Algorithmic Insights II - Greedy and Dynamic Programming

K. Subramani[1]

[1] Lane Department of Computer Science and Electrical Engineering
West Virginia University

February 16 and February 23, 2015

Outline

Outline

Outline

Review

Review

Algorithmic Insights

Review

Algorithmic Insights

1. Recursion.

Review

Algorithmic Insights

1. Recursion.
2. Divide and Conquer.

Review

Algorithmic Insights

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

Review

Algorithmic Insights

1. Recursion.
2. Divide and Conquer.
3. Greedy.
4. Dynamic Programming.

Review

Algorithmic Insights

1. Recursion.
2. Divide and Conquer.
3. Greedy.
4. Dynamic Programming.
5. Iterative approaches (Rewriting).

Review

Algorithmic Insights

1. Recursion.
2. Divide and Conquer.
3. Greedy.
4. Dynamic Programming.
5. Iterative approaches (Rewriting).
6. Transformations and reductions.

The Greedy Approach

The Greedy Approach

Main Idea

The Greedy Approach

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.

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.

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

The file storage problem

Problem

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.

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.

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

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:

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: $\sum_{i=1}^{k} l_i$.

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 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: $\mathbf{E}[cost] =$

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: $\mathbf{E}[cost] = \frac{1}{n} \cdot \sum_{i=1}^{n} \sum_{j=1}^{i} l_j$.

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: $\mathbf{E}[cost] = \frac{1}{n} \cdot \sum_{i=1}^{n} \sum_{j=1}^{i} l_j$.
5. Different orders of file storage give rise to different expected costs.

The file storage problem

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: $\mathbf{E}[cost] = \frac{1}{n} \cdot \sum_{i=1}^{n} \sum_{j=1}^{i} l_j$.
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?

File storage

File storage

Solution

File storage

Solution

1. The files should be stored in increasing order of length on the tape,

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.

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

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.

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.

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.

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?)

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!

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! The expected cost decreases by:

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! The expected cost decreases by: $\frac{(l_j - l_i)}{n}$.

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! 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

The Minimum Spanning Tree problem

Problem

The Minimum Spanning Tree problem

Problem

Given an edge-weighted, undirected graph $G = \langle V, E, \mathbf{c} \rangle$,

The Minimum Spanning Tree problem

Problem

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

The Minimum Spanning Tree problem

Problem

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

Greedy Approach

The Minimum Spanning Tree problem

Problem

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

Greedy Approach

Order the edges in E in ascending order of weight.

The Minimum Spanning Tree problem

Problem

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

Greedy Approach

Order the edges in E in ascending order of weight.
W.l.o.g. assume that $c(e_1) \leq c(e_2) \leq \ldots c(e_m)$.
$T \rightarrow \emptyset$.
for $(i = 1$ **to** $m)$ **do**

The Minimum Spanning Tree problem

Problem

Given an edge-weighted, undirected graph $G = \langle V, E, \mathbf{c} \rangle$, 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 $G = \langle V, E, \mathbf{c} \rangle$, 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: $T \rightarrow (T \cup \{e_i\})$.

The Minimum Spanning Tree problem

Problem

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

Greedy Approach

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

The Minimum Spanning Tree problem

Problem

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

Greedy Approach

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

Algorithm 3.13: Kruskal's algorithm

Proof of Kruskal

Proof of Kruskal

Definition

Proof of Kruskal

Definition

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

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.

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

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.

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

The fractional knapsack problem

The Problem

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.

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

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** $(i = 1$ **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** $(i = 1$ **to** $n)$ **do**
3: Pack as much of object o_i as you can in the knapsack.
4: **end for**

Correctness

Correctness

Proof of correctness

Correctness

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 α_k and α'_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 α_k and α'_k denote the fractions of the greedy and optimal solutions respectively.
5. Observe that α_k must be greater than α'_k.

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 α_k and α'_k denote the fractions of the greedy and optimal solutions respectively.
5. Observe that α_k must be greater than α'_k.
6. Use an exchange argument.

Scheduling with profits and deadlines

Scheduling with profits and deadlines

The problem

Scheduling with profits and deadlines

The problem

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

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.

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

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.

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** $(i = 1$ **t0** $n)$ **do**

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** $(i = 1$ **t0** $n)$ **do**
5:     **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** $(i = 1$ **t0** $n)$ **do**
5:     **if** $(S \cup \{J_i\}$ is feasible) **then**
6:         $S \rightarrow S \cup \{J_i\}$.

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.

