Algorithms: COMP3121/3821/9101/9801

Aleks Ignjatović

School of Computer Science and Engineering
University of New South Wales

TOPIC 5: DYNAMIC PROGRAMMING

Dynamic Programming

- **The main idea of Dynamic Programming:** build an optimal solution to the problem from optimal solutions for (carefully chosen) subproblems;

1.how to generate subproblem
2.how to order them

Dynamic Programming

- **The main idea of Dynamic Programming:** build an optimal solution to the problem from optimal solutions for (carefully chosen) subproblems;
- Subproblems are chosen in a way which allows recursive construction of optimal solutions to problems from optimal solutions to smaller size problems.

Dynamic Programming

- **The main idea of Dynamic Programming:** build an optimal solution to the problem from optimal solutions for (carefully chosen) subproblems;
- Subproblems are chosen in a way which allows recursive construction of optimal solutions to problems from optimal solutions to smaller size problems.
- Efficiency of DP comes from the fact that that the sets of subproblems needed to solve larger problems heavily overlap; each subproblem is solved only once and its solution is stored in a table for multiple use for solving many larger problems.

Dynamic Programming: Activity Selection

- **Instance:** A list of activities a_i, $1 \leq i \leq n$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Dynamic Programming: Activity Selection

- **Instance:** A list of activities a_i, $1 \leq i \leq n$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a subset of compatible activities of **maximal total duration**.

Dynamic Programming: Activity Selection

- **Instance:** A list of activities a_i, $1 \leq i \leq n$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

- **Task:** Find a subset of compatible activities of **maximal total duration**.

- Remember, we used the Greedy Method to solve a somewhat similar problem of finding a subset with the **largest possible number** of compatible activities, but the Greedy Method **does not** work for the new problem.

Dynamic Programming: Activity Selection

- **Instance:** A list of activities a_i, $1 \leq i \leq n$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a subset of compatible activities of **maximal total duration**.
- Remember, we used the Greedy Method to solve a somewhat similar problem of finding a subset with the **largest possible number** of compatible activities, but the Greedy Method **does not** work for the new problem.
- We start by sorting these activities by their finishing time into a non-decreasing sequence, so will assume that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Dynamic Programming: Activity Selection

- **Instance:** A list of activities a_i, $1 \leq i \leq n$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a subset of compatible activities of **maximal total duration**.
- Remember, we used the Greedy Method to solve a somewhat similar problem of finding a subset with the **largest possible number** of compatible activities, but the Greedy Method **does not** work for the new problem.
- We start by sorting these activities by their finishing time into a non-decreasing sequence, so will assume that $f_1 \leq f_2 \leq \ldots \leq f_n$.
- For every $i \leq n$ we solve the following subproblems:
  Subproblem $P(i)$: find a subsequence σ_i of the sequence of activities $S_i = \langle a_1, a_2, \ldots, a_i \rangle$ such that:

Dynamic Programming: Activity Selection

- **Instance:** A list of activities a_i, $1 \leq i \leq n$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a subset of compatible activities of **maximal total duration**.
- Remember, we used the Greedy Method to solve a somewhat similar problem of finding a subset with the **largest possible number** of compatible activities, but the Greedy Method **does not** work for the new problem.
- We start by sorting these activities by their finishing time into a non-decreasing sequence, so will assume that $f_1 \leq f_2 \leq \ldots \leq f_n$.
- For every $i \leq n$ we solve the following subproblems:
  Subproblem $P(i)$: find a subsequence σ_i of the sequence of activities $S_i = \langle a_1, a_2, \ldots, a_i \rangle$ such that:
  1. σ_i consists of non-overlapping activities;

Dynamic Programming: Activity Selection

- **Instance:** A list of activities a_i, $1 \leq i \leq n$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

- **Task:** Find a subset of compatible activities of **maximal total duration**.

- Remember, we used the Greedy Method to solve a somewhat similar problem of finding a subset with the **largest possible number** of compatible activities, but the Greedy Method **does not** work for the new problem.

- We start by sorting these activities by their finishing time into a non-decreasing sequence, so will assume that $f_1 \leq f_2 \leq \ldots \leq f_n$.

- For every $i \leq n$ we solve the following subproblems:
  Subproblem $P(i)$: find a subsequence σ_i of the sequence of activities $S_i = \langle a_1, a_2, \ldots, a_i \rangle$ such that:
  1. σ_i consists of non-overlapping activities;
  2. σ_i ends with activity a_i;

Dynamic Programming: Activity Selection

- **Instance:** A list of activities a_i, $1 \leq i \leq n$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

- **Task:** Find a subset of compatible activities of **maximal total duration**.

- Remember, we used the Greedy Method to solve a somewhat similar problem of finding a subset with the **largest possible number** of compatible activities, but the Greedy Method **does not** work for the new problem.

- We start by sorting these activities by their finishing time into a non-decreasing sequence, so will assume that $f_1 \leq f_2 \leq \ldots \leq f_n$.

- For every $i \leq n$ we solve the following subproblems:
  Subproblem $P(i)$: find a subsequence σ_i of the sequence of activities $S_i = \langle a_1, a_2, \ldots, a_i \rangle$ such that:
  1. σ_i consists of non-overlapping activities;
  2. σ_i ends with activity a_i;
  3. σ_i is of maximal total duration among all subsequences of S_i which satisfy 1 and 2.

Dynamic Programming: Activity Selection

- **Instance:** A list of activities a_i, $1 \leq i \leq n$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

- **Task:** Find a subset of compatible activities of **maximal total duration**.

- Remember, we used the Greedy Method to solve a somewhat similar problem of finding a subset with the **largest possible number** of compatible activities, but the Greedy Method **does not** work for the new problem.

- We start by sorting these activities by their finishing time into a non-decreasing sequence, so will assume that $f_1 \leq f_2 \leq \ldots \leq f_n$.

- For every $i \leq n$ we solve the following subproblems:
  Subproblem $P(i)$: find a subsequence σ_i of the sequence of activities $S_i = \langle a_1, a_2, \ldots, a_i \rangle$ such that:
  1. σ_i consists of non-overlapping activities;
  2. σ_i ends with activity a_i;
  3. σ_i is of maximal total duration among all subsequences of S_i which satisfy 1 and 2.

- Note: the role of condition 2 is to simplify recursion.

- Let $T(i)$ be the total duration of the optimal solution $S(i)$ of the subproblem $P(i)$.

Dynamic Programming: Activity Selection

- Let $T(i)$ be the total duration of the optimal solution $S(i)$ of the subproblem $P(i)$.
- For $S(1)$ we choose a_1; thus $T(1) = f_1 - s_1$;

Dynamic Programming: Activity Selection

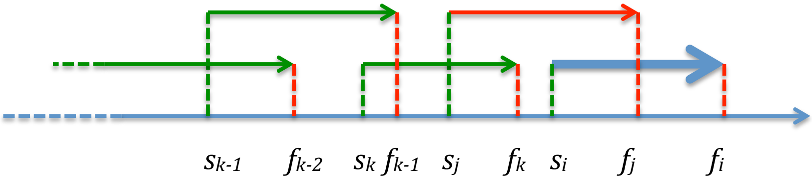
- Let $T(i)$ be the total duration of the optimal solution $S(i)$ of the subproblem $P(i)$.
- For $S(1)$ we choose a_1; thus $T(1) = f_1 - s_1$;
- **Recursion:** assuming that we have solved subproblems for all $j < i$ and stored them in a table, we let

$$T(i) = \max\{T(j) + f_i - s_i \ : \ j < i \ \ \& \ \ f_j < s_i\}$$

Dynamic Programming: Activity Selection

- Let $T(i)$ be the total duration of the optimal solution $S(i)$ of the subproblem $P(i)$.
- For $S(1)$ we choose a_1; thus $T(1) = f_1 - s_1$;
- **Recursion:** assuming that we have solved subproblems for all $j < i$ and stored them in a table, we let

$$T(i) = \max\{T(j) + f_i - s_i \ : \ j < i \ \ \& \ \ f_j < s_i\}$$



- In the table, besides $T(i)$, we also store j for which the above max is achieved.

Dynamic Programming: Activity Selection

- Why does such a recursion produce optimal solutions to subproblems $P(i)$?

Dynamic Programming: Activity Selection

- Why does such a recursion produce optimal solutions to subproblems $P(i)$?
- Let the optimal solution of subproblem $P(i)$ be the sequence $S = (a_{k_1}, a_{k_2}, \ldots, a_{k_{m-1}}, a_{k_m})$ where $k_m = i$;

Dynamic Programming: Activity Selection

- Why does such a recursion produce optimal solutions to subproblems $P(i)$?
- Let the optimal solution of subproblem $P(i)$ be the sequence $S = (a_{k_1}, a_{k_2}, \ldots, a_{k_{m-1}}, a_{k_m})$ where $k_m = i$;
- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \ldots, a_{k_{m-1}})$ is an optimal solution to subproblem $P(k_{m-1})$, where $k_{m-1} < i$.

Dynamic Programming: Activity Selection

- Why does such a recursion produce optimal solutions to subproblems $P(i)$?
- Let the optimal solution of subproblem $P(i)$ be the sequence $S = (a_{k_1}, a_{k_2}, \ldots, a_{k_{m-1}}, a_{k_m})$ where $k_m = i$;
- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \ldots, a_{k_{m-1}})$ is an optimal solution to subproblem $P(k_{m-1})$, where $k_{m-1} < i$.
- Why?

Dynamic Programming: Activity Selection

- Why does such a recursion produce optimal solutions to subproblems $P(i)$?
- Let the optimal solution of subproblem $P(i)$ be the sequence $S = (a_{k_1}, a_{k_2}, \ldots, a_{k_{m-1}}, a_{k_m})$ where $k_m = i$;
- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \ldots, a_{k_{m-1}})$ is an optimal solution to subproblem $P(k_{m-1})$, where $k_{m-1} < i$.
- Why? We apply the same "cut and paste" argument which we used to prove the optimality of the greedy solutions!

Dynamic Programming: Activity Selection

- Why does such a recursion produce optimal solutions to subproblems $P(i)$?
- Let the optimal solution of subproblem $P(i)$ be the sequence $S = (a_{k_1}, a_{k_2}, \ldots, a_{k_{m-1}}, a_{k_m})$ where $k_m = i$;
- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \ldots, a_{k_{m-1}})$ is an optimal solution to subproblem $P(k_{m-1})$, where $k_{m-1} < i$.
- Why? We apply the same "cut and paste" argument which we used to prove the optimality of the greedy solutions!
- If there were a sequence S^* of a larger total duration than the duration of sequence S' and also ending with activity $a_{k_{m-1}}$, we could obtain a sequence $\hat{S}$ by extending the sequence S^* with activity a_{k_m} and obtain a solution for subproblem $P(i)$ with a longer total duration than the total duration of sequence S, contradicting the optimality of S.

Dynamic Programming: Activity Selection

- Continuing with the solution of the problem, we now let

$$T_{max} = \max\{T(i) \ : \ i \leq n\}.$$

Dynamic Programming: Activity Selection

- Continuing with the solution of the problem, we now let

$$T_{max} = \max\{T(i) \ : \ i \leq n\}.$$

- We can now reconstruct the optimal sequence which solves our problem from the table of partial solutions, because in the i^{th} slot of the table, besides $T(i)$, we also store j such that the optimal solution of $P(i)$ extends the optimal solution of subproblem $P(j)$.

Dynamic Programming: Activity Selection

- Continuing with the solution of the problem, we now let

$$T_{max} = \max\{T(i) \;:\; i \leq n\}.$$

- We can now reconstruct the optimal sequence which solves our problem from the table of partial solutions, because in the i^{th} slot of the table, besides $T(i)$, we also store j such that the optimal solution of $P(i)$ extends the optimal solution of subproblem $P(j)$.

- Why is such solution optimal?

Dynamic Programming: Activity Selection

- Continuing with the solution of the problem, we now let

$$T_{max} = \max\{T(i) \ : \ i \leq n\}.$$

- We can now reconstruct the optimal sequence which solves our problem from the table of partial solutions, because in the i^{th} slot of the table, besides $T(i)$, we also store j such that the optimal solution of $P(i)$ extends the optimal solution of subproblem $P(j)$.
- Why is such solution optimal?
- If such optimal solution ends with activity a_k, it would have been obtained as the optimal solution of problem $P(k)$.

Dynamic Programming: Activity Selection

- Continuing with the solution of the problem, we now let

$$T_{max} = \max\{T(i) \ : \ i \leq n\}.$$

- We can now reconstruct the optimal sequence which solves our problem from the table of partial solutions, because in the i^{th} slot of the table, besides $T(i)$, we also store j such that the optimal solution of $P(i)$ extends the optimal solution of subproblem $P(j)$.

- Why is such solution optimal?

- If such optimal solution ends with activity a_k, it would have been obtained as the optimal solution of problem $P(k)$.

- Time complexity: having sorted the activities by their finishing times in time $O(n \log n)$, we need to examine n intervals in the role of the last activity in an optimal sub-sequence and for each such interval we have to find all preceding compatible intervals and their optimal solutions (to be looked up in a table). Thus, the time complexity is $O(n^2)$.

More Dynamic Programming Problems

- **Longest Increasing Subsequence:** Given a sequence of n real numbers $A[1..n]$, determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence are strictly increasing.

More Dynamic Programming Problems

- **Longest Increasing Subsequence:** Given a sequence of n real numbers $A[1..n]$, determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence are strictly increasing.

- Solution: For each $i \leq n$ we solve the following subproblems:

More Dynamic Programming Problems

- **Longest Increasing Subsequence:** Given a sequence of n real numbers $A[1..n]$, determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence are strictly increasing.

- Solution: For each $i \leq n$ we solve the following subproblems:

- ***Subproblem*** $P(i)$***:*** *Find a subsequence of the sequence* $A[1..i]$ *of maximum length in which the values are strictly increasing and which ends with* $A[i]$.

More Dynamic Programming Problems

- **Longest Increasing Subsequence:** Given a sequence of n real numbers $A[1..n]$, determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence are strictly increasing.

