Algorithm Design IX

Dynamic Programming I

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Shortest Paths in DAGs, Revisited

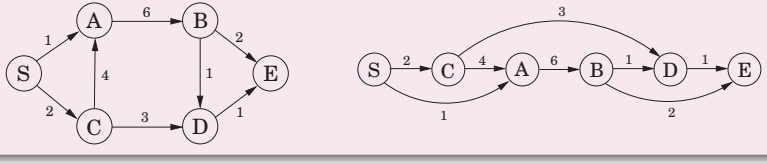
Shortest Paths in Dags, Revisited

The special distinguishing feature of a DAG is that its nodes can be linearized.

Shortest Paths in Dags, Revisited

The special distinguishing feature of a DAG is that its nodes can be linearized.

If compute `dist` values in the left-to-right order, by the time get to a node v, we already have all the information to compute `dist`(v).



Shortest Paths in Dags, Revisited

Initialize all $dist(.)$ value to ∞;
$dist(s)$=0;
for *each* $v \in V \backslash \{s\}$, *in linearized order* **do**
  $dist(v) = min_{(u,v) \in E}\{dist(u) + l(u,v)\}$;
end

Shortest Paths in Dags, Revisited

Initialize all $dist(.)$ value to ∞;
$dist(s)$=0;
for *each* $v \in V \backslash \{s\}$, *in linearized order* **do**
  $dist(v) = min_{(u,v)\in E}\{dist(u) + l(u,v)\}$;
end

This algorithm is solving a collection of subproblems, $\{dist(u) \mid u \in V\}$

This algorithm is solving a collection of subproblems, $\{\texttt{dist}(u) : u \in V\}$.

This algorithm is solving a collection of subproblems, $\{\texttt{dist}(u) : u \in V\}$.

Start with the smallest of them, $\texttt{dist}(s)$.

This algorithm is solving a collection of subproblems, $\{\texttt{dist}(u) : u \in V\}$.

Start with the smallest of them, $\texttt{dist}(s)$.Then proceed with progressively "larger" subproblems, where a subproblem is large if a lot of other subproblems is solved before get to it.

This algorithm is solving a collection of subproblems, $\{\texttt{dist}(u) : u \in V\}$.

Start with the smallest of them, $\texttt{dist}(s)$.Then proceed with progressively "larger" subproblems, where a subproblem is large if a lot of other subproblems is solved before get to it.

Dynamic programming is a powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling smallest first, using the answers to small problems to figure out larger ones, until the whole lot of them is solved.

This algorithm is solving a collection of subproblems, $\{\texttt{dist}(u) : u \in V\}$.

Start with the smallest of them, $\texttt{dist}(s)$.Then proceed with progressively "larger" subproblems, where a subproblem is large if a lot of other subproblems is solved before get to it.

Dynamic programming is a powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling smallest first, using the answers to small problems to figure out larger ones, until the whole lot of them is solved.

In dynamic programming we are not given a DAG; the DAG is implicit.

Longest Increasing Subsequences

The Problem

The input of the longest increasing subsequence problem is a sequence of numbers $a_1, \ldots, a_n$.

The Problem

The input of the longest increasing subsequence problem is a sequence of numbers $a_1, \ldots, a_n$.

A subsequence is any subset of these numbers taken in order, of the form

$$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

The Problem

The input of the longest increasing subsequence problem is a sequence of numbers $a_1, \ldots, a_n$.

A subsequence is any subset of these numbers taken in order, of the form

$$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

An increasing subsequence is one in which the numbers are getting strictly larger.

The Problem

The input of the longest increasing subsequence problem is a sequence of numbers $a_1, \ldots, a_n$.

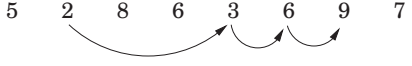
A subsequence is any subset of these numbers taken in order, of the form

$$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

An increasing subsequence is one in which the numbers are getting strictly larger.