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** $(i = 1$ **t0** $n)$ **do**
5: **if** $(S \cup \{J_i\}$ is feasible) **then**
6: $S \rightarrow S \cup \{J_i\}$.
7: **end if**

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

Order the jobs in descending order of profit.
Assume that $p_1 \geq p_2 \ldots \geq p_n$.
Let $S = \emptyset$.
for $(i = 1$ **t0** $n)$ **do**
    if $(S \cup \{J_i\}$ is feasible) **then**
        $S \rightarrow S \cup \{J_i\}$.
    end if
end for

Algorithm 3.28: Job scheduling

Correctness

Correctness

Theorem

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

The process scheduling problem

The Problem

The process scheduling problem

The Problem

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

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.

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.

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

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 $f_j \leq s_i$.

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 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.

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

The Greedy Algorithm

The Greedy Algorithm

The Greedy Approach

The Greedy Algorithm

The Greedy Approach

The Greedy Algorithm

The Greedy Approach

1: Order the processes in non-decreasing order of start time.

The Greedy Algorithm

The Greedy Approach

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

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**

The Greedy Algorithm

The Greedy Approach

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

The Greedy Algorithm

The Greedy Approach

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

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.

The Greedy Algorithm

The Greedy Approach

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

The Greedy Algorithm

The Greedy Approach

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

The Greedy Algorithm

The Greedy Approach

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

Correctness

The Greedy Algorithm

The Greedy Approach

```
Order the processes in non-decreasing order of start time.
W.l.o.g. assume that s_1 ≤ s_2 ≤ ... s_n.
for (i = 1 to n) do
    Assign P_i to the first available machine.
    if (no machine is available) then
        Assign it to a new machine.
    end if
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

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

Correctness

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

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

The Greedy Algorithm

The Greedy Approach

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

Correctness

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

Let process P_l 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

```
Order the processes in non-decreasing order of start time.
W.l.o.g. assume that s_1 ≤ s_2 ≤ ... s_n.
for (i = 1 to n) do
    Assign P_i to the first available machine.
    if (no machine is available) then
        Assign it to a new machine.
    end if
end for
```

Correctness

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

Let process P_l 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 minimum weight matroid problem

The minimum weight matroid problem

Definition

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:

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 $\mathcal{I}(M)$ of $2^{E(M)}$ that satisfies the following properties:

1. $\emptyset \in \mathcal{I}(M)$.
2. $(Y \in \mathcal{I}(M)$

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. $(Y \in \mathcal{I}(M) \wedge (X \subseteq Y)$

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

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. $(Y \in \mathcal{I}(M) \wedge (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M)$.
3. $(X, Y \in \mathcal{I}(M)$

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. $(Y \in \mathcal{I}(M) \wedge (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M)$.
3. $(X, Y \in \mathcal{I}(M) \wedge (|Y| > |X|)$

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

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. $(Y \in \mathcal{I}(M) \wedge (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M)$.
3. $(X, Y \in \mathcal{I}(M) \wedge (|Y| > |X|) \Rightarrow \exists e \in Y \backslash X$, 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. $(Y \in \mathcal{I}(M) \wedge (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M)$.
3. $(X, Y \in \mathcal{I}(M) \wedge (|Y| > |X|) \Rightarrow \exists e \in Y \backslash X$, 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. $(Y \in \mathcal{I}(M) \wedge (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M)$.
3. $(X, Y \in \mathcal{I}(M) \wedge (|Y| > |X|) \Rightarrow \exists e \in Y \backslash X$, 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

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. $(Y \in \mathcal{I}(M) \wedge (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M)$.
3. $(X, Y \in \mathcal{I}(M) \wedge (|Y| > |X|) \Rightarrow \exists e \in Y \backslash X$, 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 $E(M) = \{s_1, s_2, \ldots, s_n\}$.

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. $(Y \in \mathcal{I}(M) \wedge (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M)$.
3. $(X, Y \in \mathcal{I}(M) \wedge (|Y| > |X|) \Rightarrow \exists e \in Y \setminus X$, 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.

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. $(Y \in \mathcal{I}(M) \wedge (X \subseteq Y) \Rightarrow X \in \mathcal{I}(M)$.
3. $(X, Y \in \mathcal{I}(M) \wedge (|Y| > |X|) \Rightarrow \exists e \in Y \backslash X$, 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.

Find a basis of minimum weight.

The matroid lemma

The matroid lemma

Lemma

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.

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.

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

Dynamic Programming

Main ideas

Dynamic Programming

Main ideas

1. Characterize the structure 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.

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.

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.

The Rod Cutting problem

The Rod Cutting problem

The Problem

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.

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

Length i	1	2	3	4	5	6	7
Price p_i	1	5	8	9	10	17	17

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

Length i	1	2	3	4	5	6	7
Price p_i	1	5	8	9	10	17	17

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

Optimal substructure property

Optimal substructure property

Recurrence

Optimal substructure property

Recurrence

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

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.

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,

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,

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

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,

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

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

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,

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

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

Recurrence (1) can be expressed more succinctly as:

$$r_n \quad =$$

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,

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

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

Recurrence (1) can be expressed more succinctly as:

$$\begin{aligned} r_n &= \max_{1 \le i \le n} (p_i + r_{n-i}) \\ r_0 &= 0 \end{aligned} \tag{2}$$

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,

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

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

Recurrence (1) can be expressed more succinctly as:

$$\begin{aligned} r_n &= \max_{1 \le i \le n} (p_i + r_{n-i}) \\ r_0 &= 0 \end{aligned} \tag{2}$$

Why are Recurrence (1) and Recurrence (2) equivalent?

A recursive implementation

A recursive implementation

Recursive Algorithm

A recursive implementation

Recursive Algorithm

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

1: **if** ($n = 0$) **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)

```
if (n = 0) then
    return(0).
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)

if $(n = 0)$ **then**
    return(0).
end if
$q = -\infty$.
for $(i = 1$ **to** $n)$ **do**

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

```
if (n = 0) then
    return(0).
end if
q = −∞.
for (i = 1 to n) do
    q = max(q,
```

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

if $(n = 0)$ **then**
    return(0).
end if
$q = -\infty$.
for $(i = 1$ **to** $n)$ **do**
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$.

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

if ($n = 0$) **then**
    return(0).
end if
$q = -\infty$.
for ($i = 1$ **to** n) **do**
    $q = \max(q, p[i] +$ CUT-ROD$(p, n - i))$.
end for

Algorithm 4.12: The recursive rod-cutting algorithm

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

if ($n = 0$) **then**
  return(0).
end if
$q = -\infty$.
for ($i = 1$ **to** n) **do**
  $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$.
end for

Algorithm 4.13: The recursive rod-cutting algorithm

Analysis

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: $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$.
7: **end for**

Algorithm 4.14: The recursive rod-cutting algorithm

Analysis

$$T(n) \quad =$$

A recursive implementation

Recursive Algorithm

Function CUT-ROD(p, n)

```
if (n = 0) then
    return(0).
end if
q = −∞.
for (i = 1 to n) do
    q = max(q, p[i] + CUT-ROD(p, n − i)).
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

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 4.16: 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

Analysis of the recursive algorithm

Analysis (contd.)

Analysis of the recursive algorithm

Analysis (contd.)

$$T(n) \quad =$$

Analysis of the recursive algorithm

Analysis (contd.)

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

Analysis of the recursive algorithm

Analysis (contd.)

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

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 approach

The bottom-up algorithm

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ be a new array.

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

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

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ be a new array.
2: $r[0] = 0$.
3: **for** $(j = 1$ **to** $n)$ **do**

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

```
Let r[0 · ·n] be a new array.
r[0] = 0.
for (j = 1 to n) do
    q = −∞.
```

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ 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 $r[0 \cdot \cdot n]$ be a new array.
2: $r[0] = 0$.
3: **for** ($j = 1$ **to** n) **do**
4: $\quad q = -\infty$.
5: $\quad$ **for** ($i = 1$ **to** j) **do**
6: $\quad\quad q = \max(q, p[i] + r[j - i])$.

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

```
Let r[0 · ·n] be a new array.
r[0] = 0.
for (j = 1 to n) do
    q = −∞.
    for (i = 1 to j) do
        q = max(q, p[i] + r[j − i]).
    end for
```

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

```
Let r[0 · ·n] be a new array.
r[0] = 0.
for (j = 1 to n) do
    q = −∞.
    for (i = 1 to j) do
        q = max(q, p[i] + r[j − i]).
    end for
    r[j] = q.
```

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ be a new array.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
  $q = -\infty$.
  for ($i = 1$ **to** j) **do**
    $q = \max(q, p[i] + r[j - i])$.
  end for
  $r[j] = q$.
end for

The Bottom-up approach

The bottom-up algorithm

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ be a new array.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
    $q = -\infty$.
    for ($i = 1$ **to** j) **do**
        $q = \max(q, p[i] + r[j - i])$.
    end for
    $r[j] = q$.
end for
return($r[n]$).

Algorithm 4.29: Bottom-up rod-cutting

Analyzing the bottom-up approach

Analyzing the bottom-up approach

Analysis

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.

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,

$$T(n) =$$

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,

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

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 $T(n) =$

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

Reconstructing the Solution

The bottom-up algorithm with solution

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
2: $r[0] = 0$.

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

1: Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
2: $r[0] = 0$.
3: **for** ($j = 1$ **to** n) **do**

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

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

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

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

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
    $q = -\infty$.
    for ($i = 1$ **to** j) **do**
        if ($q < p[i] + r[j - i]$) **then**

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

```
Let r[0 · ·n] and s[0 · ·n] be new arrays.
r[0] = 0.
for (j = 1 to n) do
    q = −∞.
    for (i = 1 to j) do
        if (q < p[i] + r[j − i]) then
            q = p[i] + r[j − i].
```

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
$r[0] = 0$.
for $(j = 1$ **to** $n)$ **do**
    $q = -\infty$.
    for $(i = 1$ **to** $j)$ **do**
        if $(q < p[i] + r[j - i])$ **then**
            $q = p[i] + r[j - i]$.
            $s[j] = i$. {The unsplittable left side is recorded.}

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
    $q = -\infty$.
    for ($i = 1$ **to** j) **do**
        if ($q < p[i] + r[j - i]$) **then**
            $q = p[i] + r[j - i]$.
            $s[j] = i$. {The unsplittable left side is recorded.}
        end if

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

```
Let r[0 · ·n] and s[0 · ·n] be new arrays.
r[0] = 0.
for (j = 1 to n) do
    q = −∞.
    for (i = 1 to j) do
        if (q < p[i] + r[j − i]) then
            q = p[i] + r[j − i].
            s[j] = i. {The unsplittable left side is recorded.}
        end if
    end for
```

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

```
Let r[0 · ·n] and s[0 · ·n] be new arrays.
r[0] = 0.
for (j = 1 to n) do
    q = −∞.
    for (i = 1 to j) do
        if (q < p[i] + r[j − i]) then
            q = p[i] + r[j − i].
            s[j] = i. {The unsplittable left side is recorded.}
        end if
    end for
    r[j] = q.
```

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

Let $r[0 \cdot \cdot n]$ and $s[0 \cdot \cdot n]$ be new arrays.
$r[0] = 0$.
for ($j = 1$ **to** n) **do**
    $q = -\infty$.
    for ($i = 1$ **to** j) **do**
        if ($q < p[i] + r[j - i]$) **then**
            $q = p[i] + r[j - i]$.
            $s[j] = i$. {The unsplittable left side is recorded.}
        end if
    end for
    $r[j] = q$.
end for

Reconstructing the Solution

The bottom-up algorithm with solution

Function BOTTOM-ROD-CUT(p, n)

```
Let r[0··n] and s[0··n] be new arrays.
r[0] = 0.
for (j = 1 to n) do
    q = −∞.
    for (i = 1 to j) do
        if (q < p[i] + r[j − i]) then
            q = p[i] + r[j − i].
            s[j] = i. {The unsplittable left side is recorded.}
        end if
    end for
    r[j] = q.
end for
return(r[n]).
```

Algorithm 4.45: Bottom-up rod-cutting

Outputting the solution

Outputting the solution

Printing the Solution

Outputting the solution

Printing the Solution

Function PRINT-SOLUTION(p, n)

Outputting the solution

Printing the Solution

Function PRINT-SOLUTION(p, n)

1: **while** ($n > 0$) **do**

Outputting the solution

Printing the Solution

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)

while ($n > 0$) **do**
    print $s[n]$.
    $n = n - s[n]$.

Outputting the solution

Printing the Solution

Function PRINT-SOLUTION(p, n)

```
while (n > 0) do
    print s[n].
    n = n − s[n].
end while
```

Algorithm 4.52: Extracting the solution

The Matrix Chain Multiplication problem

The Matrix Chain Multiplication problem

The Problem

You are required to compute the matrix product $A_1 \cdot A_2 \cdots A_n$,

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 $d_{i-1} \times d_i$,

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.

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

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.

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 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 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

$$T(n) =$$

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

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

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 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 $\Omega(\frac{4^n}{n^{\frac{3}{2}}})$.

Optimality Substructure

Optimality Substructure

Substructure

Optimality Substructure

Substructure

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!

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.

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?)

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?)

Therefore, the optimality substructure applies.

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?)

Therefore, the optimality substructure applies.

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

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?)

Therefore, the optimality substructure applies.

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

$$m[i, j] =$$

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?)

Therefore, the optimality substructure applies.

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

$$m[i, j] = \begin{cases} 0, & \\ & \end{cases}$$

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?)

Therefore, the optimality substructure applies.

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

$$m[i,j] = \begin{cases} 0, & \text{if } j = i \\ & \end{cases}$$

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?)

Therefore, the optimality substructure applies.

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

$$m[i, j] = \begin{cases} 0, & \text{if } j = i \\ \min_{i \le k < j}(m[i, k] + m[k+1, j] \end{cases}$$

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?)

Therefore, the optimality substructure applies.

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

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

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?)

Therefore, the optimality substructure applies.

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

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

Resource analysis

Resource analysis

Analysis

Resource analysis

Analysis

1. For space usage, observe that we need an array $m[i, j]$ and some variable space.

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.

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 $O(n^3)$.

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 $\Theta(n^3)$?

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;

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.

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;

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 $m[i,j]$.

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

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

The Longest Common Subsequence problem

The problem

The Longest Common Subsequence problem

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 Σ.

The Longest Common Subsequence problem

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 Σ.
2. A subsequence of a sequence is defined as as a sequence whose characters occur in the same order as the original sequence

The Longest Common Subsequence problem

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

The Longest Common Subsequence problem

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 Σ.
2. A subsequence of a sequence is defined as 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 Longest Common Subsequence problem

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 Σ.
2. A subsequence of a sequence is defined as 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 Longest Common Subsequence problem

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 Σ.
2. A subsequence of a sequence is defined as 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

The Longest Common Subsequence problem

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 Σ.
2. A subsequence of a sequence is defined as 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

Optimal Substructure

Theorem

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.*

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* $x_m = y_n$,

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}*.*

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* $x_m \neq y_n$*,*

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*

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 $Y = Y_n$.*

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 $x_m \neq y_n$,*

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*

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}.*

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

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.

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,

$$c[i, j] =$$

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,

$$c[i,j] = \begin{cases} 0, & \\ & \\ & \end{cases}$$

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,

$$c[i, j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ & \\ & \end{cases}$$

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,

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

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,

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

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,

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

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,

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

Analysis

Analysis

Resource Analysis

Analysis

Resource Analysis

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

Analysis

Resource Analysis

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

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 $O(m \cdot n)$.

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

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

The Pretty Typesetting problem

The Problem

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.

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

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.

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 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

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

Formulating the cost function

Cost structure

Formulating the cost function

Cost structure

There are 2 key observations to make:

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.

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.

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.

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:

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:

$$s[i,j] \quad = \quad \left\{ \right.$$

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:

$$s[i,j] \quad = \quad \begin{cases} t_{ij}^3, & \\ & \end{cases}$$

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:

$$s[i, j] = \begin{cases} t_{ij}^3, & \text{if } t_{ij} \geq 0 \\ 0, & \\ & \end{cases}$$

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:

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

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:

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

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:

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

Optimal substructure

Optimal substructure

Recursive solution

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

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 $m[1, n]$.

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:

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:

$$m[i, j] \quad = \quad \left\{ \right.$$

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:

$$m[i, j] = \begin{cases} s[i, j], & \\ & \end{cases}$$

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:

$$m[i,j] \quad = \quad \begin{cases} s[i,j], & \text{if } t_{ij} \geq 0 \\ \end{cases}$$

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:

$$m[i,j] = \begin{cases} s[i,j], & \text{if } t_{ij} \geq 0 \\ \min_{i \leq k \leq j}(s[i,k] + m[k+1,j]), & \text{otherwise} \end{cases} \tag{4}$$

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:

$$m[i, j] = \begin{cases} s[i, j], & \text{if } t_{ij} \geq 0 \\ \min_{i \leq k \leq j}(s[i, k] + m[k+1, j]), & \text{otherwise} \end{cases} \tag{4}$$

Analyze the resources of the above algorithm.

The Reachability problem

The Reachability problem

The Problem

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

Graph Exploration

The Exploration Algorithm

Function EXPLORE(G, s, t)

Graph Exploration

The Exploration Algorithm

Function $\textsc{Explore}(G, s, t)$

1: $Q = \{s\}$.

Graph Exploration

The Exploration Algorithm

Function EXPLORE(G, s, t)

1: $Q = \{s\}$.
2: **while** $(Q \neq \emptyset)$ **do**

Graph Exploration

The Exploration Algorithm

Function $\textsc{Explore}(G, s, t)$

$Q = \{s\}$.
while $(Q \neq \emptyset)$ **do**
    Remove a vertex u from Q.

Graph Exploration

The Exploration Algorithm

Function 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.$\}$

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. $\{$If $u = t$, then t is reachable from u.$\}$
4: Mark u.

Graph Exploration

The Exploration Algorithm

Function $\textsc{Explore}(G, s, t)$

```
Q = {s}.
while (Q ≠ ∅) do
    Remove a vertex u from Q. {If u = t, then t is reachable from u.}
    Mark u.
    for (all unmarked neighbors v of u) do
```

Graph Exploration

The Exploration Algorithm

Function EXPLORE(G, s, t)

$Q = \{s\}$.
while $(Q \neq \emptyset)$ **do**
    Remove a vertex u from Q. $\{$If $u = t$, then t is reachable from u.$\}$
    Mark u.
    for (all unmarked neighbors v of u) **do**
        Insert v into Q.

Graph Exploration

The Exploration Algorithm

Function EXPLORE(G, s, t)

$Q = \{s\}$.
while $(Q \neq \emptyset)$ **do**
    Remove a vertex u from Q. $\{$If $u = t$, then t is reachable from u.$\}$
    Mark u.
    for (all unmarked neighbors v of u) **do**
        Insert v into Q.
    end for

Graph Exploration

The Exploration Algorithm

Function EXPLORE(G, s, t)

```
Q = {s}.
while (Q ≠ ∅) do
    Remove a vertex u from Q. {If u = t, then t is reachable from u.}
    Mark u.
    for (all unmarked neighbors v of u) do
        Insert v into Q.
    end for
end while
```

Algorithm 4.63: The generic algorithm

Two common search techniques

Two common search techniques

Breadth-first Search

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

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

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

Middle First Search

Definition

Middle First Search

Definition

The adjacency matrix of a graph with n vertices, is a $n \times n$ matrix $\mathbf{A}$ where $A_{ij} = 1$, if there is an edge from vertex i to j and 0 otherwise.

Middle First Search

Definition

The adjacency matrix of a graph with n vertices, is a $n \times n$ matrix $\mathbf{A}$ where $A_{ij} = 1$, if there is an edge from vertex i to j and 0 otherwise.

Observation

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 $A \cdot A \cdots A$ (t times).

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 $A \cdot A \cdots A$ (t times).

Middle First Search

Definition

The adjacency matrix of a graph with n vertices, is a $n \times n$ matrix $\mathbf{A}$ 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

An Example

Example

An Example

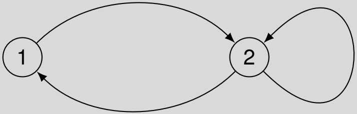
Example

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:



Path Theorem

Path Theorem

Theorem

Path Theorem

Theorem

Given a graph G with n vertices and adjacency matrix $\mathbf{A}$, there is a path from s to t if and only if $(\mathbf{I} + \mathbf{A})^{n-1}_{st}$ is non-zero.

Path Theorem

Theorem

Given a graph G with n vertices and adjacency matrix $\mathbf{A}$, *there is a path from s to t if and only if* $(\mathbf{I}+\mathbf{A})^{n-1}_{st}$ *is non-zero.*

Computing A^n - The naive approach

Path Theorem

Theorem

Given a graph G with n vertices and adjacency matrix $\mathbf{A}$*, there is a path from s to t if and only if* $(\mathbf{I} + \mathbf{A})^{n-1}_{st}$ *is non-zero.*

Computing A^n - The naive approach

Path Theorem

Theorem

Given a graph G with n vertices and adjacency matrix $\mathbf{A}$, there is a path from s to t if and only if $(\mathbf{I}+\mathbf{A})_{st}^{n-1}$ is non-zero.

Computing A^n - The naive approach

1: $\mathbf{B} = \mathbf{I}$.

Path Theorem

Theorem

Given a graph G with n vertices and adjacency matrix $\mathbf{A}$*, there is a path from s to t if and only if* $(\mathbf{I}+\mathbf{A})^{n-1}_{st}$ *is non-zero.*

Computing A^n - The naive approach

1: $\mathbf{B} = \mathbf{I}$.
2: **for** $(i = 1$ **to** $n)$ **do**

Path Theorem

Theorem

Given a graph G with n vertices and adjacency matrix $\mathbf{A}$*, there is a path from s to t if and only if* $(\mathbf{I}+\mathbf{A})^{n-1}_{st}$ *is non-zero.*

Computing A^n - The naive approach

1: $\mathbf{B} = \mathbf{I}$.
2: **for** $(i = 1$ **to** $n)$ **do**
3: $\quad\mathbf{B} \rightarrow \mathbf{B} \cdot (\mathbf{I}+\mathbf{A})$.

Path Theorem

Theorem

Given a graph G with n vertices and adjacency matrix $\mathbf{A}$*, there is a path from s to t if and only if* $(\mathbf{I}+\mathbf{A})^{n-1}_{st}$ *is non-zero.*

Computing A^n - The naive approach

$\mathbf{B} = \mathbf{I}$.
for $(i = 1$ **to** $n)$ **do**
    $\mathbf{B} \rightarrow \mathbf{B} \cdot (\mathbf{I}+\mathbf{A})$.
end for

Algorithm 4.72: First approach for reachability

A smarter approach for reachability

A smarter approach for reachability

Computing A^n - The smart approach

A smarter approach for reachability

Computing A^n - The smart approach

A smarter approach for reachability

Computing A^n - The smart approach

1: $\mathbf{B} = \mathbf{I}$.

A smarter approach for reachability

Computing A^n - The smart approach

1: $\mathbf{B} = \mathbf{I}$.
2: **for** ($i = 1$ **to** $\log n$) **do**

A smarter approach for reachability

Computing A^n - The smart approach

1: $\mathbf{B} = \mathbf{I}$.
2: **for** ($i = 1$ **to** $\log n$) **do**
3: $\quad \mathbf{B} \rightarrow \mathbf{B} \cdot \mathbf{B}$.

A smarter approach for reachability

Computing A^n - The smart approach

1: $\mathbf{B} = \mathbf{I}$.
2: **for** ($i = 1$ **to** $\log n$) **do**
3: $\quad \mathbf{B} \rightarrow \mathbf{B} \cdot \mathbf{B}$.
4: **end for**

Algorithm 4.79: Repeated squaring

Some observations

Some observations

Observation

Some observations

Observation

1. *A multiplications step is implemented as:*

Some observations

Observation

1. *A multiplications step is implemented as:*

$$B_{ij} \to \sum_k B_{ik} \cdot B_{kj}$$

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!*

Some observations

Observation

1. *A multiplications step is implemented as:*

$$B_{ij} \to \sum_k B_{ik} \cdot B_{kj}$$

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:*

$$B_{ij} \rightarrow \bigvee_k (B_{ik} \wedge B_{kj})$$

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:*

$$B_{ij} \rightarrow \bigvee_{k} (B_{ik} \wedge B_{kj})$$

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:*

$$B_{ij} \to \sum_k B_{ik} \cdot B_{kj}$$

2. *However the case where* **B** *is a boolean matrix is sufficient for our needs! Accordingly, we can replace matrix multiplication with:*

$$B_{ij} \to \bigvee_k (B_{ik} \wedge B_{kj})$$

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 All-Pairs Shortest Path problem

The Problem

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} (**W**), 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 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.

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

A DP based algorithm

Shortest path algorithm

Function SHORTEST-PATHS(G, W)

A DP based algorithm

Shortest path algorithm

Function SHORTEST-PATHS(G, W)

1: $\mathbf{B} = \mathbf{W}$.

A DP based algorithm

Shortest path algorithm

Function SHORTEST-PATHS(G, W)

1: $\mathbf{B} = \mathbf{W}$.
2: **for** ($i = 1$ **to** $\log n$) **do**

A DP based algorithm

Shortest path algorithm

Function SHORTEST-PATHS(G, W)

1: $\mathbf{B} = \mathbf{W}$.
2: **for** ($i = 1$ **to** $\log n$) **do**
3: $\quad \mathbf{B} \rightarrow \mathbf{B} \cdot \mathbf{B}$.

A DP based algorithm

Shortest path algorithm

Function SHORTEST-PATHS(G, W)

1: $\mathbf{B} = \mathbf{W}$.
2: **for** ($i = 1$ **to** $\log n$) **do**
3: $\quad \mathbf{B} \rightarrow \mathbf{B} \cdot \mathbf{B}$.
4: $\quad$ {The multiplication in the above step is actually implemented as:

A DP based algorithm

Shortest path algorithm

Function SHORTEST-PATHS(G, W)

1: $\mathbf{B} = \mathbf{W}$.
2: **for** ($i = 1$ **to** $\log n$) **do**
3: $\mathbf{B} \rightarrow \mathbf{B} \cdot \mathbf{B}$.
4: {The multiplication in the above step is actually implemented as:

$$B_{ij} \rightarrow \min_k (B_{ik} + B_{kj}).$$

}

A DP based algorithm

Shortest path algorithm

Function SHORTEST-PATHS(G, W)

1: $\mathbf{B} = \mathbf{W}$.
2: **for** ($i = 1$ **to** $\log n$) **do**
3: $\quad \mathbf{B} \rightarrow \mathbf{B} \cdot \mathbf{B}$.
4: $\quad$ {The multiplication in the above step is actually implemented as:

$$B_{ij} \rightarrow \min_{k}(B_{ik} + B_{kj}).$$

$\quad$ }
5: **end for**

Algorithm 4.87: Repeated squaring for shortest paths

Iterative All-Pairs shortest path algorithm

Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function SHORTEST-PATHS(G, W)

Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function SHORTEST-PATHS(G, W)

1: Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.

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**

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**

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**

Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function SHORTEST-PATHS(G, W)

Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
for $(m = 1$ **to** $\log n)$ **do**
  for $(i = 1$ **to** $n)$ **do**
    for $(j = 1$ **to** $n)$ **do**
      $B_{ij}(m) = B_{ij}(m-1)$.

Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function SHORTEST-PATHS(G, W)

Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
for $(m = 1$ **to** $\log n)$ **do**
  for $(i = 1$ **to** $n)$ **do**
    for $(j = 1$ **to** $n)$ **do**
      $B_{ij}(m) = B_{ij}(m-1)$.
      for $(k = 1$ **to** $n)$ **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:             **for** $(k = 1$ **to** $n)$ **do**
7:                 $B_{ij}(m) =$

Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function SHORTEST-PATHS(G, W)

Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
for $(m = 1$ **to** $\log n)$ **do**
  for $(i = 1$ **to** $n)$ **do**
    for $(j = 1$ **to** $n)$ **do**
      $B_{ij}(m) = B_{ij}(m - 1)$.
      for $(k = 1$ **to** $n)$ **do**
        $B_{ij}(m) = \min(B_{ij}(m), B_{ik}(m - 1) + B_{kj}(m - 1))$.

Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function SHORTEST-PATHS(G, W)

Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
for $(m = 1$ **to** $\log n)$ **do**
    for $(i = 1$ **to** $n)$ **do**
        for $(j = 1$ **to** $n)$ **do**
            $B_{ij}(m) = B_{ij}(m-1)$.
            for $(k = 1$ **to** $n)$ **do**
                $B_{ij}(m) = \min(B_{ij}(m), B_{ik}(m-1) + B_{kj}(m-1))$.
            end for

Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function SHORTEST-PATHS(G, W)

```
Initialize B_ij(0) = 0, for all i, j = 1, 2, ... n.
for (m = 1 to log n) do
    for (i = 1 to n) do
        for (j = 1 to n) do
            B_ij(m) = B_ij(m − 1).
            for (k = 1 to n) do
                B_ij(m) = min(B_ij(m), B_ik(m − 1) + B_kj(m − 1)).
            end for
        end for
```

Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function SHORTEST-PATHS(G, W)

Initialize $B_{ij}(0) = 0$, for all $i, j = 1, 2, \ldots n$.
for $(m = 1$ **to** $\log n)$ **do**
    for $(i = 1$ **to** $n)$ **do**
        for $(j = 1$ **to** $n)$ **do**
            $B_{ij}(m) = B_{ij}(m-1)$.
            for $(k = 1$ **to** $n)$ **do**
                $B_{ij}(m) = \min(B_{ij}(m), B_{ik}(m-1) + B_{kj}(m-1))$.
            end for
        end for
    end for

Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function SHORTEST-PATHS(G, W)

```
Initialize B_ij(0) = 0, for all i, j = 1, 2, ... n.
for (m = 1 to log n) do
    for (i = 1 to n) do
        for (j = 1 to n) do
            B_ij(m) = B_ij(m − 1).
            for (k = 1 to n) do
                B_ij(m) = min(B_ij(m), B_ik(m − 1) + B_kj(m − 1)).
            end for
        end for
    end for
end for
```

Iterative All-Pairs shortest path algorithm

Iterative Implementation

Function SHORTEST-PATHS(G, W)

```
Initialize B_ij(0) = 0, for all i, j = 1, 2, ... n.
for (m = 1 to log n) do
    for (i = 1 to n) do
        for (j = 1 to n) do
            B_ij(m) = B_ij(m − 1).
            for (k = 1 to n) do
                B_ij(m) = min(B_ij(m), B_ik(m − 1) + B_kj(m − 1)).
            end for
        end for
    end for
end for
return(B(log n)).
```

Algorithm 4.102: Implementing the shortest paths algorithm

Analysis

Analysis

Time bounds

Computing a specific B_{ij} requires $\Theta(n)$ time.

Analysis

Time bounds

Computing a specific B_{ij} requires $\Theta(n)$ time.

Computing **B** therefore requires $\Theta(n^3)$ time.

Analysis

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

Correctness of Dynamic Programming Algorithms

Correctness

Correctness of Dynamic Programming Algorithms

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

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.

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 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.*

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:

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.

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 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.

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 $2^m \geq$

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 $2^m \geq n$.