- Solution: For each $i \leq n$ we solve the following subproblems:

- ***Subproblem*** $P(i)$***:*** *Find a subsequence of the sequence* $A[1..i]$ *of maximum length in which the values are strictly increasing and which ends with* $A[i]$.

- Recursion: Assume we have solved all the subproblems for $j < i$;

- **Longest Increasing Subsequence:** Given a sequence of n real numbers $A[1..n]$, determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence are strictly increasing.

- Solution: For each $i \leq n$ we solve the following subproblems:

- ***Subproblem*** $P(i)$***:*** *Find a subsequence of the sequence* $A[1..i]$ *of maximum length in which the values are strictly increasing and which ends with* $A[i]$*.*

- Recursion: Assume we have solved all the subproblems for $j < i$;

- We now look for all $A[m]$ such that $m < i$ and such that $A[m] < A[i]$;

More Dynamic Programming Problems

- **Longest Increasing Subsequence:** Given a sequence of n real numbers $A[1..n]$, determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence are strictly increasing.

- Solution: For each $i \leq n$ we solve the following subproblems:

- ***Subproblem*** $P(i)$***:*** *Find a subsequence of the sequence* $A[1..i]$ *of maximum length in which the values are strictly increasing and which ends with* $A[i]$.

- Recursion: Assume we have solved all the subproblems for $j < i$;

- We now look for all $A[m]$ such that $m < i$ and such that $A[m] < A[i]$;

- Among those we pick m which produced the longest increasing subsequence ending with $A[m]$ and extend it with $A[i]$ to obtain the longest increasing subsequence which ends with $A[i]$;

More Dynamic Programming Problems

- **Longest Increasing Subsequence:** Given a sequence of n real numbers $A[1..n]$, determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence are strictly increasing.
- Solution: For each $i \leq n$ we solve the following subproblems:
- ***Subproblem*** $P(i)$***:*** *Find a subsequence of the sequence* $A[1..i]$ *of maximum length in which the values are strictly increasing and which ends with* $A[i]$.
- Recursion: Assume we have solved all the subproblems for $j < i$;
- We now look for all $A[m]$ such that $m < i$ and such that $A[m] < A[i]$;
- Among those we pick m which produced the longest increasing subsequence ending with $A[m]$ and extend it with $A[i]$ to obtain the longest increasing subsequence which ends with $A[i]$;
- Why does this produce optimal solutions to subproblems?

More Dynamic Programming Problems

- **Longest Increasing Subsequence:** Given a sequence of n real numbers $A[1..n]$, determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence are strictly increasing.

- Solution: For each $i \leq n$ we solve the following subproblems:

- ***Subproblem*** $P(i)$***:*** *Find a subsequence of the sequence* $A[1..i]$ *of maximum length in which the values are strictly increasing and which ends with* $A[i]$.

- Recursion: Assume we have solved all the subproblems for $j < i$;

- We now look for all $A[m]$ such that $m < i$ and such that $A[m] < A[i]$;

- Among those we pick m which produced the longest increasing subsequence ending with $A[m]$ and extend it with $A[i]$ to obtain the longest increasing subsequence which ends with $A[i]$;

- Why does this produce optimal solutions to subproblems? The same "cut and paste argument:

More Dynamic Programming Problems

- Truncating optimal solution for $P(i)$ must produce optimal solution of the subproblem $P(m)$, otherwise we could find a better solution for $P(i)$ as well, by extending such better solution of $P(m)$.

- Truncating optimal solution for $P(i)$ must produce optimal solution of the subproblem $P(m)$, otherwise we could find a better solution for $P(i)$ as well, by extending such better solution of $P(m)$.

- We store in the i^{th} slot of the table the length of the longest increasing subsequence ending with $A[i]$ and m such that the optimal solution for $P(i)$ extends the optimal solution for $P(m)$.

More Dynamic Programming Problems

- Truncating optimal solution for $P(i)$ must produce optimal solution of the subproblem $P(m)$, otherwise we could find a better solution for $P(i)$ as well, by extending such better solution of $P(m)$.

- We store in the i^{th} slot of the table the length of the longest increasing subsequence ending with $A[i]$ and m such that the optimal solution for $P(i)$ extends the optimal solution for $P(m)$.

- So, we have found for every $i \leq n$ the longest increasing subsequence of the sequence $A[1..i]$ which ends with $A[i]$.

More Dynamic Programming Problems

- Truncating optimal solution for $P(i)$ must produce optimal solution of the subproblem $P(m)$, otherwise we could find a better solution for $P(i)$ as well, by extending such better solution of $P(m)$.
- We store in the i^{th} slot of the table the length of the longest increasing subsequence ending with $A[i]$ and m such that the optimal solution for $P(i)$ extends the optimal solution for $P(m)$.
- So, we have found for every $i \leq n$ the longest increasing subsequence of the sequence $A[1..i]$ which ends with $A[i]$.
- Finally, from all such subsequences we pick the longest one.

More Dynamic Programming Problems

- Truncating optimal solution for $P(i)$ must produce optimal solution of the subproblem $P(m)$, otherwise we could find a better solution for $P(i)$ as well, by extending such better solution of $P(m)$.
- We store in the i^{th} slot of the table the length of the longest increasing subsequence ending with $A[i]$ and m such that the optimal solution for $P(i)$ extends the optimal solution for $P(m)$.
- So, we have found for every $i \leq n$ the longest increasing subsequence of the sequence $A[1..i]$ which ends with $A[i]$.
- Finally, from all such subsequences we pick the longest one.
- Again, the condition for the sequence to end with $A[i]$ is not restrictive because if the optimal solution ends with some $A[m]$, it would have been constructed as the solution for $P(m)$.

More Dynamic Programming Problems

- Truncating optimal solution for $P(i)$ must produce optimal solution of the subproblem $P(m)$, otherwise we could find a better solution for $P(i)$ as well, by extending such better solution of $P(m)$.
- We store in the i^{th} slot of the table the length of the longest increasing subsequence ending with $A[i]$ and m such that the optimal solution for $P(i)$ extends the optimal solution for $P(m)$.
- So, we have found for every $i \leq n$ the longest increasing subsequence of the sequence $A[1..i]$ which ends with $A[i]$.
- Finally, from all such subsequences we pick the longest one.
- Again, the condition for the sequence to end with $A[i]$ is not restrictive because if the optimal solution ends with some $A[m]$, it would have been constructed as the solution for $P(m)$.
- Time complexity: $O(n^2)$.

More Dynamic Programming Problems

- Truncating optimal solution for $P(i)$ must produce optimal solution of the subproblem $P(m)$, otherwise we could find a better solution for $P(i)$ as well, by extending such better solution of $P(m)$.
- We store in the i^{th} slot of the table the length of the longest increasing subsequence ending with $A[i]$ and m such that the optimal solution for $P(i)$ extends the optimal solution for $P(m)$.
- So, we have found for every $i \leq n$ the longest increasing subsequence of the sequence $A[1..i]$ which ends with $A[i]$.
- Finally, from all such subsequences we pick the longest one.
- Again, the condition for the sequence to end with $A[i]$ is not restrictive because if the optimal solution ends with some $A[m]$, it would have been constructed as the solution for $P(m)$.
- Time complexity: $O(n^2)$.
- Exercise: (somewhat tough, but very useful) Design an algorithm for solving this problem which runs in time $n \log n$.

- **Making Change.** You are given n types of coin denominations of values $v(1) < v(2) < ... < v(n)$ (all integers). Assume $v(1) = 1$, so that you can always make change for any integer amount. Give an algorithm which makes change for any given integer amount C with as few coins as possible, assuming that you have an unlimited supply of coins of each denomination.

More Dynamic Programming Problems

- **Making Change.** You are given n types of coin denominations of values $v(1) < v(2) < ... < v(n)$ (all integers). Assume $v(1) = 1$, so that you can always make change for any integer amount. Give an algorithm which makes change for any given integer amount C with as few coins as possible, assuming that you have an unlimited supply of coins of each denomination.
- Solution: DP Recursion on the amount C. We will fill a table containing C many slots, so that an optimal solution for an amount i is stored in slot i.

More Dynamic Programming Problems

- **Making Change.** You are given n types of coin denominations of values $v(1) < v(2) < ... < v(n)$ (all integers). Assume $v(1) = 1$, so that you can always make change for any integer amount. Give an algorithm which makes change for any given integer amount C with as few coins as possible, assuming that you have an unlimited supply of coins of each denomination.

- Solution: DP Recursion on the amount C. We will fill a table containing C many slots, so that an optimal solution for an amount i is stored in slot i.

- If $C = 1$ the solution is trivial: just use one coin of denomination $v(1) = 1$;

More Dynamic Programming Problems

- **Making Change.** You are given n types of coin denominations of values $v(1) < v(2) < ... < v(n)$ (all integers). Assume $v(1) = 1$, so that you can always make change for any integer amount. Give an algorithm which makes change for any given integer amount C with as few coins as possible, assuming that you have an unlimited supply of coins of each denomination.

- Solution: DP Recursion on the amount C. We will fill a table containing C many slots, so that an optimal solution for an amount i is stored in slot i.

- If $C = 1$ the solution is trivial: just use one coin of denomination $v(1) = 1$;

- Assume we have found optimal solutions for every amount $j < i$ and now want to find an optimal solution for amount i.

More Dynamic Programming Problems

- We consider optimal solutions $opt(i - v(k))$ for every amount of the form $i - v(k)$, where k ranges from 1 to n. (Recall $v(1), \dots, v(n)$ are all of the available denominations.)

More Dynamic Programming Problems

- We consider optimal solutions $opt(i - v(k))$ for every amount of the form $i - v(k)$, where k ranges from 1 to n. (Recall $v(1), \ldots, v(n)$ are all of the available denominations.)

- Among all of these optimal solutions (which we find in the table we are constructing recursively!) we pick one which uses the fewest number of coins, say this is $opt(i - v(m))$ for some m, $1 \leq m \leq n$.

More Dynamic Programming Problems

- We consider optimal solutions $opt(i - v(k))$ for every amount of the form $i - v(k)$, where k ranges from 1 to n. (Recall $v(1), \ldots, v(n)$ are all of the available denominations.)

- Among all of these optimal solutions (which we find in the table we are constructing recursively!) we pick one which uses the fewest number of coins, say this is $opt(i - v(m))$ for some m, $1 \leq m \leq n$.

- We obtain an optimal solution $opt(i)$ for amount i by adding to $opt(i - v(m))$ one coin of denomination $v(m)$.

More Dynamic Programming Problems

- We consider optimal solutions $opt(i - v(k))$ for every amount of the form $i - v(k)$, where k ranges from 1 to n. (Recall $v(1), \ldots, v(n)$ are all of the available denominations.)

- Among all of these optimal solutions (which we find in the table we are constructing recursively!) we pick one which uses the fewest number of coins, say this is $opt(i - v(m))$ for some m, $1 \leq m \leq n$.

- We obtain an optimal solution $opt(i)$ for amount i by adding to $opt(i - v(m))$ one coin of denomination $v(m)$.

- Why does this produce an optimal solution for amount $i \leq C$?

More Dynamic Programming Problems

- We consider optimal solutions $opt(i - v(k))$ for every amount of the form $i - v(k)$, where k ranges from 1 to n. (Recall $v(1), \ldots, v(n)$ are all of the available denominations.)
- Among all of these optimal solutions (which we find in the table we are constructing recursively!) we pick one which uses the fewest number of coins, say this is $opt(i - v(m))$ for some m, $1 \leq m \leq n$.
- We obtain an optimal solution $opt(i)$ for amount i by adding to $opt(i - v(m))$ one coin of denomination $v(m)$.
- Why does this produce an optimal solution for amount $i \leq C$?
- Consider an optimal solution for amount $i \leq C$; and say such solution includes at least one coin of denomination $v(m)$ for some $1 \leq m \leq n$.

More Dynamic Programming Problems

- We consider optimal solutions $opt(i - v(k))$ for every amount of the form $i - v(k)$, where k ranges from 1 to n. (Recall $v(1), \ldots, v(n)$ are all of the available denominations.)

- Among all of these optimal solutions (which we find in the table we are constructing recursively!) we pick one which uses the fewest number of coins, say this is $opt(i - v(m))$ for some m, $1 \leq m \leq n$.

- We obtain an optimal solution $opt(i)$ for amount i by adding to $opt(i - v(m))$ one coin of denomination $v(m)$.

- Why does this produce an optimal solution for amount $i \leq C$?

- Consider an optimal solution for amount $i \leq C$; and say such solution includes at least one coin of denomination $v(m)$ for some $1 \leq m \leq n$. But then removing such a coin must produce an optimal solution for the amount $i - v(m)$ again by our cut-and-paste argument.

More Dynamic Programming Problems

- We consider optimal solutions $opt(i - v(k))$ for every amount of the form $i - v(k)$, where k ranges from 1 to n. (Recall $v(1), \dots, v(n)$ are all of the available denominations.)

- Among all of these optimal solutions (which we find in the table we are constructing recursively!) we pick one which uses the fewest number of coins, say this is $opt(i - v(m))$ for some m, $1 \leq m \leq n$.

- We obtain an optimal solution $opt(i)$ for amount i by adding to $opt(i - v(m))$ one coin of denomination $v(m)$.

- Why does this produce an optimal solution for amount $i \leq C$?

- Consider an optimal solution for amount $i \leq C$; and say such solution includes at least one coin of denomination $v(m)$ for some $1 \leq m \leq n$. But then removing such a coin must produce an optimal solution for the amount $i - v(m)$ again by our cut-and-paste argument.

- However, we do not know which coins the optimal solution includes, so we try all the available coins and then pick m for which the optimal solution for amount $i - v(m)$ uses the fewest number of coins.

- It is enough to store in the i^{th} slot of the table such m and $opt(i)$ because this allows us to reconstruct the optimal solution by looking at m_1 stored in the i^{th} slot, then look at m_2 stored in the slot $i - v(m_1)$, then look at m_2 stored in the slot $i - v(m_1) - v(m_2)$, etc.

- It is enough to store in the i^{th} slot of the table such m and $opt(i)$ because this allows us to reconstruct the optimal solution by looking at m_1 stored in the i^{th} slot, then look at m_2 stored in the slot $i - v(m_1)$, then look at m_2 stored in the slot $i - v(m_1) - v(m_2)$, etc.