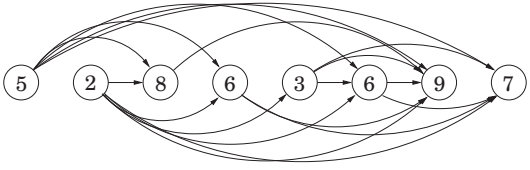
The task is to find the increasing subsequence of greatest length.



Graph Reformulation

Create a graph of all permissible transitions:

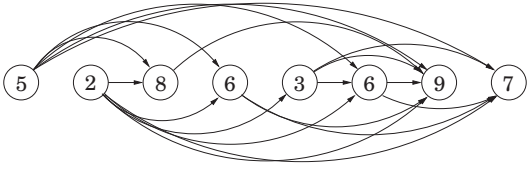
- a node i for each element a_i,
- a directed edge (i, j) if possible for a_i and a_j to be consecutive elements in an increasing subsequence: $i < j$ and $a_i < a_j$.



Graph Reformulation

Create a graph of all permissible transitions:

- a node i for each element a_i,
- a directed edge (i, j) if possible for a_i and a_j to be consecutive elements in an increasing subsequence: $i < j$ and $a_i < a_j$.

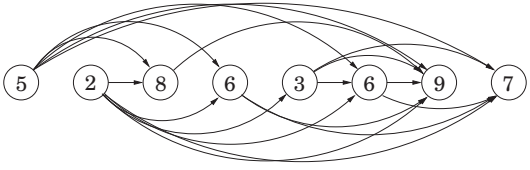


This graph $G = (V, E)$ is a DAG, since all edges (i, j) have $i < j$.

Graph Reformulation

Create a graph of all permissible transitions:

- a node i for each element a_i,
- a directed edge (i, j) if possible for a_i and a_j to be consecutive elements in an increasing subsequence: $i < j$ and $a_i < a_j$.



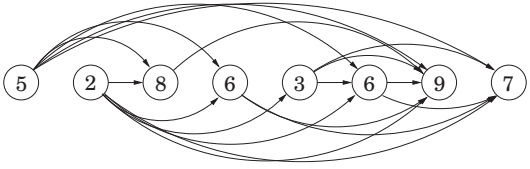
This graph $G = (V, E)$ is a DAG, since all edges (i, j) have $i < j$.

There is a one-to-one correspondence between increasing subsequences and paths in this DAG.

Graph Reformulation

Create a graph of all permissible transitions:

- a node i for each element a_i,
- a directed edge (i, j) if possible for a_i and a_j to be consecutive elements in an increasing subsequence: $i < j$ and $a_i < a_j$.



This graph $G = (V, E)$ is a DAG, since all edges (i, j) have $i < j$.

There is a one-to-one correspondence between increasing subsequences and paths in this DAG.

Therefore, the goal is to find the longest path in the DAG!

The Algorithm

for $j = 1$ *to* n **do**
  $L(j) = 1 + \max\{L(i) \mid (i, j) \in E\}$;
end
`return(`$\max_j L(j)$`);`

The Algorithm

for $j = 1$ *to* n **do**
  $L(j) = 1 + \max\{L(i) \mid (i, j) \in E\}$;
end
`return(`$\max_j L(j)$`);`

$L(j)$ is the length of the longest path - the longest increasing subsequence - ending at j (plus 1).

for $j = 1$ *to* n **do**
  $L(j) = 1 + \max\{L(i) \mid (i, j) \in E\}$;
end
`return(`$\max_j L(j)$`);`

$L(j)$ is the length of the longest path - the longest increasing subsequence - ending at j (plus 1).

If there are no edges into j , we take zero.

The Algorithm

for $j = 1$ *to* n **do**
  $L(j) = 1 + \max\{L(i) \mid (i, j) \in E\}$;
end
`return(`$\max_j L(j)$`);`

$L(j)$ is the length of the longest path - the longest increasing subsequence - ending at j (plus 1).

