_j 
\end{cases}$$
Optimal Substructure

Theorem

Let \( X = \langle x_1, x_2, \ldots, x_m \rangle \) and \( Y = \langle y_1, y_2, \ldots, y_n \rangle \) be two sequences and let \( Z = \langle z_1, z_2, \ldots, z_k \rangle \) denote their LCS.

1. If \( x_m = y_n \), then \( z_k = x_m \) and \( Z_{k-1} \) is an LCS of \( X_{m-1} \) and \( Y_{n-1} \).
2. If \( x_m \neq y_n \), then \( z_k \neq x_m \) implies that \( Z \) is an LCS of \( X_{m-1} \) and \( Y = Y_n \).
3. If \( x_m \neq y_n \), then \( z_k \neq y_n \) implies that \( Z \) is an LCS of \( X_m = X \) and \( Y_{n-1} \).

Recursive solution

Let \( c[i, j] \) denote the length of the LCS between \( X_i \) and \( Y_j \). Then,

\[
c[i, j] = \begin{cases} 
0, & \text{if } i = 0 \text{ or } j = 0 \\
\max(c[i-1, j-1] + 1, \max(c[i, j-1], c[i-1, j])), & \text{if } x_i \neq y_j \\
\max(c[i, j-1], c[i-1, j]), & \text{if } x_i = y_j
\end{cases}
\]
Theorem

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ denote their LCS.

1. If $x_m = y_n$, then $z_k = x_m$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y = Y_n$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X_m = X$ and $Y_{n-1}$.

Recursive solution

Let $c[i, j]$ denote the length of the LCS between $X_i$ and $Y_j$. Then,

$$c[i, j] = \begin{cases} 
0, & \text{if } i = 0 \text{ or } j = 0 \\
\max(c[i - 1, j - 1] + 1, \max(c[i, j - 1], c[i - 1, j])), & \text{otherwise}
\end{cases}$$
Resource Analysis

We require to store the matrix $c_{i,j}$ and some auxiliary space. Thus, the space needed is $O(m \cdot n)$.

Each entry in the table can be computed in $O(1)$ time and hence the total time taken is $O(m \cdot n)$.

Example

Find the LCS of $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. 
We require to store the matrix $c[i,j]$ and some auxiliary space. Thus, the space needed is $O(m \cdot n)$. Each entry in the table can be computed in $O(1)$ time and hence the total time taken is $O(m \cdot n)$.

Example
Find the LCS of $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$.
Resource Analysis

We require to store the matrix $c[i, j]$ and some auxiliary space.

Example

Find the LCS of $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. 
Analysis

Resource Analysis

We require to store the matrix $c[i, j]$ and some auxiliary space. Thus, the space needed is $O(m \cdot n)$. 

Resource Analysis

We require to store the matrix $c[i, j]$ and some auxiliary space. Thus, the space needed is $O(m \cdot n)$. Each entry in the table can be computed in $O(1)$ time and hence the total time taken is $O(m \cdot n)$. 

Example

Find the LCS of $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. 
Resource Analysis

We require to store the matrix $c[i, j]$ and some auxiliary space. Thus, the space needed is $O(m \cdot n)$. Each entry in the table can be computed in $O(1)$ time and hence the total time taken is $O(m \cdot n)$.

Example
Analysis

Resource Analysis

We require to store the matrix $c[i,j]$ and some auxiliary space. Thus, the space needed is $O(m \cdot n)$. Each entry in the table can be computed in $O(1)$ time and hence the total time taken is $O(m \cdot n)$.

Example

Find the LCS of $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. 
The Pretty Typesetting problem

You are given $n$ words $w_1, w_2, \ldots, w_n$, which need to be packed into a paragraph.

On each line, you can pack at most $M$ characters.

There needs to be exactly one space (one character) between consecutive words on a line.

The cost of a packing for a given line is the cube of the number of spaces left over.

The cost of packing the entire set of words is the sum of the costs of packing over each line.

The cost of a packing is infinity, if the number of words plus the accompanying spaces exceeds $M$.

There is no cost for packing on the last line.

Find the minimum cost of packing the words into lines.
The Pretty Typesetting problem

The Problem

You are given $n$ words $w_1, w_2, ..., w_n$, which need to be packed into a paragraph.

On each line, you can pack at most $M$ characters.

There needs to be exactly one space (one character) between consecutive words on a line.

The cost of a packing for a given line is the cube of the number of spaces left over.

The cost of packing the entire set of words is the sum of the costs of packing over each line.

The cost of a packing is infinity, if the number of words plus the accompanying spaces exceeds $M$.

There is no cost for packing on the last line.

Find the minimum cost of packing the words into lines.
The Pretty Typesetting problem

The Problem

You are given $n$ words $w_1, w_2, \ldots w_n$, which need to be packed into a paragraph.
The Pretty Typesetting problem

The Problem

1. You are given $n$ words $w_1, w_2, \ldots w_n$, which need to be packed into a paragraph.
2. Word $w_i$ has length $l_i$, $i = 1, 2, \ldots n$. 
The Pretty Typesetting problem

The Problem

1. You are given $n$ words $w_1, w_2, \ldots, w_n$, which need to be packed into a paragraph.
2. Word $w_i$ has length $l_i$, $i = 1, 2, \ldots, n$.
3. On each line, you can pack at most $M$ characters.
The Pretty Typesetting problem

The Problem

1. You are given $n$ words $w_1, w_2, \ldots w_n$, which need to be packed into a paragraph.
2. Word $w_i$ has length $l_i$, $i = 1, 2, \ldots n$.
3. On each line, you can pack at most $M$ characters.
4. There needs to be exactly one space
The Pretty Typesetting problem

The Problem

1. You are given $n$ words $w_1, w_2, \ldots w_n$, which need to be packed into a paragraph.
2. Word $w_i$ has length $l_i$, $i = 1, 2, \ldots n$.
3. On each line, you can pack at most $M$ characters.
4. There needs to be exactly one space (one character) between consecutive words on a line.
The Pretty Typesetting problem

The Problem

1. You are given $n$ words $w_1, w_2, \ldots w_n$, which need to be packed into a paragraph.
2. Word $w_i$ has length $l_i$, $i = 1, 2, \ldots n$.
3. On each line, you can pack at most $M$ characters.
4. There needs to be exactly one space (one character) between consecutive words on a line.
5. The cost of a packing for a given line is the cube of the number of spaces left over.
The Pretty Typesetting problem

1. You are given \( n \) words \( w_1, w_2, \ldots w_n \), which need to be packed into a paragraph.
2. Word \( w_i \) has length \( l_i, i = 1, 2, \ldots n \).
3. On each line, you can pack at most \( M \) characters.
4. There needs to be exactly one space (one character) between consecutive words on a line.
5. The cost of a packing for a given line is the cube of the number of spaces left over.
6. The cost of packing the entire set of words is the sum of the costs of packing over each line.
The Pretty Typesetting problem

The Problem

1. You are given \( n \) words \( w_1, w_2, \ldots w_n \), which need to be packed into a paragraph.
2. Word \( w_i \) has length \( l_i, i = 1, 2, \ldots n. \)
3. On each line, you can pack at most \( M \) characters.
4. There needs to be exactly one space (one character) between consecutive words on a line.
5. The cost of a packing for a given line is the cube of the number of spaces left over.
6. The cost of packing the entire set of words is the sum of the costs of packing over each line.
7. The cost of a packing is infinity, if the number of words plus the accompanying spaces exceeds \( M \).
The Problem

1. You are given \( n \) words \( w_1, w_2, \ldots w_n \), which need to be packed into a paragraph.
2. Word \( w_i \) has length \( l_i, i = 1, 2, \ldots n \).
3. On each line, you can pack at most \( M \) characters.
4. There needs to be exactly one space (one character) between consecutive words on a line.
5. The cost of a packing for a given line is the cube of the number of spaces left over.
6. The cost of packing the entire set of words is the sum of the costs of packing over each line.
7. The cost of a packing is infinity, if the number of words plus the accompanying spaces exceeds \( M \).
8. There is no cost for packing on the last line.
The Pretty Typesetting problem

The Problem

1. You are given \( n \) words \( w_1, w_2, \ldots w_n \), which need to be packed into a paragraph.
2. Word \( w_i \) has length \( l_i, i = 1, 2, \ldots n \).
3. On each line, you can pack at most \( M \) characters.
4. There needs to be exactly one space (one character) between consecutive words on a line.
5. The cost of a packing for a given line is the cube of the number of spaces left over.
6. The cost of packing the entire set of words is the sum of the costs of packing over each line.
7. The cost of a packing is infinity, if the number of words plus the accompanying spaces exceeds \( M \).
8. There is no cost for packing on the last line.