- Finally, after C many such steps we obtain $opt(C)$ which is the solution we need.

More Dynamic Programming Problems

- It is enough to store in the i^{th} slot of the table such m and $opt(i)$ because this allows us to reconstruct the optimal solution by looking at m_1 stored in the i^{th} slot, then look at m_2 stored in the slot $i - v(m_1)$, then look at m_2 stored in the slot $i - v(m_1) - v(m_2)$, etc.

- Finally, after C many such steps we obtain $opt(C)$ which is the solution we need.

- Time complexity of our algorithm is nC.

More Dynamic Programming Problems

- It is enough to store in the i^{th} slot of the table such m and $opt(i)$ because this allows us to reconstruct the optimal solution by looking at m_1 stored in the i^{th} slot, then look at m_2 stored in the slot $i - v(m_1)$, then look at m_2 stored in the slot $i - v(m_1) - v(m_2)$, etc.
- Finally, after C many such steps we obtain $opt(C)$ which is the solution we need.
- Time complexity of our algorithm is nC.
- **Note:** Our algorithm is **NOT** a polynomial time algorithm in the **length** of the input, because the length of a representation of C is only $\log C$, while the running time is nC.

- It is enough to store in the i^{th} slot of the table such m and $opt(i)$ because this allows us to reconstruct the optimal solution by looking at m_1 stored in the i^{th} slot, then look at m_2 stored in the slot $i - v(m_1)$, then look at m_2 stored in the slot $i - v(m_1) - v(m_2)$, etc.
- Finally, after C many such steps we obtain $opt(C)$ which is the solution we need.
- Time complexity of our algorithm is nC.
- **Note:** Our algorithm is **NOT** a polynomial time algorithm in the **length** of the input, because the length of a representation of C is only $\log C$, while the running time is nC.
- But this is the best what we can do...

More Dynamic Programming Problems

Integer Knapsack Problem (Duplicate Items Allowed) You have n types of items; all items of kind i are identical and of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible. You can take any number of items of each kind.

More Dynamic Programming Problems

Integer Knapsack Problem (Duplicate Items Allowed) You have n types of items; all items of kind i are identical and of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible. You can take any number of items of each kind.

- Solution: DP recursion on the capacity C of the knapsack.

Integer Knapsack Problem (Duplicate Items Allowed) You have n types of items; all items of kind i are identical and of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible. You can take any number of items of each kind.

- Solution: DP recursion on the capacity C of the knapsack.
- We build a table of optimal solutions for all knapsacks of capacities $i \leq C$.

More Dynamic Programming Problems

Integer Knapsack Problem (Duplicate Items Allowed) You have n types of items; all items of kind i are identical and of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible. You can take any number of items of each kind.

- Solution: DP recursion on the capacity C of the knapsack.
- We build a table of optimal solutions for all knapsacks of capacities $i \leq C$.
- Assume we have solved the problem for all knapsacks of capacities $j < i$.

More Dynamic Programming Problems

Integer Knapsack Problem (Duplicate Items Allowed) You have n types of items; all items of kind i are identical and of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible. You can take any number of items of each kind.

- Solution: DP recursion on the capacity C of the knapsack.
- We build a table of optimal solutions for all knapsacks of capacities $i \leq C$.
- Assume we have solved the problem for all knapsacks of capacities $j < i$.
- We now look at optimal solutions $opt(i - w_m)$ for all knapsacks of capacities $i - w_m$ for all $1 \leq m \leq n$;

More Dynamic Programming Problems

Integer Knapsack Problem (Duplicate Items Allowed) You have n types of items; all items of kind i are identical and of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible. You can take any number of items of each kind.

- Solution: DP recursion on the capacity C of the knapsack.
- We build a table of optimal solutions for all knapsacks of capacities $i \leq C$.
- Assume we have solved the problem for all knapsacks of capacities $j < i$.
- We now look at optimal solutions $opt(i - w_m)$ for all knapsacks of capacities $i - w_m$ for all $1 \leq m \leq n$;
- Chose the one for which $opt(i - w_m) + v_m$ is the largest;

More Dynamic Programming Problems

Integer Knapsack Problem (Duplicate Items Allowed) You have n types of items; all items of kind i are identical and of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible. You can take any number of items of each kind.

- Solution: DP recursion on the capacity C of the knapsack.
- We build a table of optimal solutions for all knapsacks of capacities $i \leq C$.
- Assume we have solved the problem for all knapsacks of capacities $j < i$.
- We now look at optimal solutions $opt(i - w_m)$ for all knapsacks of capacities $i - w_m$ for all $1 \leq m \leq n$;
- Chose the one for which $opt(i - w_m) + v_m$ is the largest;
- Add to such optimal solution for the knapsack of size $i - w_m$ item m to obtain a packing of a knapsack of size i of the highest possible value.

More Dynamic Programming Problems

- Thus, $opt(i) = \max\{opt(i - w_m) + v_m : 1 \leq m \leq n\}$.

- Thus, $opt(i) = \max\{opt(i - w_m) + v_m : 1 \leq m \leq n\}$.
- After C many steps we obtain $opt(C)$ which is what we need.

- Thus, $opt(i) = \max\{opt(i - w_m) + v_m : 1 \leq m \leq n\}$.
- After C many steps we obtain $opt(C)$ which is what we need.
- Again, our algorithm is **NOT** polynomial in the **length** of the input.

- **Integer Knapsack Problem (Duplicate Items NOT Allowed)** You have n items (some of which can be identical); item I_i is of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

More Dynamic Programming Problems

- **Integer Knapsack Problem (Duplicate Items NOT Allowed)** You have n items (some of which can be identical); item I_i is of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

- This is an example of a "2D" recursion; we will be filling a table of size $n \times C$, row by row; subproblems $P(i, c)$ for all $i \leq n$ and $c \leq C$ will be of the form:

  chose from items $I_1, I_2, \ldots, I_i$ a subset which fits in a knapsack of capacity c and is of the largest possible total value.

More Dynamic Programming Problems

- **Integer Knapsack Problem (Duplicate Items NOT Allowed)** You have n items (some of which can be identical); item I_i is of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

- This is an example of a "2D" recursion; we will be filling a table of size $n \times C$, row by row; subproblems $P(i, c)$ for all $i \leq n$ and $c \leq C$ will be of the form:

  chose from items $I_1, I_2, \ldots, I_i$ a subset which fits in a knapsack of capacity c and is of the largest possible total value.

- Fix now $i \leq n$ and $c \leq C$ and assume we have solved the subproblems for:

More Dynamic Programming Problems

- **Integer Knapsack Problem (Duplicate Items NOT Allowed)** You have n items (some of which can be identical); item I_i is of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

- This is an example of a "2D" recursion; we will be filling a table of size $n \times C$, row by row; subproblems $P(i, c)$ for all $i \leq n$ and $c \leq C$ will be of the form:

  chose from items $I_1, I_2, \ldots, I_i$ a subset which fits in a knapsack of capacity c and is of the largest possible total value.

- Fix now $i \leq n$ and $c \leq C$ and assume we have solved the subproblems for:
  1. all $j < i$ and all knapsacks of capacities from 1 to C;

More Dynamic Programming Problems

- **Integer Knapsack Problem (Duplicate Items NOT Allowed)** You have n items (some of which can be identical); item I_i is of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

- This is an example of a "2D" recursion; we will be filling a table of size $n \times C$, row by row; subproblems $P(i, c)$ for all $i \leq n$ and $c \leq C$ will be of the form:

  chose from items $I_1, I_2, \ldots, I_i$ a subset which fits in a knapsack of capacity c and is of the largest possible total value.

- Fix now $i \leq n$ and $c \leq C$ and assume we have solved the subproblems for:
  1. all $j < i$ and all knapsacks of capacities from 1 to C;
  2. for i we have solved the problem for all capacities $d < c$.

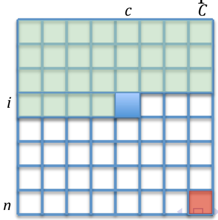
More Dynamic Programming Problems

- **Integer Knapsack Problem (Duplicate Items NOT Allowed)** You have n items (some of which can be identical); item I_i is of weight w_i and value v_i. You also have a knapsack of capacity C. Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

- This is an example of a "2D" recursion; we will be filling a table of size $n \times C$, row by row; subproblems $P(i, c)$ for all $i \leq n$ and $c \leq C$ will be of the form:

  chose from items $I_1, I_2, \ldots, I_i$ a subset which fits in a knapsack of capacity c and is of the largest possible total value.

- Fix now $i \leq n$ and $c \leq C$ and assume we have solved the subproblems for:
  1. all $j < i$ and all knapsacks of capacities from 1 to C;
  2. for i we have solved the problem for all capacities $d < c$.



More Dynamic Programming Problems

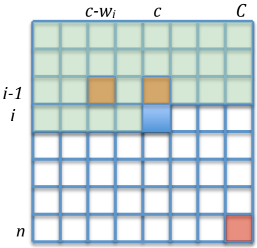
- we now have two options: either we take item I_i or we do not;

More Dynamic Programming Problems

- we now have two options: either we take item I_i or we do not;
- so we look look at optimal solutions $opt(i-1, c-w_i)$ and $opt(i-1, c)$;

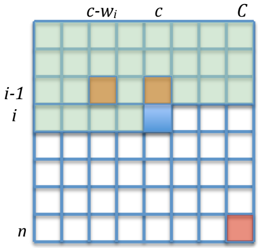
More Dynamic Programming Problems

- we now have two options: either we take item I_i or we do not;
- so we look look at optimal solutions $opt(i-1, c-w_i)$ and $opt(i-1, c)$;



More Dynamic Programming Problems

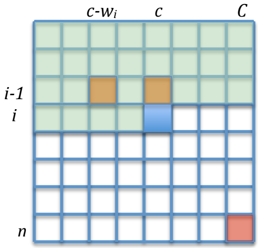
- we now have two options: either we take item I_i or we do not;
- so we look look at optimal solutions $opt(i-1, c-w_i)$ and $opt(i-1, c)$;



- **if** $opt(i-1, c-w_i) + v_i > opt(i-1, c)$
  then $opt(i, c) = opt(i-1, c-w_i) + v_i$;
  else $opt(i, c) = opt(i-1, c)$.

More Dynamic Programming Problems

- we now have two options: either we take item I_i or we do not;
- so we look look at optimal solutions $opt(i-1, c-w_i)$ and $opt(i-1, c)$;



- **if** $opt(i-1, c-w_i) + v_i > opt(i-1, c)$
  then $opt(i, c) = opt(i-1, c-w_i) + v_i$;
  else $opt(i, c) = opt(i-1, c)$.
- Final solution will be given by $opt(n, C)$.

More Dynamic Programming Problems

- **Balanced Partition** You have a set of n integers. Partition these integers into two subsets such that you minimise $|S_1 - S_2|$, where S_1 and S_2 denote the sums of the elements in each of the two subsets.

More Dynamic Programming Problems

- **Balanced Partition** You have a set of n integers. Partition these integers into two subsets such that you minimise $|S_1 - S_2|$, where S_1 and S_2 denote the sums of the elements in each of the two subsets.

- **Solution:** Let S be the total sum of all integers in the set; consider the Knapsack problem (with duplicate items not allowed) with the knapsack of size $S/2$ and with each integer x_i of both size and value equal to x_i.

More Dynamic Programming Problems

- **Balanced Partition** You have a set of n integers. Partition these integers into two subsets such that you minimise $|S_1 - S_2|$, where S_1 and S_2 denote the sums of the elements in each of the two subsets.

- **Solution:** Let S be the total sum of all integers in the set; consider the Knapsack problem (with duplicate items not allowed) with the knapsack of size $S/2$ and with each integer x_i of both size and value equal to x_i.

- **Claim:** the best packing of such knapsack produces optimally balanced partition, with S_1 being all the integers in the knapsack and S_2 all the integers left out of the knapsack.

More Dynamic Programming Problems

- **Balanced Partition** You have a set of n integers. Partition these integers into two subsets such that you minimise $|S_1 - S_2|$, where S_1 and S_2 denote the sums of the elements in each of the two subsets.
- **Solution:** Let S be the total sum of all integers in the set; consider the Knapsack problem (with duplicate items not allowed) with the knapsack of size $S/2$ and with each integer x_i of both size and value equal to x_i.
- **Claim:** the best packing of such knapsack produces optimally balanced partition, with S_1 being all the integers in the knapsack and S_2 all the integers left out of the knapsack.
- Why?

More Dynamic Programming Problems

- **Balanced Partition** You have a set of n integers. Partition these integers into two subsets such that you minimise $|S_1 - S_2|$, where S_1 and S_2 denote the sums of the elements in each of the two subsets.
- **Solution:** Let S be the total sum of all integers in the set; consider the Knapsack problem (with duplicate items not allowed) with the knapsack of size $S/2$ and with each integer x_i of both size and value equal to x_i.
- **Claim:** the best packing of such knapsack produces optimally balanced partition, with S_1 being all the integers in the knapsack and S_2 all the integers left out of the knapsack.
- Why? Since $S = S_1 + S_2$ we obtain

$$\frac{S}{2} - S_1 = \frac{S_1 + S_2}{2} - S_1 = \frac{S_2 - S_1}{2}$$

More Dynamic Programming Problems

- **Balanced Partition** You have a set of n integers. Partition these integers into two subsets such that you minimise $|S_1 - S_2|$, where S_1 and S_2 denote the sums of the elements in each of the two subsets.

- **Solution:** Let S be the total sum of all integers in the set; consider the Knapsack problem (with duplicate items not allowed) with the knapsack of size $S/2$ and with each integer x_i of both size and value equal to x_i.

- **Claim:** the best packing of such knapsack produces optimally balanced partition, with S_1 being all the integers in the knapsack and S_2 all the integers left out of the knapsack.

- Why? Since $S = S_1 + S_2$ we obtain
$$\frac{S}{2} - S_1 = \frac{S_1 + S_2}{2} - S_1 = \frac{S_2 - S_1}{2}$$
i.e. $S_2 - S_1 = 2(S/2 - S_1)$.