If there are no edges into j , we take zero.

The final answer is the largest $L(j)$, since any ending position is allowed.

Q: How long does this step take?

Q: How long does this step take?

It requires the predecessors of j to be known;

Q: How long does this step take?

It requires the predecessors of j to be known; for this the adjacency list of the reverse graph G^R, constructible in linear time is handy.

Q: How long does this step take?

It requires the predecessors of j to be known; for this the adjacency list of the reverse graph G^R, constructible in linear time is handy.

The computation of $L(j)$ then takes time proportional to the indegree of j, giving an overall running time linear in $|E|$.

Q: How long does this step take?

It requires the predecessors of j to be known; for this the adjacency list of the reverse graph G^R, constructible in linear time is handy.

The computation of $L(j)$ then takes time proportional to the indegree of j, giving an overall running time linear in $|E|$.

This is at most $O(n^2)$, the maximum being when the input array is sorted in increasing order.

This Is Dynamic Programming

In order to solve our original problem, we have defined a collection of subproblems $\{L(j) \mid 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

This Is Dynamic Programming

In order to solve our original problem, we have defined a collection of subproblems $\{L(j) \mid 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

> There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

This Is Dynamic Programming

In order to solve our original problem, we have defined a collection of subproblems $\{L(j) \mid 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

> There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

In our case, each subproblem is solved using the relation

$$L(j) = 1 + \max\{L(i) | (i, j) \in E\}$$

Why Not Recursion

The formula for $L(j)$ also suggests recursive algorithm.

Why Not Recursion

The formula for $L(j)$ also suggests recursive algorithm.

Recursion is a very bad idea: the resulting procedure would require exponential time!

Why Not Recursion

The formula for $L(j)$ also suggests recursive algorithm.

Recursion is a very bad idea: the resulting procedure would require exponential time!

- e.g. (i, j) for all $i < j$. The formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

Why Not Recursion

The formula for $L(j)$ also suggests recursive algorithm.

Recursion is a very bad idea: the resulting procedure would require exponential time!

- e.g. (i, j) for all $i < j$. The formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

Why did recursion work so well with divide-and-conquer? In divide-and-conquer, a problem is expressed in terms of subproblems that are substantially smaller, say, half the size.

Why Not Recursion

The formula for $L(j)$ also suggests recursive algorithm.

Recursion is a very bad idea: the resulting procedure would require exponential time!

- e.g. (i, j) for all $i < j$. The formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

Why did recursion work so well with divide-and-conquer? In divide-and-conquer, a problem is expressed in terms of subproblems that are substantially smaller, say, half the size.

In a dynamic programming, a problem is reduced to subproblems that are slightly smaller. Thus the full recursion tree has polynomial depth and an exponential number of nodes.

Edit Distance

The problem

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by.

The problem

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by.

Q: What is the appropriate notion of closeness in this case?

The problem

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by.

Q: What is the appropriate notion of closeness in this case?

A natural measure of the distance between two strings is the extent to which they can be aligned, or matched up.

The problem

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by.

Q: What is the appropriate notion of closeness in this case?

A natural measure of the distance between two strings is the extent to which they can be aligned, or matched up.

Technically, an alignment is simply a way of writing the strings one above the other.

S	–	N	O	W		Y
S	U	N	N	–		Y

Cost: 3

–	S	N	O	W	–	Y
S	U	N	–	–	N	Y

Cost: 5

The problem

S	–	N	O	W		Y
S	U	N	N	–		Y

Cost: 3

	–	S	N	O	W	–	Y
	S	U	N	–	–	N	Y

Cost: 5

The cost of an alignment is the number of columns in which the letters differ.

The problem

S	–	N	O	W		Y
S	U	N	N	–		Y

Cost: 3

–	S	N	O	W	–	Y
S	U	N	–	–	N	Y

Cost: 5

The cost of an alignment is the number of columns in which the letters differ.