Find the minimum cost of packing the words into lines.
Formulating the cost function

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n-p)$ words on the remaining $(k-1)$ lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let $t_{ij}$ denote the space left over on a line, when packing the words $w_i, \ldots, w_j$ are packed into that line.

It is not hard to see that $t_{ij} = M - (j - i) - \sum_{j=k}^{i} l_k$.

Let $s_{[i,j]}$ denote the packing cost of packing words $w_i$ through $w_j$ in one line.

The following equations are immediate:

$$s_{[i,j]} =\begin{cases} t_{ij}, & \text{if } t_{ij} \geq 0 \\ 0, & \text{if } t_{ij} \geq 0 \text{ and } j = n \\ \infty, & \text{if } t_{ij} < 0 \end{cases}$$ (3)
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on \( k \) lines consists of \( p \) (say) words on the first line and the remaining \( (n - p) \) words on the remaining \( (k - 1) \) lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let \( t_{ij} \) denote the space left over on a line, when packing the words \( w_i \ldots w_j \) are packed into that line.

It is not hard to see that \( t_{ij} = M - (j - i) - \sum_{j=k}^{i} l_k \).

Let \( s_{[i,j]} \) denote the packing cost of packing words \( w_i \ldots w_j \) in one line.

The following equations are immediate:

\[
\begin{align*}
    s_{[i,j]} &= \begin{cases} 
        t_{ij}, & \text{if } t_{ij} \geq 0 \\
        0, & \text{if } t_{ij} \geq 0 \text{ and } j = n \\
        \infty, & \text{if } t_{ij} < 0
    \end{cases}
\end{align*}
\]
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n-p)$ words on the remaining $(k-1)$ lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let $t_{ij}$ denote the space left over on a line, when packing the words $w_i \ldots w_j$ are packed into that line.

It is not hard to see that $t_{ij} = M - (j-i) - \sum_{j=1}^{k} a_l$.

Let $s_{[i,j]}$ denote the packing cost of packing words $w_i \ldots w_j$ in one line.

The following equations are immediate:

$$s_{[i,j]} = \begin{cases} t_{ij}, & \text{if } t_{ij} \geq 0 \\ 0, & \text{if } t_{ij} \geq 0 \text{ and } j = n \\ \infty, & \text{if } t_{ij} < 0 \end{cases}$$
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n - p)$ words on the remaining $(k - 1)$ lines.

$$t_{ij} = M - (j - i) - \sum_{j} k = l_{ik}.$$

$$s_{[i, j]} = \begin{cases} 
  t_{ij}, & \text{if } t_{ij} \geq 0 \\
  0, & \text{if } t_{ij} \geq 0 \text{ and } j = n \\
  \infty, & \text{otherwise} 
\end{cases}.$$
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n - p)$ words on the remaining $(k - 1)$ lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

Let $t_{ij}$ denote the space left over on a line, when packing the words $w_i$...$w_j$ are packed into that line. It is not hard to see that $t_{ij} = M - (j - i) - \sum_{j=k}^{i} l_k$. Let $s_{[i,j]}$ denote the packing cost of packing words $w_i$ through $w_j$ in one line. The following equations are immediate:

$$s_{[i,j]} = \begin{cases} t_{ij}, & \text{if } t_{ij} \geq 0 \\ 0, & \text{if } t_{ij} \geq 0 \text{ and } j = n \\ \infty, & \text{if } t_{ij} < 0 \end{cases}$$
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on \( k \) lines consists of \( p \) (say) words on the first line and the remaining \( (n – p) \) words on the remaining \( (k – 1) \) lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.
Formulating the cost function

**Cost structure**

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n-p)$ words on the remaining $(k-1)$ lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let $t_{ij}$ denote the space left over on a line, when packing the words $w_i \ldots w_j$ are packed into that line.

\[
\begin{align*}
t_{ij} & = \begin{cases} 
\text{if } t_{ij} \geq 0 \\
0 & \text{if } t_{ij} \geq 0 \text{ and } j = n \\
-\infty & \text{otherwise}
\end{cases} 
\end{align*}
\]
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n-p)$ words on the remaining $(k-1)$ lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let $t_{ij}$ denote the space left over on a line, when packing the words $w_i \ldots w_j$ are packed into that line.

It is not hard to see that $t_{ij} =$
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n - p)$ words on the remaining $(k - 1)$ lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let $t_{ij}$ denote the space left over on a line, when packing the words $w_i \ldots w_j$ are packed into that line.

It is not hard to see that $t_{ij} = M - (j - i) - \sum_{k=i}^{j} l_k$. 

Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on \(k\) lines consists of \(p\) (say) words on the first line and the remaining \((n - p)\) words on the remaining \((k - 1)\) lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let \(t_{ij}\) denote the space left over on a line, when packing the words \(w_i \ldots w_j\) are packed into that line.

It is not hard to see that \(t_{ij} = M - (j - i) - \sum_{k=i}^{j} l_k\).

Let \(s[i,j]\) denote the packing cost of packing words \(w_i\) through \(w_j\)
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n - p)$ words on the remaining $(k - 1)$ lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let $t_{ij}$ denote the space left over on a line, when packing the words $w_i \ldots w_j$ are packed into that line.

It is not hard to see that $t_{ij} = M - (j - i) - \sum_{k=i}^{j} l_k$.

Let $s[i, j]$ denote the packing cost of packing words $w_i$ through $w_j$ in one line.
## Formulating the cost function

### Cost structure

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n - p)$ words on the remaining $(k - 1)$ lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let $t_{ij}$ denote the space left over on a line, when packing the words $w_i \ldots w_j$ are packed into that line.

It is not hard to see that $t_{ij} = M - (j - i) - \sum_{k=i}^{j} l_k$.

Let $s[i,j]$ denote the packing cost of packing words $w_i$ through $w_j$ in one line. The following equations are immediate:

<table>
<thead>
<tr>
<th>$s[i,j]$</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{ij}$</td>
<td>$M - (j - i) - \sum_{k=i}^{j} l_k$</td>
</tr>
</tbody>
</table>

*Note: $M$ represents the maximum available space on the page.*
There are 2 key observations to make:

1. Any optimal solution on \( k \) lines consists of \( p \) (say) words on the first line and the remaining \((n - p)\) words on the remaining \((k - 1)\) lines.
2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let \( t_{ij} \) denote the space left over on a line, when packing the words \( w_i \ldots w_j \) are packed into that line.

It is not hard to see that

\[
  t_{ij} = M - (j - i) - \sum_{k=i}^{j} l_k.
\]

Let \( s[i, j] \) denote the packing cost of packing words \( w_i \) through \( w_j \) \textit{in one line}.

The following equations are immediate:

\[
  s[i, j] = \left\{
  \begin{array}{l}
    t_{ij}, \\ 
    0, \\ 
    \infty,
  \end{array}
  \right. \quad \text{if} \quad t_{ij} \geq 0 \\
  \text{if} \quad t_{ij} < 0
\]
Cost structure

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n-p)$ words on the remaining $(k-1)$ lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let $t_{ij}$ denote the space left over on a line, when packing the words $w_i \ldots w_j$ are packed into that line.

It is not hard to see that $t_{ij} = M - (j - i) - \sum_{k=i}^{j} l_k$.