- **Balanced Partition** You have a set of n integers. Partition these integers into two subsets such that you minimise $|S_1 - S_2|$, where S_1 and S_2 denote the sums of the elements in each of the two subsets.

- **Solution:** Let S be the total sum of all integers in the set; consider the Knapsack problem (with duplicate items not allowed) with the knapsack of size $S/2$ and with each integer x_i of both size and value equal to x_i.

- **Claim:** the best packing of such knapsack produces optimally balanced partition, with S_1 being all the integers in the knapsack and S_2 all the integers left out of the knapsack.

- Why? Since $S = S_1 + S_2$ we obtain
$$\frac{S}{2} - S_1 = \frac{S_1 + S_2}{2} - S_1 = \frac{S_2 - S_1}{2}$$
i.e. $S_2 - S_1 = 2(S/2 - S_1)$.

- Thus, minimising $S/2 - S_1$ will minimise $S_2 - S_1$.

More Dynamic Programming Problems

- **Balanced Partition** You have a set of n integers. Partition these integers into two subsets such that you minimise $|S_1 - S_2|$, where S_1 and S_2 denote the sums of the elements in each of the two subsets.

- **Solution:** Let S be the total sum of all integers in the set; consider the Knapsack problem (with duplicate items not allowed) with the knapsack of size $S/2$ and with each integer x_i of both size and value equal to x_i.

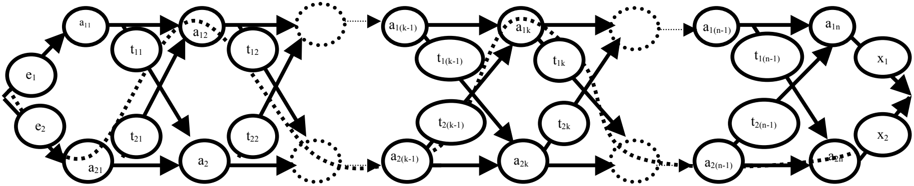
- **Claim:** the best packing of such knapsack produces optimally balanced partition, with S_1 being all the integers in the knapsack and S_2 all the integers left out of the knapsack.

- Why? Since $S = S_1 + S_2$ we obtain
$$\frac{S}{2} - S_1 = \frac{S_1 + S_2}{2} - S_1 = \frac{S_2 - S_1}{2}$$
i.e. $S_2 - S_1 = 2(S/2 - S_1)$.

- Thus, minimising $S/2 - S_1$ will minimise $S_2 - S_1$.

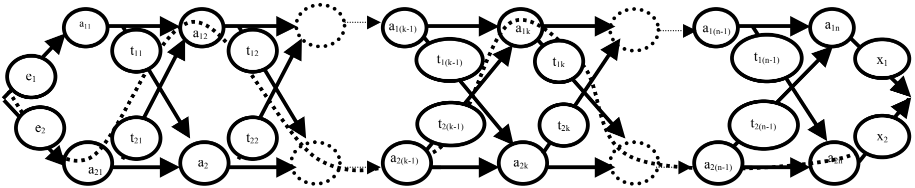
- So, all we have to do is find a subset of these numbers with the largest possible total sum which fits inside a knapsack of size $S/2$.

Dynamic Programming: Assembly line scheduling



Instance: Two assembly lines with workstations for n jobs.

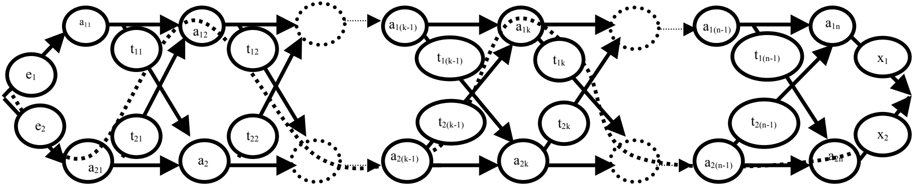
Dynamic Programming: Assembly line scheduling



Instance: Two assembly lines with workstations for n jobs.

- On the first assembly line the k^{th} job takes $a_{1,k}$ $(1 \leq k \leq n)$ units of time to complete; on the second assembly line the same job takes $a_{2,k}$ units of time.

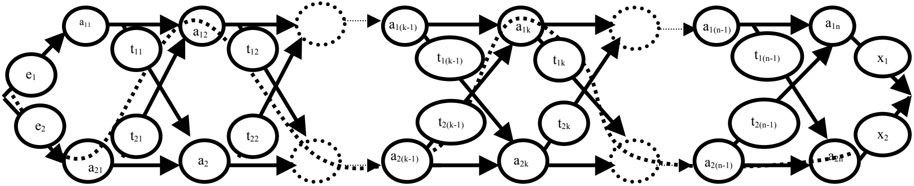
Dynamic Programming: Assembly line scheduling



Instance: Two assembly lines with workstations for n jobs.

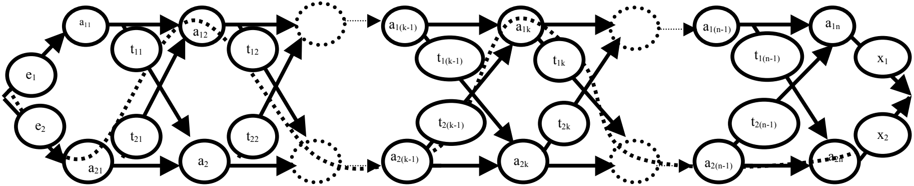
- On the first assembly line the k^{th} job takes $a_{1,k}$ $(1 \leq k \leq n)$ units of time to complete; on the second assembly line the same job takes $a_{2,k}$ units of time.
- To move the product from station $k-1$ on the first assembly line to station k on the second line it takes $t_{1,k-1}$ units of time.

Dynamic Programming: Assembly line scheduling



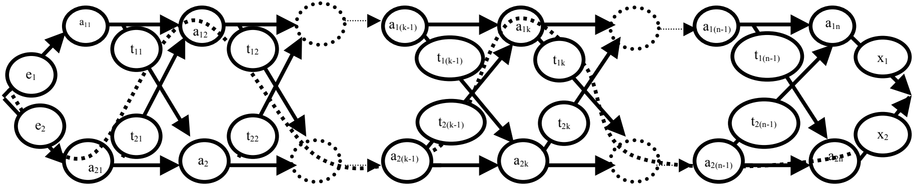
Instance: Two assembly lines with workstations for n jobs.

- On the first assembly line the k^{th} job takes $a_{1,k}$ $(1 \leq k \leq n)$ units of time to complete; on the second assembly line the same job takes $a_{2,k}$ units of time.
- To move the product from station $k-1$ on the first assembly line to station k on the second line it takes $t_{1,k-1}$ units of time.
- Likewise, to move the product from station $k-1$ on the second assembly line to station k on the first assembly line it takes $t_{2,k-1}$ units of time.



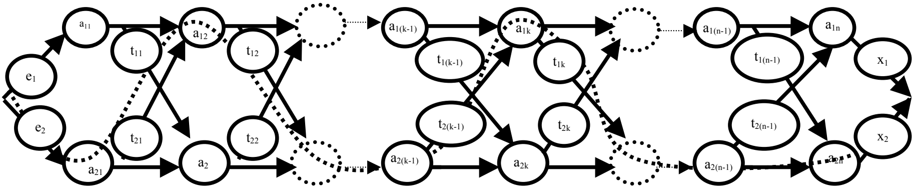
- To bring an unfinished product to the first assembly line it takes e_1 units of time.

Dynamic Programming: Assembly line scheduling



- To bring an unfinished product to the first assembly line it takes e_1 units of time.
- To bring an unfinished product to the second assembly line it takes e_2 units of time.

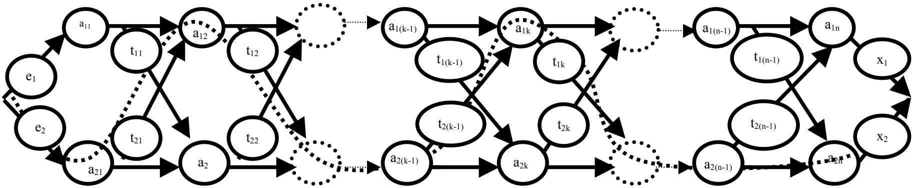
Dynamic Programming: Assembly line scheduling



- To bring an unfinished product to the first assembly line it takes e_1 units of time.
- To bring an unfinished product to the second assembly line it takes e_2 units of time.

- To get a finished product from the first assembly line to the warehouse it takes x_1 units of time;

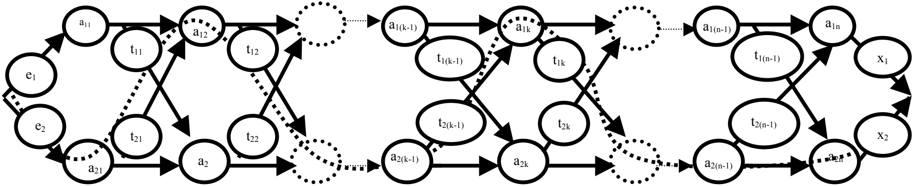
Dynamic Programming: Assembly line scheduling



- To bring an unfinished product to the first assembly line it takes e_1 units of time.
- To bring an unfinished product to the second assembly line it takes e_2 units of time.

- To get a finished product from the first assembly line to the warehouse it takes x_1 units of time;
- To get a finished product from the second assembly line to the warehouse it takes x_2 units.

Dynamic Programming: Assembly line scheduling

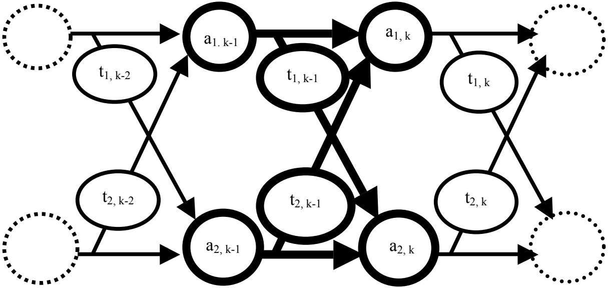


- To bring an unfinished product to the first assembly line it takes e_1 units of time.
- To bring an unfinished product to the second assembly line it takes e_2 units of time.

- To get a finished product from the first assembly line to the warehouse it takes x_1 units of time;
- To get a finished product from the second assembly line to the warehouse it takes x_2 units.

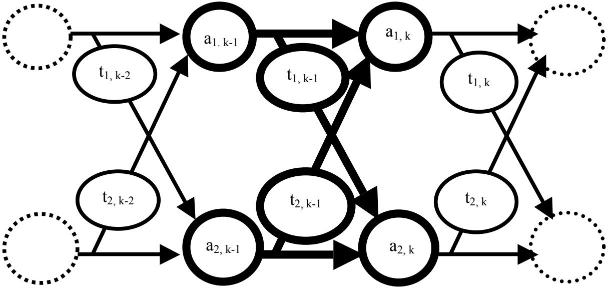
- **Task:** Find a *fastest way* to assemble a product using both lines as necessary.

Dynamic Programming: Assembly line scheduling



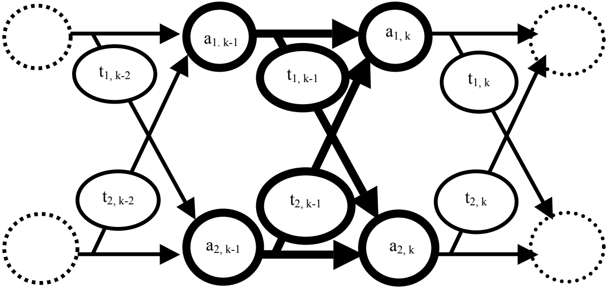
- For each $k \leq n$, we solve subproblems $P(1, k)$ and $P(2, k)$ by a **simultaneous recursion** on k:

Dynamic Programming: Assembly line scheduling

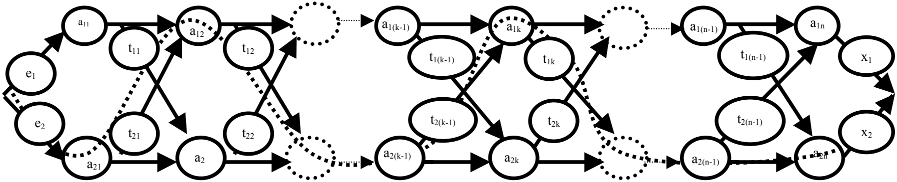


- For each $k \leq n$, we solve subproblems $P(1,k)$ and $P(2,k)$ by a **simultaneous recursion** on k:
- $P(1,k)$: find the minimal amount of time $m(1,k)$ needed to finish the first k jobs, such the k^{th} job is finished on the k^{th} workstation on the **first** assembly line;

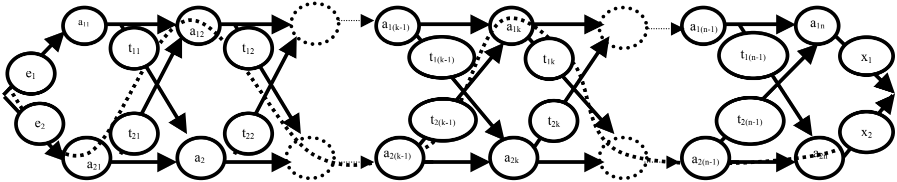
Dynamic Programming: Assembly line scheduling



- For each $k \leq n$, we solve subproblems $P(1, k)$ and $P(2, k)$ by a **simultaneous recursion** on k:
- $P(1, k)$: find the minimal amount of time $m(1, k)$ needed to finish the first k jobs, such the k^{th} job is finished on the k^{th} workstation on the **first** assembly line;
- $P(2, k)$: find the minimal amount of time $m(2, k)$ needed to finish the first k jobs, such the k^{th} job is finished on the k^{th} workstation on the **second** assembly line.