The edit distance between two strings is the cost of their best possible alignment.

The problem

S	–	N	O	W		Y
S	U	N	N	–		Y

Cost: 3

	–	S	N	O	W	–	Y
	S	U	N	–	–	N	Y

Cost: 5

The cost of an alignment is the number of columns in which the letters differ.

The edit distance between two strings is the cost of their best possible alignment.

Edit distance is so named because it can also be thought of as the minimum number of edits.

The problem

SHANGHAI JIAO TONG UNIVERSITY

S	–	N	O	W		Y
S	U	N	N	–		Y

Cost: 3

	–	S	N	O	W	–	Y
	S	U	N	–	–	N	Y

Cost: 5

The cost of an alignment is the number of columns in which the letters differ.

The edit distance between two strings is the cost of their best possible alignment.

Edit distance is so named because it can also be thought of as the minimum number of edits.

The number of insertions, deletions, and substitutions of characters needed to transform the first string into the second.

A Dynamic Programming Solution

What are the subproblems?

What are the subproblems?

The goal is to find the edit distance between two strings, $x[1, \ldots, m]$ and $y[1, \ldots, n]$.

What are the subproblems?

The goal is to find the edit distance between two strings, $x[1, \ldots, m]$ and $y[1, \ldots, n]$.

For every i, j with $1 \leq i \leq m$ and $1 \leq j \leq n$, let $E(i, j)$: the edit distance between some prefix of the first string, $x[1, \ldots, i]$, and some prefix of the second, $y[1, \ldots, j]$.

What are the subproblems?

The goal is to find the edit distance between two strings, $x[1, \ldots, m]$ and $y[1, \ldots, n]$.

For every i, j with $1 \leq i \leq m$ and $1 \leq j \leq n$, let $E(i, j)$: the edit distance between some prefix of the first string, $x[1, \ldots, i]$, and some prefix of the second, $y[1, \ldots, j]$.

$$E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), \texttt{diff}(i, j) + E(i-1, j-1)\}$$

where $\texttt{diff}(i, j)$ is defined to be 0 if $x[i] = y[j]$ and 1 otherwise.

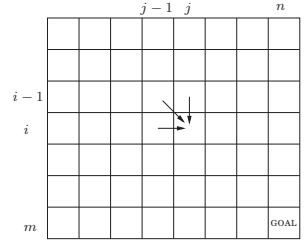
An Example

Edit distance between EXPONENTIAL and POLYNOMIAL, subproblem $E(4,3)$ corresponds to the prefixes EXPO and POL. The rightmost column of their best alignment must be one of the following:

$$\begin{matrix}\mathtt{O}\\ \mathtt{-}\end{matrix} \quad \text{or} \quad \begin{matrix}\mathtt{-}\\ \mathtt{L}\end{matrix} \quad \text{or} \quad \begin{matrix}\mathtt{O}\\ \mathtt{L}\end{matrix}$$

Thus, $E(4,3) = \min\{1 + E(3,3), 1 + E(4,2); 1 + E(3,2)\}$.

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

The Algorithm

for $i = 0$ *to* m **do**
  $E(i, 0) = i$;
end
for $j = 1$ *to* n **do**
  $E(0, j) = j$;
end
for $i = 1$ *to* m **do**
  for $j = 1$ *to* m **do**
    $E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), \texttt{diff}(i, j) + E(i-1, j-1)\}$;
  end
end
`return(`$E(m, n)$`);`

The Algorithm

for $i = 0$ *to* m **do**
  $E(i, 0) = i$;
end
for $j = 1$ *to* n **do**
  $E(0, j) = j$;
end
for $i = 1$ *to* m **do**
  for $j = 1$ *to* m **do**
    $E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), \texttt{diff}(i, j) + E(i-1, j-1)\}$;
  end
end
`return(`$E(m, n)$`);`

The over running time is $O(m \cdot n)$.