Let $s[i,j]$ denote the packing cost of packing words $w_i$ through $w_j$ in one line. The following equations are immediate:

$$s[i,j] = \begin{cases} t_{ij}^3, & \text{if } t_{ij} \geq 0 \\ 0, & \text{if } t_{ij} \geq 0 \text{ and } j = n \\ \infty, & \text{if } t_{ij} < 0 \end{cases}$$
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on \( k \) lines consists of \( p \) (say) words on the first line and the remaining \((n - p)\) words on the remaining \((k - 1)\) lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let \( t_{ij} \) denote the space left over on a line, when packing the words \( w_i \ldots w_j \) are packed into that line.

It is not hard to see that \( t_{ij} = M - (j - i) - \sum_{k=i}^{j} l_k \).

Let \( s[i,j] \) denote the packing cost of packing words \( w_i \) through \( w_j \) in one line. The following equations are immediate:

\[
s[i,j] = \begin{cases} 
t_{ij}^3, & \text{if } t_{ij} \geq 0 
\end{cases}
\]
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on \( k \) lines consists of \( p \) (say) words on the first line and the remaining \( (n - p) \) words on the remaining \( (k - 1) \) lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let \( t_{ij} \) denote the space left over on a line, when packing the words \( w_i \ldots w_j \) are packed into that line.

It is not hard to see that \( t_{ij} = M - (j - i) - \sum_{k=i}^{j} l_k \).

Let \( s[i, j] \) denote the packing cost of packing words \( w_i \) through \( w_j \) in one line.

The following equations are immediate:

\[
\begin{aligned}
s[i, j] &= \begin{cases} 
  t_{ij}^3, & \text{if } t_{ij} \geq 0 \\
  0, & \text{otherwise}
\end{cases}
\end{aligned}
\]
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n-p)$ words on the remaining $(k-1)$ lines.
2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let $t_{ij}$ denote the space left over on a line, when packing the words $w_i \ldots w_j$ are packed into that line.

It is not hard to see that $t_{ij} = M - (j - i) - \sum_{k=i}^{j} l_k$.

Let $s[i,j]$ denote the packing cost of packing words $w_i$ through $w_j$ in one line. The following equations are immediate:

$$s[i,j] = \begin{cases} t_{ij}^3, & \text{if } t_{ij} \geq 0 \\ 0, & \text{if } t_{ij} \geq 0 \text{ and } j = n \end{cases}$$
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on \( k \) lines consists of \( p \) (say) words on the first line and the remaining \( (n - p) \) words on the remaining \( (k - 1) \) lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let \( t_{ij} \) denote the space left over on a line, when packing the words \( w_i \ldots w_j \) are packed into that line.

It is not hard to see that \( t_{ij} = M - (j - i) - \sum_{k=i}^{j} l_k \).

Let \( s[i, j] \) denote the packing cost of packing words \( w_i \) through \( w_j \) in one line.

The following equations are immediate:

\[
s[i, j] = \begin{cases} 
  t_{ij}^3, & \text{if } t_{ij} \geq 0 \\
  0, & \text{if } t_{ij} \geq 0 \text{ and } j = n \\
  \infty, & \text{otherwise}
\end{cases}
\]
Formulating the cost function

Cost structure

There are 2 key observations to make:

1. Any optimal solution on $k$ lines consists of $p$ (say) words on the first line and the remaining $(n - p)$ words on the remaining $(k - 1)$ lines.

2. If all the words fit on one line, it is sub-optimal to break up the words into two or more lines.

We first formulate the cost function.

Let $t_{ij}$ denote the space left over on a line, when packing the words $w_i \ldots w_j$ are packed into that line.

It is not hard to see that $t_{ij} = M - (j - i) - \sum_{k=i}^{j} l_k$.

Let $s[i, j]$ denote the packing cost of packing words $w_i$ through $w_j$ in one line. The following equations are immediate:

$$s[i, j] = \begin{cases} 
    t_{ij}^3, & \text{if } t_{ij} \geq 0 \\
    0, & \text{if } t_{ij} \geq 0 \text{ and } j = n \\
    \infty, & \text{if } t_{ij} < 0 
\end{cases}$$  \hspace{1cm} (3)
Review of concepts
The Greedy Approach
Dynamic Programming

Optimal substructure

Let $m[i, j]$ be the optimal cost of packing words $w_i$ through $w_j$ with word $w_i$ starting on a fresh line. Hence, we are interested in $m[1, n]$.

The following recurrence is immediate:

$$m[i, j] = \begin{cases} 
  s[i, j], & \text{if } t_{ij} \geq 0 \\
  \min_{i \leq k \leq j} (s[i, k] + m[k+1, j]), & \text{otherwise (4)}
\end{cases}$$

Analyze the resources of the above algorithm.

Algorithmic Insights
Computational Complexity
Optimal substructure

Let $m[i,j]$ be the optimal cost of packing words $w_i$ through $w_j$ with word $w_i$ starting on a fresh line. Hence, we are interested in $m[1,n]$.

The following recurrence is immediate:

$$m[i,j] = \begin{cases} s[i,j], & \text{if } t_{ij} \geq 0 \\ \min_{i \leq k \leq j} (s[i,k] + m[k+1,j]) & \text{otherwise} \end{cases}$$

Analyze the resources of the above algorithm.

Algorithmic Insights
Computational Complexity
Optimal substructure

Recursive solution

Let $m[i, j]$ be the optimal cost of packing words $w_i$ through $w_j$ with word $w_i$ starting on a fresh line.
Optimal substructure

Recursive solution

Let $m[i, j]$ be the optimal cost of packing words $w_i$ through $w_j$ with word $w_i$ starting on a fresh line.

Hence, we are interested in
Recursive solution

Let $m[i, j]$ be the optimal cost of packing words $w_i$ through $w_j$ with word $w_i$ starting on a fresh line.

Hence, we are interested in $m[1, n]$. 

The following recurrence is immediate:

$$m[i, j] = \begin{cases} s[i, j], & \text{if } t_{ij} \geq 0 \\ \min_{i \leq k \leq j} (s[i, k] + m[k+1, j]) & \text{otherwise} \end{cases}$$

Analyze the resources of the above algorithm.
Optimal substructure

**Recursive solution**

Let $m[i, j]$ be the optimal cost of packing words $w_i$ through $w_j$ with word $w_i$ starting on a fresh line.

Hence, we are interested in $m[1, n]$.

The following recurrence is immediate:

$$m[i, j] = \begin{cases} 
    s[i, j], & \text{if } t_{ij} \geq 0 \\
    \min_{i \leq k \leq j} (s[i, k] + m[k+1, j]), & \text{otherwise} 
\end{cases}$$
Recursive solution

Let $m[i, j]$ be the optimal cost of packing words $w_i$ through $w_j$ with word $w_i$ starting on a fresh line.

Hence, we are interested in $m[1, n]$.

The following recurrence is immediate:

$$m[i, j] = \begin{cases} s[i, j], & \text{if } t_{ij} \geq 0 \\ \min_{i \leq k \leq j} (s[i, k] + m[k+1, j]), & \text{otherwise} \end{cases}$$
Review of concepts
The Greedy Approach
Dynamic Programming

Optimal substructure

Recursive solution

Let $m[i, j]$ be the optimal cost of packing words $w_i$ through $w_j$ with word $w_i$ starting on a fresh line.

Hence, we are interested in $m[1, n]$.

The following recurrence is immediate:

$$m[i, j] = \begin{cases} s[i, j], & \text{if } t_{ij} \geq 0 \\ \min_{i \leq k \leq j} (s[i, k] + m[k+1, j]) & \text{otherwise} \end{cases}$$

(4)
Review of concepts
The Greedy Approach
Dynamic Programming

Optimal substructure

Recursive solution

Let $m[i,j]$ be the optimal cost of packing words $w_i$ through $w_j$ with word $w_i$ starting on a fresh line.

Hence, we are interested in $m[1,n]$.

The following recurrence is immediate:

$$m[i,j] = \begin{cases} s[i,j], & \text{if } t_{ij} \geq 0 \\ \min_{i \leq k \leq j} (s[i,k] + m[k+1,j]) & \text{otherwise} \end{cases}$$

Analyse the resources of the above algorithm.

Algorithmic Insights
Computational Complexity
Recursive solution

Let $m[i, j]$ be the optimal cost of packing words $w_i$ through $w_j$ with word $w_i$ starting on a fresh line.

Hence, we are interested in $m[1, n]$.

The following recurrence is immediate:

$$m[i, j] = \begin{cases} 
    s[i, j], & \text{if } t_{ij} \geq 0 \\
    \min_{i \leq k \leq j} (s[i, k] + m[k + 1, j]), & \text{otherwise}
\end{cases}$$

(4)
Optimal substructure

Recursive solution

Let $m[i, j]$ be the optimal cost of packing words $w_i$ through $w_j$ with word $w_i$ starting on a fresh line.

Hence, we are interested in $m[1, n]$.

The following recurrence is immediate:

$$m[i, j] = \begin{cases} s[i, j], & \text{if } t_{ij} \geq 0 \\ \min_{i \leq k \leq j} (s[i, k] + m[k + 1, j]), & \text{otherwise} \end{cases}$$

(4)

Analyze the resources of the above algorithm.
The Reachability problem

Given a directed, unweighted graph $G = \langle V, E \rangle$ and a pair of vertices $s, t \in V$, is there a dipath from $s$ to $t$?
The Reachability problem

The Problem

Given a directed, unweighted graph \( G = \langle V, E \rangle \) and a pair of vertices \( s, t \in V \), is there a dipath from \( s \) to \( t \)?
The Reachability problem

The Problem

Given a directed, unweighted graph $G = \langle V, E \rangle$ and a pair of vertices $s, t \in V$, is there a dipath from $s$ to $t$?
Graph Exploration

Algorithm 4.53: The generic algorithm

Function \( \text{EXPLORE}(G, s, t) \)

1: \( Q = \{s\} \).
2: while \( Q \neq \emptyset \) do
3:   Remove a vertex \( u \) from \( Q \).
4:   if \( u = t \), then \( t \) is reachable from \( u \).
5:   Mark \( u \).
6:   for (all unmarked neighbors \( v \) of \( u \)) do
7:     Insert \( v \) into \( Q \).
8:  end for
9: end while

Algorithmic Insights

Computational Complexity
Graph Exploration

The Exploration Algorithm

**Function** \( \text{EXPLORE}(G, s, t) \)

1. \( Q = \{ s \} \).
2. While \( Q \neq \emptyset \) do
   3. Remove a vertex \( u \) from \( Q \).
   4. Mark \( u \).
   5. For (all unmarked neighbors \( v \) of \( u \)) do
      6. Insert \( v \) into \( Q \).
   7. End for
3. End while

Algorithm 4.54: The generic algorithm
The Exploration Algorithm

Function $\text{EXPLORE}(G, s, t)$

1: $Q = \{s\}$. 
Graph Exploration

The Exploration Algorithm

Function `EXPLORE(G, s, t)`

1: \( Q = \{ s \} \).
2: while \( Q \neq \emptyset \) do

The Exploration Algorithm
The Exploration Algorithm

**Function** \( \text{EXPLORE}(G, s, t) \)

1. \( Q = \{s\} \).
2. while \( (Q \neq \emptyset) \) do
3. Remove a vertex \( u \) from \( Q \).
The Exploration Algorithm

Function `EXPLORE(G, s, t)`

1: \( Q = \{s\} \).
2: \textbf{while} \( Q \neq \emptyset \) \textbf{do}
3: \hspace{1em} Remove a vertex \( u \) from \( Q \). \{If \( u = t \), then \( t \) is reachable from \( u \).\}

Algorithm 4.58: The generic algorithm
Graph Exploration

The Exploration Algorithm

**Function** \( EXPLORE(G, s, t) \)

1: \( Q = \{s\} \).
2: \textbf{while} \ (Q \neq \emptyset) \ \textbf{do}
3: \text{Remove a vertex } u \text{ from } Q. \ \{\text{If } u = t, \text{ then } t \text{ is reachable from } u.\} \\
4: \text{Mark } u.
The Exploration Algorithm

**Function** \( \text{EXPLORE}(G, s, t) \)

1. \( Q = \{s\} \).
2. \( \textbf{while} \ (Q \neq \emptyset) \ \textbf{do} \)
3. Remove a vertex \( u \) from \( Q \). \{If \( u = t \), then \( t \) is reachable from \( u \).\}
4. Mark \( u \).
5. \( \textbf{for} \ (\text{all unmarked neighbors} \ v \ \text{of} \ u) \ \textbf{do} \)

The Exploration Algorithm

Function \text{EXPLORE}(G, s, t)

1. $Q = \{s\}$.
2. \textbf{while} ($Q \neq \emptyset$) \textbf{do}
3. \hspace{1em} Remove a vertex $u$ from $Q$. \{If $u = t$, then $t$ is reachable from $u$.\}
4. \hspace{1em} Mark $u$.
5. \hspace{1em} \textbf{for} (all unmarked neighbors $v$ of $u$) \textbf{do}
6. \hspace{2em} Insert $v$ into $Q$. 
The Exploration Algorithm

Function \textsc{Explore}(G, s, t)

1: \( Q = \{s\} \).
2: \textbf{while} \ (Q \neq \emptyset) \ \textbf{do}
3: \quad \text{Remove a vertex } u \text{ from } Q. \ {\text{If } u = t, \text{ then } t \text{ is reachable from } u.}\}
4: \quad \text{Mark } u.
5: \quad \textbf{for} \ (\text{all unmarked neighbors } v \text{ of } u) \ \textbf{do}
6: \quad \quad \text{Insert } v \text{ into } Q.
7: \quad \textbf{end for}
8: \textbf{end while}
The Exploration Algorithm

Function $\text{EXPLORE}(G, s, t)$

1: $Q = \{s\}$.
2: while $(Q \neq \emptyset)$ do
3:     Remove a vertex $u$ from $Q$. {If $u = t$, then $t$ is reachable from $u$.}
4:     Mark $u$.
5:     for (all unmarked neighbors $v$ of $u$) do
6:         Insert $v$ into $Q$.
7:     end for
8: end while

Algorithm 4.63: The generic algorithm
Two common search techniques

Breadth-first Search
Implement Q as a queue.

Depth-first Search
Implement Q as a stack.

Analysis
Both algorithms run in $O(m + n)$ time.

Algorithmic Insights
Computational Complexity
Two common search techniques

Breadth-first Search

Implement \( Q \) as a queue.

Depth-first Search

Implement \( Q \) as a stack.

Analysis

Both algorithms run in \( O(m + n) \) time.
Two common search techniques

Breadth-first Search

Implement $Q$ as a queue.
Two common search techniques

Breadth-first Search
Implement Q as a queue.

Depth-first Search
Two common search techniques

Breadth-first Search
Implement $Q$ as a queue.

Depth-first Search
Implement $Q$ as a stack.
Two common search techniques

Breadth-first Search
Implement $Q$ as a queue.

Depth-first Search
Implement $Q$ as a stack.

Analysis

Both algorithms run in $O(m + n)$ time.
Two common search techniques

**Breadth-first Search**
Implement $Q$ as a queue.

**Depth-first Search**
Implement $Q$ as a stack.

**Analysis**
Both algorithms run in $O(m + n)$ time.
Two common search techniques

Breadth-first Search
Implement $Q$ as a queue.

Depth-first Search
Implement $Q$ as a stack.

Analysis
Both algorithms run in $O(m + n)$ time.
Middle First Search

Definition

The adjacency matrix of a graph with $n$ vertices, is a $n \times n$ matrix $A$ where $A_{ij} = 1$, if there is an edge from vertex $i$ to $j$ and 0 otherwise.

Observation

Let $A_t$ denote the matrix product $A \cdot A \cdot \ldots \cdot A$ ($t$ times). Then, $(A_t)_{ij}$ is the number of paths of length $t$ from $i$ to $j$. 

Algorithmic Insights

Computational Complexity
Definition

The adjacency matrix of a graph with \( n \) vertices, is a \( n \times n \) matrix \( A \) where \( A_{ij} = 1 \), if there is an edge from vertex \( i \) to \( j \) and 0 otherwise.

Observation

Let \( A_t \) denote the matrix product \( A \cdot A \cdot \ldots \cdot A \) (\( t \) times).

Then, \( (A_t)_{ij} \) is the number of paths of length \( t \) from \( i \) to \( j \).
Definition

The adjacency matrix of a graph with \( n \) vertices, is a \( n \times n \) matrix \( A \) where \( A_{ij} = 1 \), if there is an edge from vertex \( i \) to \( j \) and 0 otherwise.
Definition

The adjacency matrix of a graph with \( n \) vertices, is a \( n \times n \) matrix \( A \) where \( A_{ij} = 1 \), if there is an edge from vertex \( i \) to \( j \) and 0 otherwise.

Observation

Let \( A_t \) denote the matrix product \( A \cdot A \cdot \ldots \cdot A \) (\( t \) times).

Then, \( (A_t)_{ij} \) is the number of paths of length \( t \) from \( i \) to \( j \).
Middle First Search

**Definition**
The adjacency matrix of a graph with $n$ vertices, is a $n \times n$ matrix $A$ where $A_{ij} = 1$, if there is an edge from vertex $i$ to $j$ and 0 otherwise.

**Observation**
Let $A^t$ denote the matrix product $A \cdot A \cdot \cdots A$ ($t$ times).
**Definition**

The adjacency matrix of a graph with \( n \) vertices, is a \( n \times n \) matrix \( A \) where \( A_{ij} = 1 \), if there is an edge from vertex \( i \) to \( j \) and 0 otherwise.

**Observation**

Let \( A^t \) denote the matrix product \( A \cdot A \cdots A \) (\( t \) times).
Definition

The adjacency matrix of a graph with $n$ vertices, is a $n \times n$ matrix $A$ where $A_{ij} = 1$, if there is an edge from vertex $i$ to $j$ and 0 otherwise.

Observation

Let $A^t$ denote the matrix product $A \cdot A \cdots A$ ($t$ times). Then, $(A^t)_{ij}$ is the number of paths of length $t$ from $i$ to $j$. 
An Example

Example

Compute the matrix powers of the adjacency matrix of the following graph:
An Example

Example

Compute the matrix powers of the adjacency matrix of the following graph:
An Example

Example

Compute the matrix powers of the adjacency matrix of the following graph:
An Example

Example

Compute the matrix powers of the adjacency matrix of the following graph:

![Graph Diagram]
Path Theorem

Given a graph $G$ with $n$ vertices and adjacency matrix $A$, there is a path from $s$ to $t$ if and only if $(I + A)^{n-1}$ is non-zero.

Computing $A^n$ - The naive approach

1: $B = I$
2: for ($i = 1$ to $n$) do
3: $B \rightarrow B \cdot (I + A)$
4: end for

Algorithm 4.64: First approach for reachability
Theorem

Given a graph $G$ with $n$ vertices and adjacency matrix $A$, there is a path from $s$ to $t$ if and only if $(I + A)^{n-1}$ is non-zero.

Algorithm 4.65: First approach for reachability

1. $B = I$.
2. for ($i = 1$ to $n$) do
3. $B ightarrow B \cdot (I + A)$.
4. end for
Path Theorem

Theorem

Given a graph G with n vertices and adjacency matrix A, there is a path from s to t if and only if \((I + A)^{n-1}\) is non-zero.
Path Theorem

**Theorem**

*Given a graph G with n vertices and adjacency matrix A, there is a path from s to t if and only if $(I + A)^{n-1}_{st}$ is non-zero.*

**Computing $A^n$ - The naive approach**
Path Theorem

Theorem

Given a graph $G$ with $n$ vertices and adjacency matrix $A$, there is a path from $s$ to $t$ if and only if $(I + A)^{n-1}_{st}$ is non-zero.

Computing $A^n$ - The naive approach
Path Theorem

Theorem

Given a graph $G$ with $n$ vertices and adjacency matrix $A$, there is a path from $s$ to $t$ if and only if $(I + A)_{st}^{n-1}$ is non-zero.

Computing $A^n$ - The naive approach

1: $B = I$. 
Path Theorem

Theorem

Given a graph \( G \) with \( n \) vertices and adjacency matrix \( A \), there is a path from \( s \) to \( t \) if and only if \( (I + A)^{n-1} \) is non-zero.

Computing \( A^n \) - The naive approach

1: \( B = I \).
2: \( \text{for } (i = 1 \text{ to } n) \text{ do} \)
### Theorem

Given a graph $G$ with $n$ vertices and adjacency matrix $A$, there is a path from $s$ to $t$ if and only if $(I + A)^{n-1}_{st}$ is non-zero.

### Computing $A^n$ - The naive approach

1: $B = I$.
2: for $(i = 1$ to $n)$ do
3:   $B \rightarrow B \cdot (I + A)$.  

**Path Theorem**

**Theorem**

*Given a graph G with n vertices and adjacency matrix A, there is a path from s to t if and only if \((I + A)^{n-1}_{st}\) is non-zero.*

**Computing \(A^n\) - The naive approach**

1: \(B = I\).
2: \(\text{for } (i = 1 \text{ to } n) \text{ do}\)
3: \(B \rightarrow B \cdot (I + A)\).
4: \(\text{end for}\)

**Algorithm 4.72:** First approach for reachability
A smarter approach for reachability

1: \( B = I \).
2: for \( i = 1 \) to \( \log n \) do
3: \( B \to B \cdot B \).
4: end for

Algorithm 4.73: Repeated squaring
A smarter approach for reachability

Computing $A^n$ - The smart approach

Algorithm 4.74: Repeated squaring

1: $B = I$
2: for ($i = 1$ to $\log n$) do
3: $B \rightarrow B \cdot B$
4: end for
Computing $A^n$ - The smart approach
Computing $A^n$ - The smart approach

1: $B = I$. 
A smarter approach for reachability

Computing $A^n$ - The smart approach

1: $B = I$.
2: for ($i = 1$ to log $n$) do
A smarter approach for reachability

Computing $A^n$ - The smart approach

1: $B = I$.
2: for ($i = 1$ to $\log n$) do
3: $B \rightarrow B \cdot B$. 
A smarter approach for reachability

Computing $A^n$ - The smart approach

1: $B = I$.
2: for $(i = 1$ to $\log n)$ do
3: \hspace{1em} $B \rightarrow B \cdot B$.
4: end for

**Algorithm 4.79**: Repeated squaring
Some observations

1. A multiplication step is implemented as:
   \[ B_{ij} \rightarrow \sum_k B_{ik} \cdot B_{kj} \]

2. However, the case where \( B \) is a boolean matrix is sufficient for our needs!

3. Accordingly, we can replace matrix multiplication with:
   \[ B_{ij} \rightarrow \lor_k (B_{ik} \land B_{kj}) \]

4. Strategy is called middle-first search, because we find to try to find a vertex \( k \) between vertices \( i \) and \( j \).

5. Strategy is inefficient in terms of time, but efficient in terms of memory.
Some observations

Observation

A multiplication step is implemented as:

\[ B_{ij} \rightarrow \sum_k B_{ik} \cdot B_{kj} \]

However, the case where \( B \) is a boolean matrix is sufficient for our needs!

Accordingly, we can replace matrix multiplication with:

\[ B_{ij} \rightarrow \bigvee_k (B_{ik} \land B_{kj}) \]

Strategy is called middle-first search, because we find to try to find a vertex \( k \) between vertices \( i \) and \( j \).

Strategy is inefficient in terms of time, but efficient in terms of memory.
Some observations

Observation

1. A multiplications step is implemented as:

   \[ B_{ij} \rightarrow \sum_k B_{ik} \cdot B_{kj} \]

   However the case where \( B \) is a boolean matrix is sufficient for our needs! Accordingly, we can replace matrix multiplication with:

   \[ B_{ij} \rightarrow \bigvee_k (B_{ik} \land B_{kj}) \]

   Strategy is called middle-first search, because we find to try to find a vertex \( k \) between vertices \( i \) and \( j \).

4. Strategy is inefficient in terms of time, but efficient in terms of memory.
Some observations

Observation

1. A multiplications step is implemented as:

\[ B_{ij} \rightarrow \sum_k B_{ik} \cdot B_{kj} \]
Observation

1. A multiplications step is implemented as:

   \[ B_{ij} \rightarrow \sum_k B_{ik} \cdot B_{kj} \]

2. However the case where \( B \) is a boolean matrix is sufficient for our needs!
Some observations

**Observation**

1. A multiplications step is implemented as:

   \[ B_{ij} \rightarrow \sum_k B_{ik} \cdot B_{kj} \]

2. However the case where \( B \) is a boolean matrix is sufficient for our needs! Accordingly, we can replace matrix multiplication with:
Some observations

Observation

1. A multiplications step is implemented as:

\[ B_{ij} \rightarrow \sum_k B_{ik} \cdot B_{kj} \]

2. However the case where B is a boolean matrix is sufficient for our needs! Accordingly, we can replace matrix multiplication with:

\[ B_{ij} \rightarrow \bigvee_k \]
Some observations

**Observation**

1. A multiplications step is implemented as:

   \[ B_{ij} \rightarrow \sum_k B_{ik} \cdot B_{kj} \]

2. However the case where \( B \) is a boolean matrix is sufficient for our needs! Accordingly, we can replace matrix multiplication with:

   \[ B_{ij} \rightarrow \bigvee_k (B_{ik} \land B_{kj}) \]
Some observations

Observation

1. A multiplications step is implemented as:

\[ B_{ij} \rightarrow \sum_k B_{ik} \cdot B_{kj} \]

2. However the case where \( B \) is a boolean matrix is sufficient for our needs! Accordingly, we can replace matrix multiplication with:

\[ B_{ij} \rightarrow \bigvee_k (B_{ik} \land B_{kj}) \]

3. Strategy is called middle-first search, because we find to try to find a vertex \( k \) between vertices \( i \) and \( j \).
Some observations

Observation

1. A multiplications step is implemented as:

\[ B_{ij} \rightarrow \sum_k B_{ik} \cdot B_{kj} \]

2. However the case where B is a boolean matrix is sufficient for our needs! Accordingly, we can replace matrix multiplication with:

\[ B_{ij} \rightarrow \bigvee_k (B_{ik} \land B_{kj}) \]

3. Strategy is called middle-first search, because we find to try to find a vertex k between vertices i and j.

4. Strategy is inefficient in terms of time, but efficient in terms of memory.
The All-Pairs Shortest Path problem

The Problem
Given a weighted graph $G$ with weights $w_{ij}$ on edge $e_{ij}$, find the length of the shortest path from vertex $i$ to vertex $j$, for all pairs $i$ and $j$.

Optimality Substructure
Let $p$ denote a shortest path between $s$ and $t$. Let $r$ be an intermediate vertex on $p$. What can you say about the sub-paths of $p$ from $s$ to $r$ and from $r$ to $t$?
The All-Pairs Shortest Path problem

The Problem

- Given a weighted graph $G$ with weights $w_{ij}$ on edge $e_{ij}$ ($W$), find the length of the shortest path from vertex $i$ to vertex $j$, for all pairs $i$ and $j$.

Optimality Substructure

Let $p$ denote a shortest path between $s$ and $t$. Let $r$ be an intermediate vertex on $p$. What can you say about the sub-paths of $p$ from $s$ to $r$ and from $r$ to $t$?
The All-Pairs Shortest Path problem

The Problem

Given a weighted graph $G$ with weights $w_{ij}$ on edge $e_{ij}$.
The All-Pairs Shortest Path problem

The Problem

Given a weighted graph $G$ with weights $w_{ij}$ on edge $e_{ij}$, find the length of the shortest path from vertex $i$ to vertex $j$, for all pairs $i$ and $j$. 
The All-Pairs Shortest Path problem

The Problem
Given a weighted graph $G$ with weights $w_{ij}$ on edge $e_{ij}$ ($W$), find the length of the shortest path from vertex $i$ to vertex $j$, for all pairs $i$ and $j$.

Optimality Substructure
The All-Pairs Shortest Path problem

The Problem

Given a weighted graph $G$ with weights $w_{ij}$ on edge $e_{ij}$ ($W$), find the length of the shortest path from vertex $i$ to vertex $j$, for all pairs $i$ and $j$.

Optimality Substructure

Let $p$ denote a shortest path between $s$ and $t$. 
### The Problem
Given a weighted graph $G$ with weights $w_{ij}$ on edge $e_{ij}$ ($W$), find the length of the shortest path from vertex $i$ to vertex $j$, for all pairs $i$ and $j$.

### Optimality Substructure
Let $p$ denote a shortest path between $s$ and $t$.

Let $r$ be an intermediate vertex on $p$. 
The All-Pairs Shortest Path problem

The Problem
Given a weighted graph $G$ with weights $w_{ij}$ on edge $e_{ij}$ (W), find the length of the shortest path from vertex $i$ to vertex $j$, for all pairs $i$ and $j$.

Optimality Substructure
Let $p$ denote a shortest path between $s$ and $t$.
Let $r$ be an intermediate vertex on $p$.
What can you say about the sub-paths of $p$ from $s$ to $r$ and from $r$ to $t$?
A DP based algorithm

Algorithm 4.80: Repeated squaring for shortest paths
A DP based algorithm

Shortest path algorithm

Function \texttt{SHORTEST-PATHS}(G, W)
A DP based algorithm

Shortest path algorithm

**Function**  \textsc{shortest-paths}(G, W)

1: \( B = W. \)
A DP based algorithm

**Function**  \text{SHORTEST-PATHS}(G, W)

1: \textbf{B} = \text{W}.
2: \textbf{for} \ (i = 1 \ \textbf{to} \ \log n) \ \textbf{do}
#### Function \textsc{Shortest-Paths}(G, W)

1: \textbf{B} = \textbf{W}.
2: \textbf{for} \ (i = 1 \ \textbf{to} \ \log n) \ \textbf{do}
3: \textbf{B} \rightarrow \textbf{B} \cdot \textbf{B}.
A DP based algorithm

Shortest path algorithm

```
Function SHORTEST-PATHS(G, W)
1: B = W.
2: for (i = 1 to log n) do
3:   B → B · B.
4:   {The multiplication in the above step is actually implemented as:
```
A DP based algorithm

Function $\text{SHORTEST-PATHS}(G, W)$
1: $B = W$.
2: for $(i = 1$ to $\log n)$ do
3:   $B \rightarrow B \cdot B$.
4:   {The multiplication in the above step is actually implemented as:
   
   $B_{ij} \rightarrow \min_k (B_{ik} + B_{kj})$.
   
   }

Algorithm 4.86: Repeated squaring for shortest paths
A DP based algorithm

Shortest path algorithm

Function \textsc{Shortest-Paths}(G, W)

1: \textbf{B} = W.
2: \textbf{for} (i = 1 \textbf{to} \log n) \textbf{do}
3: \textbf{B} \rightarrow \textbf{B} \cdot \textbf{B}.
4: \{\text{The multiplication in the above step is actually implemented as:}\}
   \[ B_{ij} \rightarrow \min_k (B_{ik} + B_{kj}). \]
5: \textbf{end for}

\textbf{Algorithm 4.87:} Repeated squaring for shortest paths
Iterative All-Pairs shortest path algorithm

Algorithm 4.88: Implementing the shortest paths algorithm

Review of concepts
The Greedy Approach
Dynamic Programming

Algorithmic Insights
Computational Complexity
Iterative All-Pairs shortest path algorithm

Iterative Implementation

**Function**  
**SHORTEST-PATHS**(*G*, *W*)

```plaintext
1: Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, ..., n$.
2: for ($m = 1$ to $\log n$) do
3:   for ($i = 1$ to $n$) do
4:     for ($j = 1$ to $n$) do
5:       $B_{ij}(m) = B_{ij}(m - 1)$.
6:     for ($k = 1$ to $n$) do
7:       $B_{ij}(m) = \min(B_{ij}(m), B_{ik}(m - 1) + B_{kj}(m - 1))$.
8:   end for
9: end for
10: end for
11: return ($B(\log n)$).
```

Algorithm 4.89: Implementing the shortest paths algorithm
Iterative All-Pairs shortest path algorithm

Iterative Implementation

**Function** `SHORTEST-PATHS(G, W)`

1. Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$. 

Algorithm 4.90: Implementing the shortest paths algorithm
## Iterative Implementation

**Function** \textsc{Shortest-Paths}(G, W)

1. Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
2. \textbf{for} ($m = 1$ \textbf{to} \log n) \textbf{do}
   
   3. \textbf{for} ($i = 1$ \textbf{to} $n$) \textbf{do}
      
      4. \textbf{for} ($j = 1$ \textbf{to} $n$) \textbf{do}
         
         5. $B_{ij}(m) = B_{ij}(m-1)$.
         
         6. \textbf{for} ($k = 1$ \textbf{to} $n$) \textbf{do}
            
            7. $B_{ij}(m) = \min(B_{ij}(m), B_{ik}(m-1) + B_{kj}(m-1))$.
            
            8. \textbf{end for}
         
         9. \textbf{end for}
      
      10. \textbf{end for}
   
   11. \textbf{end for}

12. \textbf{return} $(B(\log n))$. 

Algorithm 4.91: Implementing the shortest paths algorithm
Iterative All-Pairs shortest path algorithm

Iterative Implementation

**Function** \( \text{SHORTEST-PATHS}(G, W) \)

1: Initialize \( B_{ij}(0) = 0 \), for all \( i, j = 1, 2, \ldots n \).
2: \textbf{for} \ (m = 1 \textbf{ to } \log n) \textbf{ do}
3: \textbf{for} \ (i = 1 \textbf{ to } n) \textbf{ do}
Iterative All-Pairs shortest path algorithm

Iterative Implementation

**Function** SHORTEST-PATHS($G$, $W$)
1: Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
2: for $(m = 1$ to $\log n)$ do
3:     for $(i = 1$ to $n)$ do
4:         for $(j = 1$ to $n)$ do
5:             $B_{ij}(m) = B_{ij}(m - 1)$
6:         end for
7:     end for
8: end for
9: return $(B(\log n))$. 

Algorithm 4.93: Implementing the shortest paths algorithm
Iterative All-Pairs shortest path algorithm

Function \textsc{Shortest-Paths}(G, W)
1: Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots, n$.
2: \textbf{for} (\textit{m} = 1 \textbf{to} \log n) \textbf{do}
3: \hspace{1em} \textbf{for} (i = 1 \textbf{to} n) \textbf{do}
4: \hspace{2em} \textbf{for} (j = 1 \textbf{to} n) \textbf{do}
5: \hspace{3em} B_{ij}(m) = B_{ij}(m - 1)
Iterative All-Pairs shortest path algorithm

Iterative Implementation

**Function**  \textsc{shortest-paths}(G, W)

1: Initialize \( B_{ij}(0) = 0 \), for all \( i, j = 1, 2, \ldots n \).
2: \textbf{for} \ (m = 1 \textbf{to} \log n) \textbf{do}
3: \hspace{1em} \textbf{for} \ (i = 1 \textbf{to} n) \textbf{do}
4: \hspace{2em} \textbf{for} \ (j = 1 \textbf{to} n) \textbf{do}
5: \hspace{3em} \( B_{ij}(m) = B_{ij}(m - 1) \).
6: \hspace{2em} \textbf{for} \ (k = 1 \textbf{to} n) \textbf{do}
Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function $\text{SHORTEST-PATHS}(G, W)$

1: Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
2: for $(m = 1$ to $\log n)$ do
3: \hspace{1em} for $(i = 1$ to $n)$ do
4: \hspace{2em} for $(j = 1$ to $n)$ do
5: \hspace{3em} $B_{ij}(m) = B_{ij}(m - 1)$.
6: \hspace{1em} for $(k = 1$ to $n)$ do
7: \hspace{2em} $B_{ij}(m) =$
Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function $\text{SHORTEST-PATHS}(G, W)$

1: Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
2: for $(m = 1$ to $\log n)$ do
3:   for $(i = 1$ to $n)$ do
4:     for $(j = 1$ to $n)$ do
5:       $B_{ij}(m) = B_{ij}(m - 1)$.
6:     for $(k = 1$ to $n)$ do
7:       $B_{ij}(m) = \min(B_{ij}(m), B_{ik}(m - 1) + B_{kj}(m - 1))$. 

Algorithm 4.97: Implementing the shortest paths algorithm
Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function \textsc{Shortest-Paths}(G, W)
1: Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
2: \textbf{for} ($m = 1$ to $\log n$) \textbf{do}
3: \hspace{1em} \textbf{for} ($i = 1$ to $n$) \textbf{do}
4: \hspace{2em} \textbf{for} ($j = 1$ to $n$) \textbf{do}
5: \hspace{3em} $B_{ij}(m) = B_{ij}(m - 1)$.
6: \hspace{2em} \textbf{for} ($k = 1$ to $n$) \textbf{do}
7: \hspace{3em} $B_{ij}(m) = \min(B_{ij}(m), B_{ik}(m - 1) + B_{kj}(m - 1))$.
8: \hspace{1em} \textbf{end for}
9: \textbf{end for}
10: \textbf{end for}
11: \textbf{return} $(B_{ij}(\log n))$. 

Algorithm 4.98: Implementing the shortest paths algorithm
Iterative All-Pairs shortest path algorithm

Algorithmic Insights

Computational Complexity
Iterative Implementation

Function $\text{SHORTEST-PATHS}(G, W)$

1: Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
2: for $(m = 1 \text{ to } \log n)$ do
3:   for $(i = 1 \text{ to } n)$ do
4:     for $(j = 1 \text{ to } n)$ do
5:       $B_{ij}(m) = B_{ij}(m - 1)$.
6:     for $(k = 1 \text{ to } n)$ do
7:       $B_{ij}(m) = \min(B_{ij}(m), B_{ik}(m - 1) + B_{kj}(m - 1))$.
8:     end for
9:   end for
10: end for

Algorithm 4.100: Implementing the shortest paths algorithm
Iterative All-Pairs shortest path algorithm

Iterative Implementation

**Function**  
SHORTEST-PATHS($G, W$)

1: Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
2:  
3:  
4:  
5: $B_{ij}(m) = B_{ij}(m - 1)$.
6:  
7: $B_{ij}(m) = \min(B_{ij}(m), B_{ik}(m - 1) + B_{kj}(m - 1))$.
8:  
9:  
10:  
11: end for

12: return ($B_{ij}(\log n)$).

Algorithm 4.101: Implementing the shortest paths algorithm
Iterative All-Pairs shortest path algorithm

Iterative Implementation

**Function**  \textsc{Shortest-Paths}(G, W)

1: Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
2: \textbf{for} (\(m = 1\) \textbf{to} \(\log n\)) \textbf{do}
3: \quad \textbf{for} (\(i = 1\) \textbf{to} \(n\)) \textbf{do}
4: \quad \quad \textbf{for} (\(j = 1\) \textbf{to} \(n\)) \textbf{do}
5: \quad \quad \quad \(B_{ij}(m) = B_{ij}(m - 1)\).
6: \quad \quad \textbf{for} (\(k = 1\) \textbf{to} \(n\)) \textbf{do}
7: \quad \quad \quad \(B_{ij}(m) = \min(B_{ij}(m), B_{ik}(m - 1) + B_{kj}(m - 1))\).
8: \quad \quad \textbf{end for}
9: \quad \textbf{end for}
10: \end for
11: \textbf{end for}
12: \textbf{return} (B(\log n)).

**Algorithm 4.102:** Implementing the shortest paths algorithm
Analysis

Review of concepts
The Greedy Approach
Dynamic Programming

Algorithmic Insights
Computational Complexity
Time bounds

Computing a specific $B_{ij}$ requires $\Theta(n)$ time.
Time bounds

Computing a specific $B_{ij}$ requires $\Theta(n)$ time.

Computing $\mathbf{B}$ therefore requires $\Theta(n^3)$ time.
Time bounds

Computing a specific $B_{ij}$ requires $\Theta(n)$ time.

Computing $B$ therefore requires $\Theta(n^3)$ time.

It follows that the algorithm takes $\Theta(n^3 \cdot \log n)$ time.
Correctness of Dynamic Programming Algorithms

**In case of a typical dynamic programming algorithm, correctness is self-evident.**

When it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.

The main idea is that after each loop iteration (level of recursion), concrete progress has been made. Such partial guarantees are called loop invariants.

For instance, proving the correctness of the All-Pairs shortest path algorithm would consist of establishing the following two invariants:

1. $B_{ij}(m)$ where $m = 1, 2, \ldots, \log n$ is always an upper bound on the length of the shortest path from $i$ to $j$.
2. After running the outermost for loop $m$ times, $B_{ij}(m)$ equals the length of the shortest path from $i$ to $j$ that consists of at most $2^m$ edges.

We can immediately conclude that the algorithm is complete, when $2^m \geq n$. 

**Algorithmic Insights**

**Computational Complexity**
Correctness of Dynamic Programming Algorithms

**Correctness**

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
3. The main idea is that after each loop iteration (level of recursion), concrete progress has been made. Such partial guarantees are called loop invariants.
4. For instance, proving the correctness of the All-Pairs shortest path algorithm would consist of establishing the following two invariants:
   - \( B_{ij}(m) \), \( m = 1, 2, \ldots, \log n \) is always an upper bound on the length of the shortest path from \( i \) to \( j \).
   - After running the outermost for loop \( m \) times, \( B_{ij}(m) \) equals the length of the shortest path from \( i \) to \( j \) that consists of at most \( 2^m \) edges.
5. We can immediately conclude that the algorithm is complete, when \( 2^m \geq n \).
Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
Correctness of Dynamic Programming Algorithms

Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
Correctness of Dynamic Programming Algorithms

Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
3. The main idea is that after each loop iteration, concrete progress has been made.

Such partial guarantees are called loop invariants.

For instance, proving the correctness of the All-Pairs shortest path algorithm would consist of establishing the following two invariants:

1. \( B_{ij}(m) = \binom{m}{2} \) is always an upper bound on the length of the shortest path from \( i \) to \( j \).\[1\]
2. After running the outermost for loop \( m \) times, \( B_{ij}(m) \) equals the length of the shortest path from \( i \) to \( j \) that consists of at most \( 2^m \) edges.\[2\]

We can immediately conclude that the algorithm is complete, when \( 2^m \geq n \).
Correctness of Dynamic Programming Algorithms

Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
3. The main idea is that after each loop iteration (level of recursion), concrete progress has been made.

For instance, proving the correctness of the All-Pairs shortest path algorithm would consist of establishing the following two invariants:

1. $B_{ij}(m), m = 1, 2, \ldots, \log n$ is always an upper bound on the length of the shortest path from $i$ to $j$.
2. After running the outermost for loop $m$ times, $B_{ij}(m)$ equals the length of the shortest path from $i$ to $j$ that consists of at most $2^m$ edges.

We can immediately conclude that the algorithm is complete, when $2^m \geq n$. 

Algorithmic Insights

Computational Complexity
Correctness of Dynamic Programming Algorithms

Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
3. The main idea is that after each loop iteration (level of recursion), concrete progress has been made. Such partial guarantees are called...
Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
3. The main idea is that after each loop iteration (level of recursion), concrete progress has been made. Such partial guarantees are called *loop invariants*.

For instance, proving the correctness of the All-Pairs shortest path algorithm would consist of establishing the following two invariants:

1. $B_{ij}(m), m = 1, 2, \ldots, \log n$ is always an upper bound on the length of the shortest path from $i$ to $j$.
2. After running the outermost for loop $m$ times, $B_{ij}(m)$ equals the length of the shortest path from $i$ to $j$ that consists of at most $2^m$ edges.

We can immediately conclude that the algorithm is complete when $2^m \geq n$. 

Algorithmic Insights

Computational Complexity
Correctness of Dynamic Programming Algorithms

Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
3. The main idea is that after each loop iteration (level of recursion), concrete progress has been made. Such partial guarantees are called loop invariants.
4. For instance, proving the correctness of the All-Pairs shortest path algorithm would consists of establishing the following two invariants:

\[
B_{ij}(m), m = 1, 2, \ldots, \log n \text{ is always an upper bound on the length of the shortest path from } i \text{ to } j.
\]

\[
\text{After running the outermost for loop } m \text{ times, } B_{ij}(m) \text{ equals the length of the shortest path from } i \text{ to } j \text{ that consists of at most } 2^m \text{ edges.}
\]

5. We can immediately conclude that the algorithm is complete, when \(2^m \geq n\).
Correctness of Dynamic Programming Algorithms

Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
3. The main idea is that after each loop iteration (level of recursion), concrete progress has been made. Such partial guarantees are called loop invariants.
4. For instance, proving the correctness of the All-Pairs shortest path algorithm would consist of establishing the following two invariants:
   1. $B_{ij}(m), m = 1, 2, \ldots, \log n$ is always an upper bound on the length of the shortest path from $i$ to $j$. 
Correctness of Dynamic Programming Algorithms

Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
3. The main idea is that after each loop iteration (level of recursion), concrete progress has been made. Such partial guarantees are called loop invariants.
4. For instance, proving the correctness of the All-Pairs shortest path algorithm would consists of establishing the following two invariants:
   1. $B_{ij}(m), m = 1, 2, \ldots, \log n$ is always an upper bound on the length of the shortest path from $i$ to $j$.
   2. After running the outermost for loop $m$ times,
Correctness of Dynamic Programming Algorithms

Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
3. The main idea is that after each loop iteration (level of recursion), concrete progress has been made. Such partial guarantees are called *loop invariants*.
4. For instance, proving the correctness of the All-Pairs shortest path algorithm would consist of establishing the following two invariants:
   1. $B_{ij}(m), m = 1, 2, \ldots, \log n$ is always an upper bound on the length of the shortest path from $i$ to $j$.
   2. After running the outermost *for* loop $m$ times, $B_{ij}(m)$ equals the length of the shortest path from $i$ to $j$ that consists of at most $2^m$ edges.
Correctness of Dynamic Programming Algorithms

Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
3. The main idea is that after each loop iteration (level of recursion), concrete progress has been made. Such partial guarantees are called *loop invariants*.
4. For instance, proving the correctness of the All-Pairs shortest path algorithm would consist of establishing the following two invariants:
   1. \( B_{ij}(m), m = 1, 2, \ldots, \log n \) is always an upper bound on the length of the shortest path from \( i \) to \( j \).
   2. After running the outermost *for* loop \( m \) times, \( B_{ij}(m) \) equals the length of the shortest path from \( i \) to \( j \) that consists of at most \( 2^m \) edges.
5. We can immediately conclude that the algorithm is complete, when \( 2^m \geq \)
Correctness of Dynamic Programming Algorithms

Correctness

1. In case of a typical dynamic programming algorithm, correctness is self-evident.
2. In the event that it is not, correctness is established through induction on the number of levels of recursion, or the number of types a loop has run.
3. The main idea is that after each loop iteration (level of recursion), concrete progress has been made. Such partial guarantees are called loop invariants.
4. For instance, proving the correctness of the All-Pairs shortest path algorithm would consists of establishing the following two invariants:
   1. $B_{ij}(m), m = 1, 2, \ldots, \log n$ is always an upper bound on the length of the shortest path from $i$ to $j$.
   2. After running the outermost for loop $m$ times, $B_{ij}(m)$ equals the length of the shortest path from $i$ to $j$ that consists of at most $2^m$ edges.
5. We can immediately conclude that the algorithm is complete, when $2^m \geq n$. 