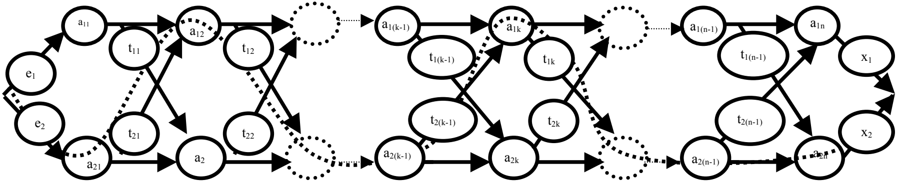


- We solve $P(1,k)$ and $P(2,k)$ by a simultaneous recursion:



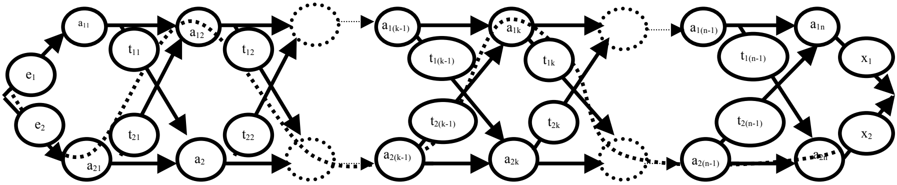
- We solve $P(1,k)$ and $P(2,k)$ by a simultaneous recursion:
- Initially $m(1,1) = e_1 + a_{1,1}$ and $m(2,1) = e_2 + a_{2,1}$.

Dynamic Programming



- We solve $P(1,k)$ and $P(2,k)$ by a simultaneous recursion:
- Initially $m(1,1) = e_1 + a_{1,1}$ and $m(2,1) = e_2 + a_{2,1}$.
- Recursion:

$$m(1,k) = \min\{m(1,k-1) + a_{1,k}, \quad m(2,k-1) + t_{2,k-1} + a_{1,k}\}$$
$$m(2,k) = \min\{m(2,k-1) + a_{2,k}, \quad m(1,k-1) + t_{1,k-1} + a_{2,k}\}$$



- We solve $P(1,k)$ and $P(2,k)$ by a simultaneous recursion:
- Initially $m(1,1) = e_1 + a_{1,1}$ and $m(2,1) = e_2 + a_{2,1}$.
- Recursion:

$$m(1,k) = \min\{m(1,k-1) + a_{1,k}, \quad m(2,k-1) + t_{2,k-1} + a_{1,k}\}$$
$$m(2,k) = \min\{m(2,k-1) + a_{2,k}, \quad m(1,k-1) + t_{1,k-1} + a_{2,k}\}$$

- Finally, after obtaining $m(1,n)$ and $m(2,n)$ we choose

$$opt = \min\{m(1,n) + x_1, \quad m(2,n) + x_2\}.$$

Dynamic Programming: Matrix chain multiplication

- For any three matrices of compatible sizes we have $A(BC) = (AB)C$.

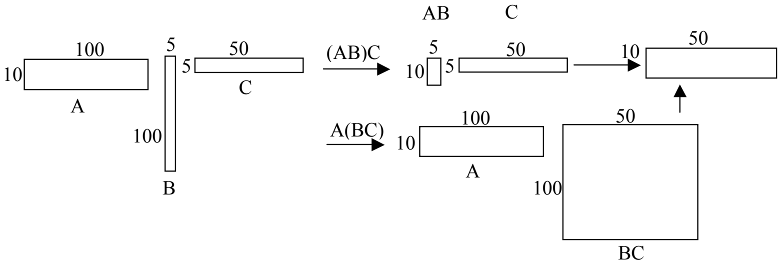
Dynamic Programming: Matrix chain multiplication

- For any three matrices of compatible sizes we have $A(BC) = (AB)C$.
- However, the number of real number multiplications needed to perform in order to obtain the matrix product can be very different:

Dynamic Programming: Matrix chain multiplication

- For any three matrices of compatible sizes we have $A(BC) = (AB)C$.
- However, the number of real number multiplications needed to perform in order to obtain the matrix product can be very different:

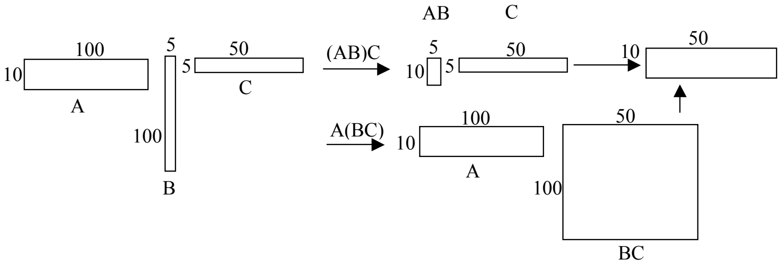
$A = 10 \times 100; \quad B = 100 \times 5; \quad C = 5 \times 50$



Dynamic Programming: Matrix chain multiplication

- For any three matrices of compatible sizes we have $A(BC) = (AB)C$.
- However, the number of real number multiplications needed to perform in order to obtain the matrix product can be very different:

$A = 10 \times 100; \quad B = 100 \times 5; \quad C = 5 \times 50$



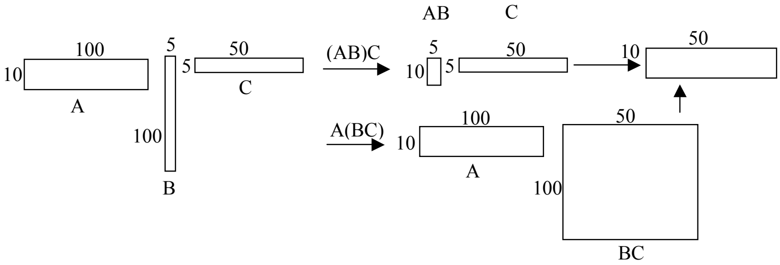
- To evaluate $(AB)C$ we need
$(10 \times 5) \times 100 + (10 \times 50) \times 5 = 5000 + 2500 = 7500$ multiplications;

Dynamic Programming: Matrix chain multiplication

- For any three matrices of compatible sizes we have $A(BC) = (AB)C$.

- However, the number of real number multiplications needed to perform in order to obtain the matrix product can be very different:

A = 10 x 100;   B = 100 x 5;   C = 5 x 50



- To evaluate $(AB)C$ we need
$(10 \times 5) \times 100 + (10 \times 50) \times 5 = 5000 + 2500 = 7500$ multiplications;
- To evaluate $A(BC)$ we need
$(100 \times 50) \times 5 + (10 \times 50) \times 100 = 25000 + 50000 = 75000$ multiplications!

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 ... A_n$;

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1A_2...A_n$;
- **Task:** Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 ... A_n$;
- **Task:** Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The total number of different distributions of brackets is equal to the number of binary trees with n leaves.

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 ... A_n$;
- **Task:** Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The total number of different distributions of brackets is equal to the number of binary trees with n leaves.
- The total number of different distributions of brackets satisfies the following recursion (why?):

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 ... A_n$;
- **Task:** Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The total number of different distributions of brackets is equal to the number of binary trees with n leaves.
- The total number of different distributions of brackets satisfies the following recursion (why?):

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

- One can show that the solution satisfies $T(n) = \Omega(2^n)$.

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 \ldots A_n$;
- **Task:** Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The total number of different distributions of brackets is equal to the number of binary trees with n leaves.
- The total number of different distributions of brackets satisfies the following recursion (why?):

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

- One can show that the solution satisfies $T(n) = \Omega(2^n)$.
- Thus, we cannot do an exhaustive search for the optimal placement of the brackets.

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 \ldots A_n$;

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 ... A_n$;
- **Task:** Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 ... A_n$;
- **Task:** Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The subproblems $P(i, j)$ to be considered are:

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 \ldots A_n$;
- **Task:** Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The subproblems $P(i,j)$ to be considered are:

  "group matrices $A_i A_{i+1} \ldots A_{j-1} A_j$ in such a way as to minimise the total number of multiplications needed to find the product matrix".

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 ... A_n$;
- **Task:** Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The subproblems $P(i,j)$ to be considered are:

  "group matrices $A_i A_{i+1} ... A_{j-1} A_j$ in such a way as to minimise the total number of multiplications needed to find the product matrix".
- Note: this looks like it is a case of a "2D recursion, but we can actually do it with a simple "linear" recursion.

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 ... A_n$;
- **Task:** Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The subproblems $P(i, j)$ to be considered are:

  "group matrices $A_i A_{i+1} ... A_{j-1} A_j$ in such a way as to minimise the total number of multiplications needed to find the product matrix".

- Note: this looks like it is a case of a "2D recursion, but we can actually do it with a simple "linear" recursion.
- We group such subproblems by the value of $j - i$ and perform a recursion on the value of $j - i$.

Dynamic Programming: Matrix chain multiplication

- **Problem Instance:** A sequence of matrices $A_1 A_2 ... A_n$;
- **Task:** Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The subproblems $P(i, j)$ to be considered are:

  "group matrices $A_i A_{i+1} ... A_{j-1} A_j$ in such a way as to minimise the total number of multiplications needed to find the product matrix".

- Note: this looks like it is a case of a "2D recursion, but we can actually do it with a simple "linear" recursion.
- We group such subproblems by the value of $j - i$ and perform a recursion on the value of $j - i$.
- At each recursive step m we solve all subproblems $P(i, j)$ for which $j - i = m$.

Dynamic Programming: Matrix chain multiplication

- Let $m(i,j)$ denote the minimal number of multiplications needed to compute the product $A_i A_{i+1} \dots A_{j-1} A_j$; let also the size of matrix A_i be $s_{i-1} \times s_i$.

Dynamic Programming: Matrix chain multiplication

- Let $m(i,j)$ denote the minimal number of multiplications needed to compute the product $A_i A_{i+1} ... A_{j-1} A_j$; let also the size of matrix A_i be $s_{i-1} \times s_i$.

- **Recursion:** We examine all possible ways to place the principal (outermost) multiplication, splitting the chain into the product $(A_i \dots A_k) \cdot (A_{k+1} \dots A_j)$.

Dynamic Programming: Matrix chain multiplication

- Let $m(i, j)$ denote the minimal number of multiplications needed to compute the product $A_i A_{i+1} ... A_{j-1} A_j$; let also the size of matrix A_i be $s_{i-1} \times s_i$.

- **Recursion:** We examine all possible ways to place the principal (outermost) multiplication, splitting the chain into the product $(A_i \ldots A_k) \cdot (A_{k+1} \ldots A_j)$.

- Note that both $k - i < j - i$ and $j - (k+1) < j - i$; thus we have the solutions of the subproblems $P(i, k)$ and $P(k+1, j)$ already computed and stored in slots $k - i$ and $j - (k+1)$, respectively, which precede slot $j - i$ we are presently filling.

Dynamic Programming: Matrix chain multiplication

- Let $m(i, j)$ denote the minimal number of multiplications needed to compute the product $A_i A_{i+1} ... A_{j-1} A_j$; let also the size of matrix A_i be $s_{i-1} \times s_i$.
- **Recursion:** We examine all possible ways to place the principal (outermost) multiplication, splitting the chain into the product $(A_i \ldots A_k) \cdot (A_{k+1} \ldots A_j)$.
- Note that both $k - i < j - i$ and $j - (k+1) < j - i$; thus we have the solutions of the subproblems $P(i, k)$ and $P(k+1, j)$ already computed and stored in slots $k - i$ and $j - (k+1)$, respectively, which precede slot $j - i$ we are presently filling.
- Note also that the matrix product $A_i \ldots A_k$ is a $s_{i-1} \times s_k$ matrix L and $A_{k+1} \ldots A_j$ is a $s_k \times s_j$ matrix R.

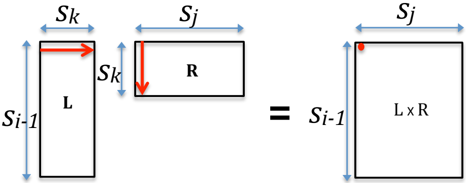
- Let $m(i,j)$ denote the minimal number of multiplications needed to compute the product $A_i A_{i+1} ... A_{j-1} A_j$; let also the size of matrix A_i be $s_{i-1} \times s_i$.

- **Recursion:** We examine all possible ways to place the principal (outermost) multiplication, splitting the chain into the product $(A_i \ldots A_k) \cdot (A_{k+1} \ldots A_j)$.

- Note that both $k - i < j - i$ and $j - (k+1) < j - i$; thus we have the solutions of the subproblems $P(i,k)$ and $P(k+1,j)$ already computed and stored in slots $k - i$ and $j - (k+1)$, respectively, which precede slot $j - i$ we are presently filling.

- Note also that the matrix product $A_i \ldots A_k$ is a $s_{i-1} \times s_k$ matrix L and $A_{k+1} \ldots A_j$ is a $s_k \times s_j$ matrix R.

- To multiply an $s_{i-1} \times s_k$ matrix L and an $s_k \times s_j$ matrix R it takes $s_{i-1} s_k s_j$ many multiplications:



Total number of multiplications: $S_{i-1}\ S_j\ \ S_k$

- The recursion:

$$m(i,j) = \min\{m(i,k) + m(k+1,j) + s_{i-1}s_js_k \ : \ i \leq k \leq j-1\}$$

Dynamic Programming: Matrix chain multiplication

- The recursion:

$$m(i,j) = \min\{m(i,k) + m(k+1,j) + s_{i-1}s_js_k \ : \ i \leq k \leq j-1\}$$

- Note that the recursion step is a brute force search but the whole algorithm is not, because all the subproblems are solved only once, and there are only $O(n^2)$ many such subproblems.

Dynamic Programming: Matrix chain multiplication

- The recursion:

$$m(i,j) = \min\{m(i,k) + m(k+1,j) + s_{i-1}s_js_k \ : \ i \leq k \leq j-1\}$$

- Note that the recursion step is a brute force search but the whole algorithm is not, because all the subproblems are solved only once, and there are only $O(n^2)$ many such subproblems.

- k for which the minimum in the recursive definition of $m(i,j)$ is achieved can be stored to retrieve the optimal placement of brackets for the whole chain $A_1 \ldots A_n$.

Dynamic Programming: Matrix chain multiplication

- The recursion:

$$m(i,j) = \min\{m(i,k) + m(k+1,j) + s_{i-1}s_js_k \ : \ i \leq k \leq j-1\}$$

- Note that the recursion step is a brute force search but the whole algorithm is not, because all the subproblems are solved only once, and there are only $O(n^2)$ many such subproblems.

- k for which the minimum in the recursive definition of $m(i,j)$ is achieved can be stored to retrieve the optimal placement of brackets for the whole chain $A_1 \dots A_n$.