Finding the right subproblem takes creativity and experimentation.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.The number of subproblems is therefore linear.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.The number of subproblems is therefore linear.
- The input is $x_1, \ldots x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots y_j$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.The number of subproblems is therefore linear.
- The input is $x_1, \ldots x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots y_j$. The number of subproblems is $O(mn)$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.The number of subproblems is therefore linear.
- The input is $x_1, \ldots x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots y_j$. The number of subproblems is $O(mn)$.
- The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.The number of subproblems is therefore linear.
- The input is $x_1, \ldots x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots y_j$. The number of subproblems is $O(mn)$.
- The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$. The number of subproblems is $O(n^2)$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.The number of subproblems is therefore linear.
- The input is $x_1, \ldots x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots y_j$. The number of subproblems is $O(mn)$.
- The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$. The number of subproblems is $O(n^2)$.
- The input is a rooted tree. A subproblem is a rooted subtree.

Knapsack

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or "knapsack") will hold a total weight of at most W pounds.

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or "knapsack") will hold a total weight of at most W pounds.

There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$.

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or "knapsack") will hold a total weight of at most W pounds.

There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$.

Q: What's the most valuable combination of items he can put into his bag?

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or "knapsack") will hold a total weight of at most W pounds.

There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$.

Q: What's the most valuable combination of items he can put into his bag?

There are two versions of this problem:

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or "knapsack") will hold a total weight of at most W pounds.

There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$.

Q: What's the most valuable combination of items he can put into his bag?

There are two versions of this problem:

- there are unlimited quantities of each item available;

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or "knapsack") will hold a total weight of at most W pounds.

There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$.

Q: What's the most valuable combination of items he can put into his bag?

There are two versions of this problem:

- there are unlimited quantities of each item available;
- there is one of each item.

Knapsack with Repetition

For every $w \leq W$ let

$$K(w) = \text{ maximum value achievable with a knapsack of capacity } w$$

Knapsack with Repetition

For every $w \leq W$ let

$$K(w) = \text{ maximum value achievable with a knapsack of capacity } w$$

We express this in terms of smaller subproblems:

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$$

Knapsack with Repetition

For every $w \leq W$ let

$$K(w) = \text{ maximum value achievable with a knapsack of capacity } w$$

We express this in terms of smaller subproblems:

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$$

$K(0) = 0$;
for $w = 1$ *to* W **do**
  $K(w) = \max_{i:w_i \leq w}\{K(w - w_i) + v_i\}$;
end
`return` $(K(W))$;

Knapsack with Repetition

For every $w \leq W$ let

$$K(w) = \text{ maximum value achievable with a knapsack of capacity } w$$

We express this in terms of smaller subproblems:

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$$

$K(0) = 0$;
for $w = 1$ *to* W **do**
  $K(w) = \max_{i:w_i \leq w}\{K(w - w_i) + v_i\}$;
end
`return` $(K(W))$;

The over running time is $O(n \cdot W)$.

Example: Knapsack with Repetition

Take $W = 10$, and

Item	Weight	Value
1	6	$30
2	3	$14
3	4	$16
4	2	$9

and there are unlimited quantities of each item available.

Knapsack without Repetition

For every $w \leq W$ and $0 \leq j \leq n$, let

$$K(w, j) = \text{ maximum value achievable with a knapsack of capacity } w \text{ and items } 1, \ldots, j$$

Knapsack without Repetition

For every $w \leq W$ and $0 \leq j \leq n$, let

$$K(w, j) = \text{ maximum value achievable with a knapsack of capacity } w \text{ and items } 1, \ldots, j$$

We express this in terms of smaller subproblems:

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

Knapsack without Repetition

Initialize all $K(0, j) = 0$ and all $K(w, 0) = 0$;
for $j = 1$ *to* n **do**
  for $w = 1$ *to* n **do**
    if $w_j > w$ **then** $K(w, j) = K(w, j-1)$;
    $K(w, j) = \max\{K(w - w_j, j-1) + v_j, K(w, j-1)\}$;
  end
end
`return(`$K(W, n)$`);`