- Thus, in the m^{th} slot of the table we are constructing we store all pairs $(m(i,j),k)$ for which $j-i=m$.

Dynamic Programming: Longest Common Subsequence

- Assume we want to compare how similar two sequences of symbols S and S^* are.

Dynamic Programming: Longest Common Subsequence

- Assume we want to compare how similar two sequences of symbols S and S^* are.

- Example: how similar are the genetic codes of two viruses.

Dynamic Programming: Longest Common Subsequence

- Assume we want to compare how similar two sequences of symbols S and S^* are.

- Example: how similar are the genetic codes of two viruses.

- This can tell us if one is just a genetic mutation of the other.

- Assume we want to compare how similar two sequences of symbols S and S^* are.
- Example: how similar are the genetic codes of two viruses.
- This can tell us if one is just a genetic mutation of the other.
- A sequence s is a **subsequence** of another sequence S if s can be obtained by deleting some of the symbols of S (while preserving the order of the remaining symbols).

Dynamic Programming: Longest Common Subsequence

- Assume we want to compare how similar two sequences of symbols S and S^* are.
- Example: how similar are the genetic codes of two viruses.
- This can tell us if one is just a genetic mutation of the other.
- A sequence s is a **subsequence** of another sequence S if s can be obtained by deleting some of the symbols of S (while preserving the order of the remaining symbols).
- Given two sequences S and S^* a sequence s is a **Longest Common Subsequence** of S, S^* if s is a common subsequence of both S and S^* and is of maximal possible length.

Dynamic Programming: Longest Common Subsequence

- **Instance:** Two sequences $S = \langle a_1, a_2, \ldots a_n \rangle$ and $S^* = \langle b_1, b_2, \ldots, b_m \rangle$.

Dynamic Programming: Longest Common Subsequence

- **Instance:** Two sequences $S = \langle a_1, a_2, \dots a_n \rangle$ and $S^* = \langle b_1, b_2, \dots, b_m \rangle$.
- **Task:** Find a longest common subsequence of S, S^*.

Dynamic Programming: Longest Common Subsequence

- **Instance:** Two sequences $S = \langle a_1, a_2, \ldots a_n \rangle$ and $S^* = \langle b_1, b_2, \ldots, b_m \rangle$.
- **Task:** Find a longest common subsequence of S, S^*.
- We first find *the length* of the longest common subsequence of S, S^*.

Dynamic Programming: Longest Common Subsequence

- **Instance:** Two sequences $S = \langle a_1, a_2, \ldots a_n \rangle$ and $S^* = \langle b_1, b_2, \ldots, b_m \rangle$.

- **Task:** Find a longest common subsequence of S, S^*.

- We first find *the length* of the longest common subsequence of S, S^*.

- "2D recursion": for all $1 \leq i \leq n$ and all $1 \leq j \leq m$ let $c[i, j]$ be the length of the longest common subsequence of the truncated sequences $S_i = \langle a_1, a_2, \ldots a_i \rangle$ and $S^*_j = \langle b_1, b_2, \ldots, b_j \rangle$.

Dynamic Programming: Longest Common Subsequence

- **Instance:** Two sequences $S = \langle a_1, a_2, \ldots a_n \rangle$ and $S^* = \langle b_1, b_2, \ldots, b_m \rangle$.
- **Task:** Find a longest common subsequence of S, S^*.
- We first find *the length* of the longest common subsequence of S, S^*.
- "2D recursion": for all $1 \leq i \leq n$ and all $1 \leq j \leq m$ let $c[i, j]$ be the length of the longest common subsequence of the truncated sequences $S_i = \langle a_1, a_2, \ldots a_i \rangle$ and $S^*_j = \langle b_1, b_2, \ldots, b_j \rangle$.
- Recursion:

Dynamic Programming: Longest Common Subsequence

- **Instance:** Two sequences $S = \langle a_1, a_2, \dots a_n \rangle$ and $S^* = \langle b_1, b_2, \dots, b_m \rangle$.
- **Task:** Find a longest common subsequence of S, S^*.
- We first find *the length* of the longest common subsequence of S, S^*.
- "2D recursion": for all $1 \leq i \leq n$ and all $1 \leq j \leq m$ let $c[i, j]$ be the length of the longest common subsequence of the truncated sequences $S_i = \langle a_1, a_2, \dots a_i \rangle$ and $S^*_j = \langle b_1, b_2, \dots, b_j \rangle$.
- Recursion: we fill the table row by row, so the ordering of subproblems is the lexicographical ordering;

Dynamic Programming: Longest Common Subsequence

- **Instance:** Two sequences $S = \langle a_1, a_2, \dots a_n \rangle$ and $S^* = \langle b_1, b_2, \dots, b_m \rangle$.
- **Task:** Find a longest common subsequence of S, S^*.
- We first find *the length* of the longest common subsequence of S, S^*.
- "2D recursion": for all $1 \leq i \leq n$ and all $1 \leq j \leq m$ let $c[i,j]$ be the length of the longest common subsequence of the truncated sequences $S_i = \langle a_1, a_2, \dots a_i \rangle$ and $S_j^* = \langle b_1, b_2, \dots, b_j \rangle$.
- Recursion: we fill the table row by row, so the ordering of subproblems is the lexicographical ordering;

$$c[i,j] = \begin{cases} 0, & \text{if } \; i = 0 \text{ or } j = 0; \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } a_i = b_j; \\ \max\{c[i-1, j], c[i, j-1]\} & \text{if } i, j > 0 \text{ and } a_i \neq b_j. \end{cases}$$

Retrieving a longest common subsequence:

```
LCS-LENGTH(X, Y)
  m ← length[X]
  n ← length[Y]
  for i ← 1 to m
       do c[i, 0] ← 0
  for j ← 0 to n
       do c[0, j] ← 0
  for i ← 1 to m
       do for j ← 1 to n
              do if x_i = y_j
                    then c[i, j] ← c[i − 1, j − 1] + 1
                         b[i, j] ← "↖"
                    else if c[i − 1, j] ≥ c[i, j − 1]
                            then c[i, j] ← c[i − 1, j]
                                 b[i, j] ← "↑"
                            else c[i, j] ← c[i, j − 1]
                                 b[i, j] ← "←"
  return c and b
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Dynamic Programming: Longest Common Subsequence

- What if we have to find a longest common subsequence of three sequences S_1, S_2, S_3?

Dynamic Programming: Longest Common Subsequence

- What if we have to find a longest common subsequence of three sequences S_1, S_2, S_3?
- Can we do LCS(LCS(S_1, S_2), S_3)?

Dynamic Programming: Longest Common Subsequence

- What if we have to find a longest common subsequence of three sequences S_1, S_2, S_3?
- Can we do $\mathrm{LCS}(\mathrm{LCS}(S_1, S_2), S_3)$?
- Not necessarily! Consider

$$
\begin{aligned}
S_1 &= ABCDEGG & \quad \mathrm{LCS}(S_1, S_2) &= ABEG \\
S_2 &= ACBEEFG & \quad \mathrm{LCS}S_2, S_3) &= ACEF \\
S_3 &= ACCEDGF & \quad \mathrm{LCS}(S_1, S_3) &= ACDG
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{LCS}(\mathrm{LCS}(S_1, S_2), S_3) &= \mathrm{LCS}(ABEG, ACCEDGF) = AEG \\
\mathrm{LCS}(\mathrm{LCS}(S_2, S_3), S_1) &= \mathrm{LCS}(ACEF, ABCDEGG) = ACE \\
\mathrm{LCS}(\mathrm{LCS}(S_1, S_3), S_2) &= \mathrm{LCS}(ACDG, ACBEEFG) = ACG
\end{aligned}
$$

But

$$\mathrm{LCS}(S_1, S_2, S_3) = ACEG$$

Dynamic Programming: Longest Common Subsequence

- What if we have to find a longest common subsequence of three sequences S_1, S_2, S_3?
- Can we do $\text{LCS}(\text{LCS}(S_1, S_2), S_3)$?
- Not necessarily! Consider

$$
\begin{aligned}
S_1 &= ABCDEGG & \qquad \text{LCS}(S_1, S_2) &= ABEG \\
S_2 &= ACBEEFG & \qquad \text{LCS}S_2, S_3) &= ACEF \\
S_3 &= ACCEDGF & \qquad \text{LCS}(S_1, S_3) &= ACDG
\end{aligned}
$$

$$
\begin{aligned}
\text{LCS}(\text{LCS}(S_1, S_2), S_3) &= \text{LCS}(ABEG, ACCEDGF) = AEG \\
\text{LCS}(\text{LCS}(S_2, S_3), S_1) &= \text{LCS}(ACEF, ABCDEGG) = ACE \\
\text{LCS}(\text{LCS}(S_1, S_3), S_2) &= \text{LCS}(ACDG, ACBEEFG) = ACG
\end{aligned}
$$

But

$$\text{LCS}(S_1, S_2, S_3) = ACEG$$

- So how would you design an algorithm which computes correctly $\text{LCS}(S_1, S_2, S_3)$?

Dynamic Programming: Longest Common Subsequence

- **Instance:** Three sequences $S = \langle a_1, a_2, \ldots a_n \rangle$, $S^* = \langle b_1, b_2, \ldots, b_m \rangle$ and $S^{**} = \langle c_1, c_2, \ldots, c_k \rangle$.

Dynamic Programming: Longest Common Subsequence

- **Instance:** Three sequences $S = \langle a_1, a_2, \ldots a_n \rangle$, $S^* = \langle b_1, b_2, \ldots, b_m \rangle$ and $S^{**} = \langle c_1, c_2, \ldots, c_k \rangle$.
- **Task:** Find a longest common subsequence of S, S^*, S^{**}.

Dynamic Programming: Longest Common Subsequence

- **Instance:** Three sequences $S = \langle a_1, a_2, \ldots a_n \rangle$, $S^* = \langle b_1, b_2, \ldots, b_m \rangle$ and $S^{**} = \langle c_1, c_2, \ldots, c_k \rangle$.
- **Task:** Find a longest common subsequence of S, S^*, S^{**}.
- We again first find *the length* of the longest common subsequence of S, S^*, S^{**}.

Dynamic Programming: Longest Common Subsequence

- **Instance:** Three sequences $S = \langle a_1, a_2, \dots a_n \rangle$, $S^* = \langle b_1, b_2, \dots, b_m \rangle$ and $S^{**} = \langle c_1, c_2, \dots, c_k \rangle$.

- **Task:** Find a longest common subsequence of S, S^*, S^{**}.

- We again first find *the length* of the longest common subsequence of S, S^*, S^{**}.

- for all $1 \leq i \leq n$, all $1 \leq j \leq m$ and all $l \leq k$, let $d[i, j, l]$ be the length of the longest common subsequence of the truncated sequences $S_i = \langle a_1, a_2, \dots a_i \rangle$, $S_j^* = \langle b_1, b_2, \dots, b_j \rangle$ and $S_l^* = \langle c_1, c_2, \dots, c_l \rangle$.

Dynamic Programming: Longest Common Subsequence

- **Instance:** Three sequences $S = \langle a_1, a_2, \dots a_n \rangle$, $S^* = \langle b_1, b_2, \dots, b_m \rangle$ and $S^{**} = \langle c_1, c_2, \dots, c_k \rangle$.
- **Task:** Find a longest common subsequence of S, S^*, S^{**}.
- We again first find *the length* of the longest common subsequence of S, S^*, S^{**}.
- for all $1 \leq i \leq n$, all $1 \leq j \leq m$ and all $l \leq k$, let $d[i, j, l]$ be the length of the longest common subsequence of the truncated sequences $S_i = \langle a_1, a_2, \dots a_i \rangle$, $S^*_j = \langle b_1, b_2, \dots, b_j \rangle$ and $S^*_l = \langle c_1, c_2, \dots, c_l \rangle$.
- Recursion:

$$d[i, j, l] = \begin{cases} 0, & \text{if } \; i = 0 \text{ or } j = 0 \text{ or } l = 0; \\ d[i-1, j-1, l-1] + 1 & \text{if } i, j, l > 0 \text{ and } a_i = b_j = c_l; \\ \max\{d[i-1, j, l], d[i, j-1, l], d[i, j, l-1]\} & \text{otherwise.} \end{cases}$$

- **Instance:** Two sequences $s = \langle a_1, a_2, \ldots a_n \rangle$ and $s^* = \langle b_1, b_2, \ldots, b_m \rangle$.

Dynamic Programming: Shortest Common Supersequence

- **Instance:** Two sequences $s = \langle a_1, a_2, \ldots a_n \rangle$ and $s^* = \langle b_1, b_2, \ldots, b_m \rangle$.
- **Task:** Find a shortest common super-sequence S of s, s^*, i.e., the shortest possible sequence S such that both s and s^* are subsequences of S.

Dynamic Programming: Shortest Common Supersequence

- **Instance:** Two sequences $s = \langle a_1, a_2, \ldots a_n \rangle$ and $s^* = \langle b_1, b_2, \ldots, b_m \rangle$.
- **Task:** Find a shortest common super-sequence S of s, s^*, i.e., the shortest possible sequence S such that both s and s^* are subsequences of S.
- **Solution:** Find the longest common subsequence $LCS(s, s^*)$ of s and s^* and then add differing elements of the two sequences at the right places, in any order; for example:

$$\begin{aligned} s &= a\mathbf{b}a\mathbf{cad}a \\ s^* &= x\mathbf{b}y\mathbf{ca}z\mathbf{d} \\ LCS(s, s^*) &= \mathbf{bcad} \\ \text{shortest super-sequence } S &= ax\mathbf{b}ya\mathbf{ca}z\mathbf{d}a \end{aligned}$$

Dynamic Programming: Bellman Ford algorithm

- One of the earliest use of Dynamic Programming (1950's) invented by Bellman.

Dynamic Programming: Bellman Ford algorithm

- One of the earliest use of Dynamic Programming (1950's) invented by Bellman.
- **Instance:** A directed weighted graph $G = (V, E)$ with weights which can be negative, but without cycles of negative total weight and a vertex $s \in V$.

Dynamic Programming: Bellman Ford algorithm

- One of the earliest use of Dynamic Programming (1950's) invented by Bellman.
- **Instance:** A directed weighted graph $G = (V, E)$ with weights which can be negative, but without cycles of negative total weight and a vertex $s \in V$.
- **Goal:** Find the shortest path from vertex s to every other vertex t.

Dynamic Programming: Bellman Ford algorithm

- One of the earliest use of Dynamic Programming (1950's) invented by Bellman.
- **Instance:** A directed weighted graph $G = (V, E)$ with weights which can be negative, but without cycles of negative total weight and a vertex $s \in V$.
- **Goal:** Find the shortest path from vertex s to every other vertex t.
- **Solution:** Since there are no negative weight cycles, the shortest path cannot contain cycles, because a cycle can be excised to produce a shorter path.

Dynamic Programming: Bellman Ford algorithm

- One of the earliest use of Dynamic Programming (1950's) invented by Bellman.
- **Instance:** A directed weighted graph $G = (V, E)$ with weights which can be negative, but without cycles of negative total weight and a vertex $s \in V$.
- **Goal:** Find the shortest path from vertex s to every other vertex t.
- **Solution:** Since there are no negative weight cycles, the shortest path cannot contain cycles, because a cycle can be excised to produce a shorter path.
- Thus, every shortest path can have at most $|V| - 1$ edges.

Dynamic Programming: Bellman Ford algorithm

- One of the earliest use of Dynamic Programming (1950's) invented by Bellman.
- **Instance:** A directed weighted graph $G = (V, E)$ with weights which can be negative, but without cycles of negative total weight and a vertex $s \in V$.
- **Goal:** Find the shortest path from vertex s to every other vertex t.
- **Solution:** Since there are no negative weight cycles, the shortest path cannot contain cycles, because a cycle can be excised to produce a shorter path.
- Thus, every shortest path can have at most $|V| - 1$ edges.
- **Subproblems:** For every $v \in V$ and every i, $(1 \leq i \leq n - 1)$, let $\mathrm{opt}(i, v)$ be the length of a shortest path from s to v which contains at most i edges.

Dynamic Programming: Bellman Ford algorithm

- One of the earliest use of Dynamic Programming (1950's) invented by Bellman.
- **Instance:** A directed weighted graph $G = (V, E)$ with weights which can be negative, but without cycles of negative total weight and a vertex $s \in V$.
- **Goal:** Find the shortest path from vertex s to every other vertex t.
- **Solution:** Since there are no negative weight cycles, the shortest path cannot contain cycles, because a cycle can be excised to produce a shorter path.
- Thus, every shortest path can have at most $|V| - 1$ edges.
- **Subproblems:** For every $v \in V$ and every i, $(1 \leq i \leq n-1)$, let $\text{opt}(i, v)$ be the length of a shortest path from s to v which contains at most i edges.
- Our goal is to find for every vertex $t \in G$ the value of $\text{opt}(n-1, t)$ and the path which achieves such a length.

Dynamic Programming: Bellman Ford algorithm

- One of the earliest use of Dynamic Programming (1950's) invented by Bellman.
- **Instance:** A directed weighted graph $G = (V, E)$ with weights which can be negative, but without cycles of negative total weight and a vertex $s \in V$.
- **Goal:** Find the shortest path from vertex s to every other vertex t.
- **Solution:** Since there are no negative weight cycles, the shortest path cannot contain cycles, because a cycle can be excised to produce a shorter path.
- Thus, every shortest path can have at most $|V| - 1$ edges.
- **Subproblems:** For every $v \in V$ and every i, $(1 \leq i \leq n - 1)$, let $\mathrm{opt}(i, v)$ be the length of a shortest path from s to v which contains at most i edges.
- Our goal is to find for every vertex $t \in G$ the value of $\mathrm{opt}(n - 1, t)$ and the path which achieves such a length.
- Note that if the shortest path from a vertex v to t is $(v, p_1, p_2, \ldots, p_k, t)$ then $(p_1, p_2, \ldots, p_k, t)$ must be the shortest path from p_1 to t, and $(v, p_1, p_2, \ldots, p_k)$ must also be the shortest path from v to p_k.

Dynamic Programming: Bellman Ford algorithm

- **Recursion:** $\mathrm{opt}(i, v) = \min(\mathrm{opt}(i-1, v), \min_{p \in V}\{\mathrm{opt}(i-1, p) + \mathrm{w}(e(p, v))\}$.
  (here $\mathrm{w}(e(p, v))$ is the weight of the edge $e(p, v)$ from vertex v to vertex p.)

Dynamic Programming: Bellman Ford algorithm

- **Recursion:** $\mathrm{opt}(i, v) = \min(\mathrm{opt}(i-1, v), \min_{p \in V}\{\mathrm{opt}(i-1, p) + \mathrm{w}(e(p, v))\}$.
  (here $\mathrm{w}(e(p, v))$ is the weight of the edge $e(p, v)$ from vertex v to vertex p.)

- Computation $\mathrm{opt}(i, v)$ runs in time $O(|V| \times |E|)$, because $i \leq |V| - 1$ and for each v, min is taken over all edges $e(p, v)$ incident to v; thus in each round all edges are inspected.

Dynamic Programming: Bellman Ford algorithm

- **Recursion:** $\mathrm{opt}(i, v) = \min(\mathrm{opt}(i-1, v), \min_{p \in V}\{\mathrm{opt}(i-1, p) + \mathrm{w}(e(p, v))\}$.
  (here $\mathrm{w}(e(p, v))$ is the weight of the edge $e(p, v)$ from vertex v to vertex p.)

- Computation $\mathrm{opt}(i, v)$ runs in time $O(|V| \times |E|)$, because $i \leq |V| - 1$ and for each v, min is taken over all edges $e(p, v)$ incident to v; thus in each round all edges are inspected.

- Algorithm produces shortest paths from s to every other vertex in the graph.

Dynamic Programming: Bellman Ford algorithm

- **Recursion:** $\text{opt}(i, v) = \min(\text{opt}(i-1, v), \min_{p \in V}\{\text{opt}(i-1, p) + \text{w}(e(p, v))\}.$
  (here $\text{w}(e(p, v))$ is the weight of the edge $e(p, v)$ from vertex v to vertex p.)

- Computation $\text{opt}(i, v)$ runs in time $O(|V| \times |E|)$, because $i \leq |V| - 1$ and for each v, min is taken over all edges $e(p, v)$ incident to v; thus in each round all edges are inspected.

- Algorithm produces shortest paths from s to every other vertex in the graph.

- The method employed is sometimes called "*relaxation*", because we progressively relax the additional constraint on how many edges the shortest paths can contain.

Dynamic Programming: Bellman Ford algorithm

- **Recursion:** $\mathrm{opt}(i, v) = \min(\mathrm{opt}(i-1, v), \min_{p \in V}\{\mathrm{opt}(i-1, p) + \mathrm{w}(e(p, v))\}$.
  (here $\mathrm{w}(e(p, v))$ is the weight of the edge $e(p, v)$ from vertex v to vertex p.)

- Computation $\mathrm{opt}(i, v)$ runs in time $O(|V| \times |E|)$, because $i \leq |V| - 1$ and for each v, min is taken over all edges $e(p, v)$ incident to v; thus in each round all edges are inspected.

- Algorithm produces shortest paths from s to every other vertex in the graph.

- The method employed is sometimes called "*relaxation*", because we progressively relax the additional constraint on how many edges the shortest paths can contain.

- As an exercise, modify the above algorithm to solve the following problem: let G be a weighted directed graph with no cycles of negative total weight and let t be a fixed vertex. Find the shortest paths from every other vertex in the graph to the destination vertex t.

Dynamic Programming: Bellman Ford algorithm

- **Recursion**: $\mathrm{opt}(i, v) = \min(\mathrm{opt}(i-1, v), \min_{p \in V}\{\mathrm{opt}(i-1, p) + \mathrm{w}(e(p, v))\}$.
  (here $\mathrm{w}(e(p, v))$ is the weight of the edge $e(p, v)$ from vertex v to vertex p.)
- Computation $\mathrm{opt}(i, v)$ runs in time $O(|V| \times |E|)$, because $i \leq |V| - 1$ and for each v, min is taken over all edges $e(p, v)$ incident to v; thus in each round all edges are inspected.
- Algorithm produces shortest paths from s to every other vertex in the graph.
- The method employed is sometimes called "*relaxation*", because we progressively relax the additional constraint on how many edges the shortest paths can contain.
- As an exercise, modify the above algorithm to solve the following problem: let G be a weighted directed graph with no cycles of negative total weight and let t be a fixed vertex. Find the shortest paths from every other vertex in the graph to the destination vertex t.

Dynamic Programming: Floyd Warshall algorithm

- Let again $G = (V, E)$ be a directed weighted graph where $V = \{v_1, v_2, \ldots, v_n\}$ and where weights $\mathrm{w}(e(v_p, v_q))$ of edges $e(v_p, v_q)$ can be negative, but there are no negative weight cycles.

Dynamic Programming: Floyd Warshall algorithm

- Let again $G = (V, E)$ be a directed weighted graph where $V = \{v_1, v_2, \ldots, v_n\}$ and where weights $\mathrm{w}(e(v_p, v_q))$ of edges $e(v_p, v_q)$ can be negative, but there are no negative weight cycles.

- We can use a somewhat similar idea to obtain the shortest paths from **every** vertex v_p to **every** vertex v_q (including back to v_p).

Dynamic Programming: Floyd Warshall algorithm

- Let again $G = (V, E)$ be a directed weighted graph where $V = \{v_1, v_2, \ldots, v_n\}$ and where weights $\mathrm{w}(e(v_p, v_q))$ of edges $e(v_p, v_q)$ can be negative, but there are no negative weight cycles.

- We can use a somewhat similar idea to obtain the shortest paths from **every** vertex v_p to **every** vertex v_q (including back to v_p).

- Let $\mathrm{opt}(k, v_p, v_q)$ be the shortest path from a vertex v_p to a vertex v_q such that all intermediate vertices are among vertices $\{v_1, v_2, \ldots, v_k\}$, $(1 \leq k \leq n)$.

Dynamic Programming: Floyd Warshall algorithm

- Let again $G = (V, E)$ be a directed weighted graph where $V = \{v_1, v_2, \ldots, v_n\}$ and where weights $\mathrm{w}(e(v_p, v_q))$ of edges $e(v_p, v_q)$ can be negative, but there are no negative weight cycles.
- We can use a somewhat similar idea to obtain the shortest paths from **every** vertex v_p to **every** vertex v_q (including back to v_p).
- Let $\mathrm{opt}(k, v_p, v_q)$ be the shortest path from a vertex v_p to a vertex v_q such that all intermediate vertices are among vertices $\{v_1, v_2, \ldots, v_k\}$, $(1 \leq k \leq n)$.
- Then

$$\mathrm{opt}(k, v_p, v_q) = \min\{\mathrm{opt}(k-1, v_p, v_q), \mathrm{opt}(k-1, v_p, v_k) + \mathrm{opt}(k-1, v_k, v_q)\}$$

Dynamic Programming: Floyd Warshall algorithm

- Let again $G = (V, E)$ be a directed weighted graph where $V = \{v_1, v_2, \ldots, v_n\}$ and where weights $\mathrm{w}(e(v_p, v_q))$ of edges $e(v_p, v_q)$ can be negative, but there are no negative weight cycles.

- We can use a somewhat similar idea to obtain the shortest paths from **every** vertex v_p to **every** vertex v_q (including back to v_p).

- Let $\mathrm{opt}(k, v_p, v_q)$ be the shortest path from a vertex v_p to a vertex v_q such that all intermediate vertices are among vertices $\{v_1, v_2, \ldots, v_k\}$, $(1 \leq k \leq n)$.

- Then

$$\mathrm{opt}(k, v_p, v_q) = \min\{\mathrm{opt}(k-1, v_p, v_q), \mathrm{opt}(k-1, v_p, v_k) + \mathrm{opt}(k-1, v_k, v_q)\}$$

- Thus, we gradually **relax** the constraint that the intermediary vertices have to belong to $\{v_1, v_2, \ldots, v_k\}$.

Dynamic Programming: Floyd Warshall algorithm

- Let again $G = (V, E)$ be a directed weighted graph where $V = \{v_1, v_2, \ldots, v_n\}$ and where weights $\mathrm{w}(e(v_p, v_q))$ of edges $e(v_p, v_q)$ can be negative, but there are no negative weight cycles.

- We can use a somewhat similar idea to obtain the shortest paths from **every** vertex v_p to **every** vertex v_q (including back to v_p).

- Let $\mathrm{opt}(k, v_p, v_q)$ be the shortest path from a vertex v_p to a vertex v_q such that all intermediate vertices are among vertices $\{v_1, v_2, \ldots, v_k\}$, $(1 \leq k \leq n)$.

- Then

$$\mathrm{opt}(k, v_p, v_q) = \min\{\mathrm{opt}(k-1, v_p, v_q), \mathrm{opt}(k-1, v_p, v_k) + \mathrm{opt}(k-1, v_k, v_q)\}$$

- Thus, we gradually **relax** the constraint that the intermediary vertices have to belong to $\{v_1, v_2, \ldots, v_k\}$.

- Algorithm runs in time $|V|^3$.

Dynamic Programming: Floyd Warshall algorithm

- Let again $G = (V, E)$ be a directed weighted graph where $V = \{v_1, v_2, \ldots, v_n\}$ and where weights $\mathrm{w}(e(v_p, v_q))$ of edges $e(v_p, v_q)$ can be negative, but there are no negative weight cycles.

- We can use a somewhat similar idea to obtain the shortest paths from **every** vertex v_p to **every** vertex v_q (including back to v_p).

- Let $\mathrm{opt}(k, v_p, v_q)$ be the shortest path from a vertex v_p to a vertex v_q such that all intermediate vertices are among vertices $\{v_1, v_2, \ldots, v_k\}$, $(1 \leq k \leq n)$.

- Then

$$\mathrm{opt}(k, v_p, v_q) = \min\{\mathrm{opt}(k-1, v_p, v_q), \mathrm{opt}(k-1, v_p, v_k) + \mathrm{opt}(k-1, v_k, v_q)\}$$

- Thus, we gradually **relax** the constraint that the intermediary vertices have to belong to $\{v_1, v_2, \ldots, v_k\}$.

- Algorithm runs in time $|V|^3$.

- As an exercise, given any directed weighted graph G, explain how we can use the above algorithm to decide if G has any cycles of negative weight.

Dynamic Programming: Edit Distance

- **Edit Distance** Given two text strings A of length n and B of length m, you want to transform A into B. You are allowed to insert a character, delete a character and to replace a character with another one. An insertion costs c_i, a deletion costs c_d and a replacement costs c_r.

Dynamic Programming: Edit Distance

- **Edit Distance** Given two text strings A of length n and B of length m, you want to transform A into B. You are allowed to insert a character, delete a character and to replace a character with another one. An insertion costs c_i, a deletion costs c_d and a replacement costs c_r.

- Task: find the lowest total cost transformation of A into B.

Dynamic Programming: Edit Distance

- **Edit Distance** Given two text strings A of length n and B of length m, you want to transform A into B. You are allowed to insert a character, delete a character and to replace a character with another one. An insertion costs c_i, a deletion costs c_d and a replacement costs c_r.

- Task: find the lowest total cost transformation of A into B.

- Note: if all operations have a unit cost, then you are looking for the minimal number of such operations required to transform A into B; this number is called *the edit distance* between A and B.

Dynamic Programming: Edit Distance

- **Edit Distance** Given two text strings A of length n and B of length m, you want to transform A into B. You are allowed to insert a character, delete a character and to replace a character with another one. An insertion costs c_i, a deletion costs c_d and a replacement costs c_r.

- Task: find the lowest total cost transformation of A into B.

- Note: if all operations have a unit cost, then you are looking for the minimal number of such operations required to transform A into B; this number is called *the edit distance* between A and B.

- If the sequences are sequences of DNA bases and the costs reflect the probabilities of the corresponding mutations, then the minimal cost represents how closely related the two sequences are.

Dynamic Programming: Edit Distance

- **Edit Distance** Given two text strings A of length n and B of length m, you want to transform A into B. You are allowed to insert a character, delete a character and to replace a character with another one. An insertion costs c_i, a deletion costs c_d and a replacement costs c_r.

- Task: find the lowest total cost transformation of A into B.

- Note: if all operations have a unit cost, then you are looking for the minimal number of such operations required to transform A into B; this number is called *the edit distance* between A and B.

- If the sequences are sequences of DNA bases and the costs reflect the probabilities of the corresponding mutations, then the minimal cost represents how closely related the two sequences are.

- Subproblems: Let $C(i, j)$ be the minimum cost of transforming the sequence $A[1..i]$ into the sequence $B[1..j]$ for all $i \leq n$ and all $j \leq m$.

Dynamic Programming: Edit Distance

- **Subproblems $P(i, j)$:** Find the minimum cost $C(i, j)$ of transforming the sequence $A[1..i]$ into the sequence $B[1..j]$ for all $i \leq n$ and all $j \leq m$.

Dynamic Programming: Edit Distance

- **Subproblems $P(i,j)$:** Find the minimum cost $C(i,j)$ of transforming the sequence $A[1..i]$ into the sequence $B[1..j]$ for all $i \leq n$ and all $j \leq m$.

- **Recursion:** we again fill the table of solutions $C(i,j)$ for subproblems $P(i,j)$ row by row (why is this OK?):

$$C(i,j) = \min \begin{cases} c_d + C(i-1,j) \\ C(i,j-1) + c_i \\ \begin{cases} C(i-1,j-1) & \text{if } \; A[i] = B[j] \\ C(i-1,j-1) + c_r & \text{if } \; A[i] \neq B[j] \end{cases} \end{cases}$$

- cost $c_d + C(i-1,j)$ corresponds to the option if you first delete $A[i]$ and then recursively transform $A[1..i-1]$ into $B[1..j]$;

Dynamic Programming: Edit Distance

- **Subproblems $P(i, j)$:** Find the minimum cost $C(i, j)$ of transforming the sequence $A[1..i]$ into the sequence $B[1..j]$ for all $i \leq n$ and all $j \leq m$.

- **Recursion:** we again fill the table of solutions $C(i, j)$ for subproblems $P(i, j)$ row by row (why is this OK?):

$$C(i,j) = \min \begin{cases} c_d + C(i-1, j) \\ C(i, j-1) + c_i \\ \begin{cases} C(i-1, j-1) & \text{if} \quad A[i] = B[j] \\ C(i-1, j-1) + c_r & \text{if} \quad A[i] \neq B[j] \end{cases} \end{cases}$$

- cost $c_d + C(i-1, j)$ corresponds to the option if you first delete $A[i]$ and then recursively transform $A[1..i-1]$ into $B[1..j]$;
- cost $C(i, j-1) + c_i$ corresponds to the option if you first transform $A[1..i]$ to $B[1..j-1]$ and then append $B[j]$ at the end;

Dynamic Programming: Edit Distance

- **Subproblems** $P(i,j)$**:** Find the minimum cost $C(i,j)$ of transforming the sequence $A[1..i]$ into the sequence $B[1..j]$ for all $i \leq n$ and all $j \leq m$.

- **Recursion:** we again fill the table of solutions $C(i,j)$ for subproblems $P(i,j)$ row by row (why is this OK?):

$$C(i,j) = \min \begin{cases} c_d + C(i-1,j) \\ C(i,j-1) + c_i \\ \begin{cases} C(i-1,j-1) & \text{if } A[i] = B[j] \\ C(i-1,j-1) + c_r & \text{if } A[i] \neq B[j] \end{cases} \end{cases}$$

- cost $c_d + C(i-1,j)$ corresponds to the option if you first delete $A[i]$ and then recursively transform $A[1..i-1]$ into $B[1..j]$;
- cost $C(i,j-1) + c_i$ corresponds to the option if you first transform $A[1..i]$ to $B[1..j-1]$ and then append $B[j]$ at the end;
- the third option corresponds to first transforming $A[1..i-1]$ to $B[1..j-1]$ and

Dynamic Programming: Edit Distance

- **Subproblems** $P(i,j)$**:** Find the minimum cost $C(i,j)$ of transforming the sequence $A[1..i]$ into the sequence $B[1..j]$ for all $i \leq n$ and all $j \leq m$.

- **Recursion:** we again fill the table of solutions $C(i,j)$ for subproblems $P(i,j)$ row by row (why is this OK?):

$$C(i,j) = \min \begin{cases} c_d + C(i-1,j) \\ C(i,j-1) + c_i \\ \begin{cases} C(i-1,j-1) & \text{if} \quad A[i] = B[j] \\ C(i-1,j-1) + c_r & \text{if} \quad A[i] \neq B[j] \end{cases} \end{cases}$$

- cost $c_d + C(i-1,j)$ corresponds to the option if you first delete $A[i]$ and then recursively transform $A[1..i-1]$ into $B[1..j]$;
- cost $C(i,j-1) + c_i$ corresponds to the option if you first transform $A[1..i]$ to $B[1..j-1]$ and then append $B[j]$ at the end;
- the third option corresponds to first transforming $A[1..i-1]$ to $B[1..j-1]$ and
  1. if $A[i]$ is already equal to $B[j]$ do nothing, thus incurring a cost of only $C(i-1,j-1)$;

Dynamic Programming: Edit Distance

- **Subproblems $P(i,j)$:** Find the minimum cost $C(i,j)$ of transforming the sequence $A[1..i]$ into the sequence $B[1..j]$ for all $i \leq n$ and all $j \leq m$.

- **Recursion:** we again fill the table of solutions $C(i,j)$ for subproblems $P(i,j)$ row by row (why is this OK?):

$$C(i,j) = \min \begin{cases} c_d + C(i-1,j) \\ C(i,j-1) + c_i \\ \begin{cases} C(i-1,j-1) & \text{if } \ A[i] = B[j] \\ C(i-1,j-1) + c_r & \text{if } \ A[i] \neq B[j] \end{cases} \end{cases}$$

- cost $c_d + C(i-1,j)$ corresponds to the option if you first delete $A[i]$ and then recursively transform $A[1..i-1]$ into $B[1..j]$;
- cost $C(i,j-1) + c_i$ corresponds to the option if you first transform $A[1..i]$ to $B[1..j-1]$ and then append $B[j]$ at the end;
- the third option corresponds to first transforming $A[1..i-1]$ to $B[1..j-1]$ and
  1. if $A[i]$ is already equal to $B[j]$ do nothing, thus incurring a cost of only $C(i-1,j-1)$;
  2. if $A[i]$ is not equal to $B[j]$ replace $A[i]$ by $B[j]$ with a total cost of $C(i-1,j-1) + c_r$.

Dynamic Programming: Maximizing an expression

- **Instance:** a sequence of numbers with operations $+, -, \times$ in between, for example

$$1 + 2 - 3 \times 6 - 1 - 2 \times 3 - 5 \times 7 + 2 - 8 \times 9$$

Dynamic Programming: Maximizing an expression

- **Instance:** a sequence of numbers with operations $+, -, \times$ in between, for example

$$1 + 2 - 3 \times 6 - 1 - 2 \times 3 - 5 \times 7 + 2 - 8 \times 9$$

- **Task:** Place brackets in a way that the resulting expression has the largest possible value.

Dynamic Programming: Maximizing an expression

- **Instance:** a sequence of numbers with operations $+, -, \times$ in between, for example

$$1 + 2 - 3 \times 6 - 1 - 2 \times 3 - 5 \times 7 + 2 - 8 \times 9$$

- **Task:** Place brackets in a way that the resulting expression has the largest possible value.

- What will be the subproblems?

Dynamic Programming: Maximizing an expression

- **Instance:** a sequence of numbers with operations $+, -, \times$ in between, for example

$$1 + 2 - 3 \times 6 - 1 - 2 \times 3 - 5 \times 7 + 2 - 8 \times 9$$

- **Task:** Place brackets in a way that the resulting expression has the largest possible value.

- What will be the subproblems?

- Maybe for a subsequence of numbers $A[i..j]$ place the brackets so that the resulting expression is maximised?

Dynamic Programming: Maximizing an expression

- **Instance:** a sequence of numbers with operations $+, -, \times$ in between, for example

$$1 + 2 - 3 \times 6 - 1 - 2 \times 3 - 5 \times 7 + 2 - 8 \times 9$$

- **Task:** Place brackets in a way that the resulting expression has the largest possible value.
- What will be the subproblems?
- Maybe for a subsequence of numbers $A[i..j]$ place the brackets so that the resulting expression is maximised?
- maybe we could consider which the principal operations should be, i.e. $A[i..k] \odot A[k+1..j]$. Here $\odot$ is whatever operation is between $A[k]$ and $A[k+1]$.

Dynamic Programming: Maximizing an expression

- **Instance:** a sequence of numbers with operations $+, -, \times$ in between, for example

$$1 + 2 - 3 \times 6 - 1 - 2 \times 3 - 5 \times 7 + 2 - 8 \times 9$$

- **Task:** Place brackets in a way that the resulting expression has the largest possible value.

- What will be the subproblems?

- Maybe for a subsequence of numbers $A[i..j]$ place the brackets so that the resulting expression is maximised?

- maybe we could consider which the principal operations should be, i.e. $A[i..k] \odot A[k+1..j]$. Here $\odot$ is whatever operation is between $A[k]$ and $A[k+1]$.

- But when would such expression be maximised if there could be both positive and negative values for $A[i..k]$ and $A[k+1..j]$ depending on the placement of brackets??

Dynamic Programming: Maximizing an expression

- **Instance:** a sequence of numbers with operations $+, -, \times$ in between, for example

$$1 + 2 - 3 \times 6 - 1 - 2 \times 3 - 5 \times 7 + 2 - 8 \times 9$$

- **Task:** Place brackets in a way that the resulting expression has the largest possible value.
- What will be the subproblems?
- Maybe for a subsequence of numbers $A[i..j]$ place the brackets so that the resulting expression is maximised?
- maybe we could consider which the principal operations should be, i.e. $A[i..k] \odot A[k+1..j]$. Here $\odot$ is whatever operation is between $A[k]$ and $A[k+1]$.
- But when would such expression be maximised if there could be both positive and negative values for $A[i..k]$ and $A[k+1..j]$ depending on the placement of brackets??
- Maybe we should look for placements of brackets not only for the maximal value but also for the minimal value!

Dynamic Programming: Maximizing an expression

- **Instance:** a sequence of numbers with operations $+, -, \times$ in between, for example

$$1 + 2 - 3 \times 6 - 1 - 2 \times 3 - 5 \times 7 + 2 - 8 \times 9$$

- **Task:** Place brackets in a way that the resulting expression has the largest possible value.
- What will be the subproblems?
- Maybe for a subsequence of numbers $A[i..j]$ place the brackets so that the resulting expression is maximised?
- maybe we could consider which the principal operations should be, i.e. $A[i..k] \odot A[k+1..j]$. Here $\odot$ is whatever operation is between $A[k]$ and $A[k+1]$.
- But when would such expression be maximised if there could be both positive and negative values for $A[i..k]$ and $A[k+1..j]$ depending on the placement of brackets??
- Maybe we should look for placements of brackets not only for the maximal value but also for the minimal value!
- Exercise: write the exact recursion for this problem.

Dynamic Programming: Turtle Tower

- **Instance:** You are given n turtles, and for each turtle you are given its weight and its strength. The strength of a turtle is the maximal weight you can put on it without cracking its shell.

Dynamic Programming: Turtle Tower

- **Instance:** You are given n turtles, and for each turtle you are given its weight and its strength. The strength of a turtle is the maximal weight you can put on it without cracking its shell.

- **Task:** Find the largest possible number of turtles which you can stack one on top of the other, without cracking any turtle.

Dynamic Programming: Turtle Tower

- **Instance:** You are given n turtles, and for each turtle you are given its weight and its strength. The strength of a turtle is the maximal weight you can put on it without cracking its shell.
- **Task:** Find the largest possible number of turtles which you can stack one on top of the other, without cracking any turtle.
- **Hint:** Order turtles in an increasing order of the sum of their weight and their strength, and proceed by recursion. You might want to first solve the longest increasing subsequence of numbers problem by a solution which runs in time $n \log n$ because both problems use similar tricks...