Knapsack without Repetition

Initialize all $K(0, j) = 0$ and all $K(w, 0) = 0$;
for $j = 1$ *to* n **do**
  for $w = 1$ *to* n **do**
    if $w_j > w$ **then** $K(w, j) = K(w, j-1)$;
    $K(w, j) = \max\{K(w - w_j, j-1) + v_j, K(w, j-1)\}$;
  end
end
`return` $(K(W, n))$;

The over running time is $O(n \cdot W)$.

Knapsack without Repetition

Take $W = 9$, and

Item	Weight	Value
1	2	\$3
2	3	\$4
3	4	\$5
4	5	\$7

and there is only one of each item available.

Problem. Given two strings $x_1 x_2 \dots x_m$ and $y_1 y_2 \dots y_n$, find a common subsequence that is as long as possible.

Longest Common Subsequence

Problem. Given two strings $x_1 x_2 \dots x_m$ and $y_1 y_2 \dots y_n$, find a common subsequence that is as long as possible.

Alternative viewpoint. Delete some characters from x; delete some character from y; a common subsequence if it results in the same string.

Longest Common Subsequence

Problem. Given two strings $x_1 x_2 \dots x_m$ and $y_1 y_2 \dots y_n$, find a common subsequence that is as long as possible.

Alternative viewpoint. Delete some characters from x; delete some character from y; a common subsequence if it results in the same string.

Example. **LCS(GGCACCACG, ACGGCGGATACG) = GGCAACG**.

Quiz

How about the longest common string?

Chain Matrix Multiplication

The Problem

Suppose that we want to multiply four matrices, A, B, C, D, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively.

The Problem

Suppose that we want to multiply four matrices, A, B, C, D, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively.

Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes $m \cdot n \cdot p$ multiplications.

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

The Problem

Suppose that we want to multiply four matrices, A, B, C, D, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively.

Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes $m \cdot n \cdot p$ multiplications.

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

Q: How do we determine the optimal order, if we want to compute $A_1 \times A_2 \times \ldots \times A_n$, where the A_i's are matrices with dimensions $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$, respectively?

Binary Tree

A particular parenthesization can be represented by a binary tree in which

- the individual matrices correspond to the leaves,
- the root is the final product, and
- interior nodes are intermediate products.

Binary Tree

A particular parenthesization can be represented by a binary tree in which

- the individual matrices correspond to the leaves,
- the root is the final product, and
- interior nodes are intermediate products.

The possible orders in which to do the multiplication correspond to the various full binary trees with n leaves.

Subproblems

For $1 \leq i \leq j \leq n$, let

$$C(i,j) = \text{ minimum cost of multiplying } A_i \times A_{i+1} \times \ldots \times A_j$$

Subproblems

For $1 \leq i \leq j \leq n$, let

$$C(i,j) = \text{ minimum cost of multiplying } A_i \times A_{i+1} \times \ldots \times A_j$$

$$C(i,j) = \min_{i \leq k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

The Program

The Program


```
for i = 1 to n do
    C(i, i) = 0;
end
for s = 1 to n − 1 do
    for i = 1 to n − s do
        j = i + s;
        C(i, j) = min_{i≤k<j}{C(i, k) + C(k + 1, j) + m_{i−1} · m_k · m_j};
    end
end
return (C(1, n));
```

The Program

for $i = 1$ *to* n **do**
  $C(i, i) = 0$;
end
for $s = 1$ *to* $n - 1$ **do**
  for $i = 1$ *to* $n - s$ **do**
    $j = i + s$;
    $C(i, j) = \min_{i \le k < j}\{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j\}$;
  end
end
`return` $(C(1, n))$;

The over running time is $O(n^3)$.

The Example

Suppose that we want to multiply four matrices, A, B, C, D, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively.