



Assignment Project Exam Help

<https://eduassistpro.github.io>

Add WeChat [edu_assist_pro](#)

Aleks Ignjatov
School of Computer Science and Engineering
University of New South Wales

DYNAMIC PROGRAMMING

Assignment Project Exam Help

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

- Sub
opti
size s

<https://eduassistpro.github.io>

- Efficiency of DP comes from the fact that the sets of subproblems to solve for a problem having a particular size are solved once and the solution is stored in a table for multiple calls to the same larger problems.

Add WeChat edu_assist_pro

Assignment Project Exam Help

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

- Sub
opti
size s

<https://eduassistpro.github.io>

- Efficiency of DP comes from the fact that the sets of subproblems to solve have overlapping subproblems. Once a solution is stored in a table for multiple subproblems, it can be reused for larger problems.

Add WeChat edu_assist_pro

Assignment Project Exam Help

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

- Sub
opti
size s

- Efficiency of DP comes from the fact that the sets of subproblems to solve larger problems heavily overlap. Each subproblem is solved once and its solution is stored in a table for multiple use in solving larger problems.

Add WeChat [edu_assist_pro](https://eduassistpro.github.io)

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.

- We start with a sequence of non-overlapping activities σ_i .

- For every $i \leq n$ we solve the following subproblem

Subproblem $\tau^*(i)$: find a subsequence τ_i

$S_i = \langle a_1, s_2, \dots, a_i \rangle$ 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.

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.

Assignment Project Exam Help

- **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.

- We start with a sequence of non-overlapping activities

<https://eduassistpro.github.io>

- For every $i \leq n$ we solve the following subproblem

Subproblem $\sigma^*(i)$: find a subsequence σ_i

$S_i = \langle a_1, s_2, \dots, a_i \rangle$ 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.

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.

- We start with a sequence of non-overlapping activities $\sigma_0 = \langle a_1, s_2, \dots, a_i \rangle$.
- For every $i \leq n$ we solve the following subproblem

Subproblem $\sigma_i^*(i)$: find a subsequence σ_i of $S_i = \langle a_1, s_2, \dots, a_i \rangle$ 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.

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.

- We see that this is a non-overlapping activities problem.

- For every $i \leq n$ we solve the following subproblem

Subproblem $\sigma^*(i)$: find a subsequence σ_i

$S_i = \langle a_1, s_2, \dots, a_i \rangle$ 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.

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.

- We solve the subproblem of finding a non-overlapping subsequence σ_i of $S_i = \langle a_1, a_2, \dots, a_i \rangle$ 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.

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.

- We start with the subproblem of finding a non-overlapping subsequence σ_i of $S_i = \langle a_1, a_2, \dots, a_i \rangle$ 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.

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.

- We solve the subproblem of finding a non-overlapping subsequence σ_i of $S_i = \langle a_1, a_2, \dots, a_i \rangle$ 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.

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.

- We start with a sequence of non-overlapping activities.

- For every $i \leq n$ we solve the following subproblem

Subproblem $P(i)$: find a subsequence σ_i

$S_i = \langle a_1, a_2, \dots, a_i \rangle$ 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.

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.

- We start with the subproblem of finding a non-overlapping subsequence of a_1, a_2, \dots, a_n .
- For every $i \leq n$ we solve the following subproblem:
Subproblem $P(i)$: find a subsequence σ_i of $S_i = \langle a_1, a_2, \dots, a_i \rangle$ 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.

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 $T(1) = f_1 - s_1$;

Assignment Project Exam Help

<https://eduassistpro.github.io>



- In the table, for every i , besides $T(i)$, we also store $\pi(i) = j$ for which the above max is achieved:

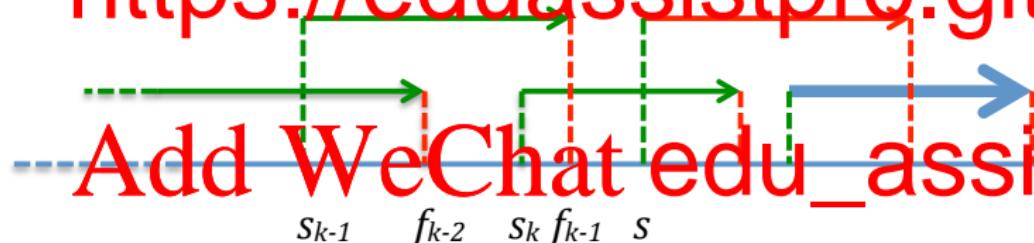
$$\pi(i) = \arg \max\{T(j) : j < i \text{ } \& \text{ } f_j \leq s_i\}$$

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 $T(1) = f_1 - s_1$;

Assignment Project Exam Help

<https://eduassistpro.github.io>



Add WeChat edu_assist_pro

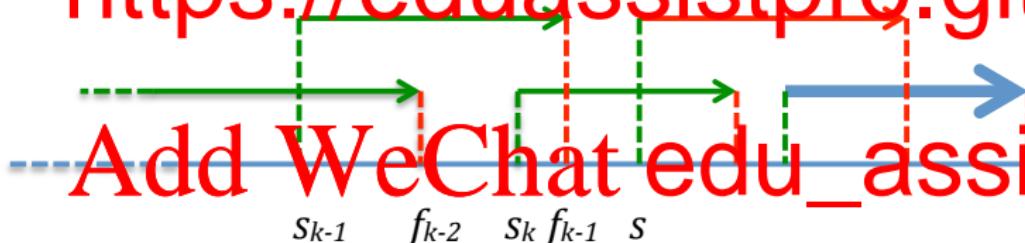
- In the table, for every i , besides $T(i)$, we also store $\pi(i) = j$ for which the above max is achieved:

$$\pi(i) = \arg \max\{T(j) : j < i \text{ } \& \text{ } f_j \leq s_i\}$$

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

<https://eduassistpro.github.io>



Add WeChat [edu_assist_pro](https://eduassistpro.github.io)

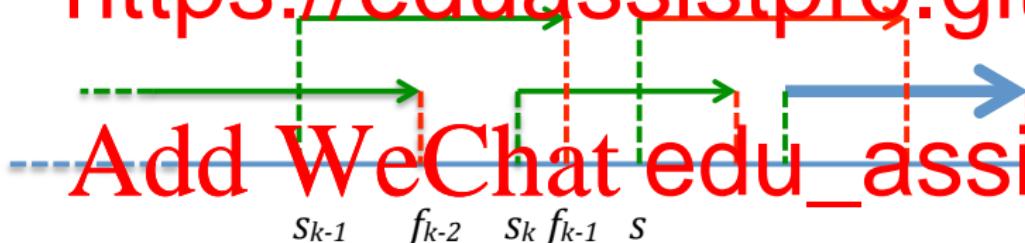
- In the table, for every i , besides $T(i)$, we also store $\pi(i) = j$ for which the above max is achieved:

$$\pi(i) = \arg \max\{T(j) : j < i \text{ } \& \text{ } f_j \leq s_i\}$$

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

<https://eduassistpro.github.io>



Add WeChat [edu_assist_pro](https://eduassistpro.github.io)

- In the table, for every i , besides $T(i)$, we also store $\pi(i) = j$ for which the above max is achieved:

$$\pi(i) = \arg \max\{T(j) : j < i \text{ } \& \text{ } f_j \leq s_i\}$$

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

Assignment Project Exam Help

- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ is an optimal solution to $P(a_{k_m})$.

• What is the optimal solution to <https://eduassistpro.github.io>?

- If there were a sequence S^* of a larger total duration than S , then S^* would also end in a_{k_m} .

• By extending the sequence S^* by a_{k_m} , we would obtain an optimal solution to subproblem $P(a_{k_m})$ with a longer total duration than the optimal solution to subproblem $P(i)$.

• This contradicts the optimality of S , so S must be optimal.

- Thus, the optimal solution $S = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}}, a_{k_m})$ for problem $P(i)$ ($= P(a_{k_m})$) is obtained from the optimal solution $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ for problem $P(a_{k_{m-1}})$ by extending it with a_{k_m} .

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

$$S = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}}, a_{k_m}) \text{ where } k_m = i;$$

Assignment Project Exam Help

- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ is an optimal solution

• When we add <https://eduassistpro.github.io> to the optimal solution S' , we get an optimal solution S for problem $P(i)$.

- If there were a sequence S^* of a larger total duration than S that was also an optimal solution for problem $P(i)$, then we could obtain a sequence S^* with the same total duration as S by extending the sequence S^* by one activity, namely a_{k_m} . This contradicts the optimality of the optimal solution S for problem $P(i)$ with a longer total duration than the optimal solution S , contradicting the optimality of S .

- Thus, the optimal solution $S = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}}, a_{k_m})$ for problem $P(i)$ ($= P(a_{k_m})$) is obtained from the optimal solution $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ for problem $P(a_{k_{m-1}})$ by extending it with a_{k_m} .

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 $S = (a_{k_1}, c_{k_1}, \dots, a_{k_{m-1}}, a_{k_m})$ where $k_m = i$.
- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ is an optimal solution to $P(a_{k_{m-1}})$.

Assignment Project Exam Help

- Why does such a recursion produce optimal solutions to subproblems $P(i)$?
- Let the optimal solution of subproblem $P(i)$ be the sequence $S = (a_{k_1}, c_{k_1}, \dots, a_{k_{m-1}}, a_{k_m})$ where $k_m = i$.
- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ is an optimal solution to $P(a_{k_{m-1}})$.
- If there were a sequence S^* of a larger total duration than S that was also feasible with activity a_i , then we could obtain a sequence S^* with a smaller total duration by extending the sequence S' by activity a_i . This contradicts the optimality of the subproblem $P(i)$ with a longer total duration than the total duration of the sequence S , contradicting the optimality of S .
- Thus, the optimal solution $S = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}}, a_{k_m})$ for problem $P(i)$ ($= P(a_{k_m})$) is obtained from the optimal solution $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ for problem $P(a_{k_{m-1}})$ by extending it with a_{k_m} .

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 $S = (a_{k_1}, c_{k_1}, \dots, a_{k_{m-1}}, a_{k_m})$ where $k_m = i$.
- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ is an optimal solution to $P(a_{k_{m-1}})$.

- Why? If there were a sequence S^* of a larger total duration than S , then S^* would also end in activity a_i .
by extending the sequence S^* by activity a_i , we get an optimal solution to subproblem $P(i)$ with a longer total duration than the total duration of sequence S , contradicting the optimality of S .
- Thus, the optimal solution $S = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}}, a_{k_m})$ for problem $P(i)$ ($= P(a_{k_m})$) is obtained from the optimal solution $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ for problem $P(a_{k_{m-1}})$ by extending it with a_{k_m} .

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 $S = (a_{k_1}, c_{k_1}, \dots, a_{k_{m-1}}, a_{k_m})$ where $k_m = i$.
- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ is an optimal solution to $P(a_{k_{m-1}})$.

- Why?
 - If there were a sequence S^* of a larger total duration than S , then S^* would also end in activity a_i .
by extending the sequence S^* by activity a_i , we get an optimal solution to subproblem $P(i)$ with a longer total duration than the total duration of sequence S , contradicting the optimality of S .
 - Thus, the optimal solution $S = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}}, a_{k_m})$ for problem $P(i)$ ($= P(a_{k_m})$) is obtained from the optimal solution $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ for problem $P(a_{k_{m-1}})$ by extending it with a_{k_m} .

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

$S = (a_{k_1}, c_{k_1}, \dots, c_{k_{m-1}}, a_{k_m})$ where $k_m = i$;

Assignment Project Exam Help

- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ is an optimal solution

- What is the optimal solution for the problem <https://eduassistpro.github.io>?

- If there were a sequence S^* of a larger total duration than the sequence S' and also ending with activity a_{k_m} , then by extending the sequence S' with activity a_{k_m} we would obtain a subproblem $P(i)$ with a longer total duration than the subproblem $P(i)$ obtained from the sequence S , contradicting the optimality of S .

- Thus, the optimal solution $S = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}}, a_{k_m})$ for problem $P(i)$ ($= P(a_{k_m})$) is obtained from the optimal solution $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ for problem $P(a_{k_{m-1}})$ by extending it with a_{k_m} .

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

$S = (a_{k_1}, c_{k_1}, \dots, c_{k_{m-1}}, a_{k_m})$ where $k_m = i$;

Assignment Project Exam Help

- We claim: the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ is an optimal solution

- What is the optimal solution for the problem <https://eduassistpro.github.io>?

- If there were a sequence S^* of a larger total duration than the sequence S' and also ending with activity a_{k_m} , then by extending the sequence S' with activity a_{k_m} we would obtain a subproblem $P(i)$ with a longer total duration than the subproblem $P(i)$ obtained from the sequence S , contradicting the optimality of S .

- Thus, the optimal solution $S = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}}, a_{k_m})$ for problem $P(i)$ ($= P(a_{k_m})$) is obtained from the optimal solution $S' = (a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}})$ for problem $P(a_{k_{m-1}})$ by extending it with a_{k_m} .

Dynamic Programming: Activity Selection

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

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

$$\text{last} = \arg \max\{T(i) : i \leq n\}.$$

The table of partial solutions, because in the i^{th} slot of the table, besides $T(i)$ we also store $\pi(i) = j$, ($j < i$) such that the optimal solution of $P(i)$ extends the o

- Thu <https://eduassistpro.github.io>
- Why is such solution optimal, i.e., why looking for optimal solutions of $P(i)$ which must end with a_i did not cause us to miss th such an additional requirement?
- Consider the optimal solution without such addition assume it ends with activity a_k ; then 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 solve n subproblems $P(i)$ for solutions ending in a_i ; for each such interval a_i we have to find all preceding compatible intervals and their optimal solutions (to be looked up in a table). This $T(n) \subseteq O(n^2)$

Dynamic Programming: Activity Selection

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

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

$$\text{last} = \arg \max\{T(i) : i \leq n\}.$$

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 $\pi(i) = j$, ($j < i$) such that the optimal solution of $P(i)$ extends the o

- Thu <https://eduassistpro.github.io>
- Why is such solution optimal, i.e., why looking for optimal solutions of $P(i)$ which must end with a_i did not cause us to miss th such an additional requirement?
- Consider the optimal solution without such addition assume it ends with activity a_k ; then 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 solve n subproblems $P(i)$ for solutions ending in a_i ; for each such interval a_i we have to find all preceding compatible intervals and their optimal solutions (to be looked up in a table). This $T(n) \subseteq O(n^2)$

Dynamic Programming: Activity Selection

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

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

$$\text{last} = \arg \max\{T(i) : i \leq n\}.$$

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 $\pi(i) = j$, ($j < i$) such that the optimal solution of $P(i)$ extends the one of $P(j)$.

- Thu <https://eduassistpro.github.io/lectures/dp/activity-selection.html>
- Why is such solution optimal, i.e., why looking for optimal solutions of $P(i)$ which must end with a_i did not cause us to miss the possibility of including such an additional requirement?
- Consider the optimal solution without such additional requirement. Assume it ends with activity a_k ; then 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 solve n subproblems $P(i)$ for solutions ending in a_i ; for each such interval a_i we have to find all preceding compatible intervals and their optimal solutions (to be looked up in a table). Thus, $T(n) = O(n^2)$.

Dynamic Programming: Activity Selection

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

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

$$\text{last} = \arg \max\{T(i) : i \leq n\}.$$

- 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 $\pi(i) = j$, ($j < i$) such that the optimal solution of $P(i)$ extends the o

- Thu <https://eduassistpro.github.io/DP/ActivitySelection.html>
- Why is such solution optimal, i.e., why looking for optimal solutions of $P(i)$ which must end with a_i did not cause us to miss the such an additional requirement?
- Consider the optimal solution without such addition assume it ends with activity a_k ; then 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 solve n subproblems $P(i)$ for solutions ending in a_i ; for each such interval a_i we have to find all preceding compatible intervals and their optimal solutions (to be looked up in a table). This $T(n) \subseteq \Theta(n^2)$

Dynamic Programming: Activity Selection

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

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

$$\text{last} = \arg \max\{T(i) : i \leq n\}.$$

- 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 $\pi(i) = j$, ($j < i$) such that the optimal solution of $P(i)$ extends the o

- Thu <https://eduassistpro.github.io/CS101/lec05.html#ActivitySelection>
- Why is such solution optimal, i.e., why looking for optimal solutions of $P(i)$ which must end with a_i did not cause us to miss the such an additional requirement?
- Consider the optimal solution without such addition assume it ends with activity a_k ; then 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 solve n subproblems $P(i)$ for solutions ending in a_i ; for each such interval a_i we have to find all preceding compatible intervals and their optimal solutions (to be looked up in a table). This $T(n) = O(n^2)$

Dynamic Programming: Activity Selection

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

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

$$\text{last} = \arg \max\{T(i) : i \leq n\}.$$

- 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 $\pi(i) = j$, ($j < i$) such that the optimal solution of $P(i)$ extends the o

- Thu <https://eduassistpro.github.io/AssignmentProjectExamHelp/>
- Why is such solution optimal, i.e., why looking for optimal solutions of $P(i)$ which must end with a_i did not cause us to miss the such an additional requirement?
- Consider the optimal solution without such addition assume it ends with activity a_k ; then 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 solve n subproblems $P(i)$ for solutions ending in a_i ; for each such interval a_i we have to find all preceding compatible intervals and their optimal solutions (to be looked up in a table). Thus, $T(n) = \Theta(n^2)$.

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.

Assignment Project Exam Help

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

- *Submax_{end}*

<https://eduassistpro.github.io>

- Recursion: Assume we have solved the subproblem *Submax_{end}* for all $j < i$. We want to solve *Submax_{end}* for i .

Add WeChat `edu_assist_pro` to get help.

- 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.

Assignment Project Exam Help

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

- *Sub*
max
end

<https://eduassistpro.github.io>

- Recursion: Assume we have solved the subproblem of finding the longest increasing subsequence ending at index j .

Add WeChat `edu_assist_pro` to get help.

- 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.

Assignment Project Exam Help

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

- $\begin{matrix} \text{Sub} \\ \max \\ \text{end} \end{matrix}$

<https://eduassistpro.github.io>

- Recursion: Assume we have solved the subproblem that we have just obtained. The values

nd
that we have just obtained. The values
 ℓ_j of the longest increasing subsequence which ends with $A[i]$.

- 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.

Assignment Project Exam Help

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

- $\begin{matrix} \text{Sub} \\ \max \\ \text{end} \end{matrix}$

<https://eduassistpro.github.io>

- Recursion: Assume we have solved the subproblem that we have put in a table S the values ℓ_j of maximal increasing sequences which end with $A[i]$.
Add WeChat edu_assist_pro
- 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.

Assignment Project Exam Help

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

- $\begin{matrix} \text{Sub} \\ \max \\ \text{end} \end{matrix}$

<https://eduassistpro.github.io>

- Recursion: Assume we have solved the subproblem that we have put in a table S the values ℓ_j of maximal increasing sequences which end with $A[i]$.
- 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.

Assignment Project Exam Help

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

- $\begin{matrix} \text{Sub} \\ \max \\ \text{end} \end{matrix}$ $\text{https://eduassistpro.github.io/}$ of $[i]$

- Recursion: Assume we have solved the subproblem that we have put in a table S the values ℓ_j of maximal increasing sequences which end with $A[j]$.
Add WeChat edu_assist_pro
- 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

$$\ell_i = \max\{\ell_m : m < i \text{ & } A[m] < A[i]\} + 1$$

$$\pi(i) = \arg \max\{\ell_m : m < i \text{ & } A[m] < A[i]\}$$

Assignment Project Exam Help

- We store in the i^{th} slot of the table the length ℓ_i of the longest increasing subsequence ending with $A[i]$ and $\pi(i) = m$ such that the optimal solution

- So, we can easily get the sequence by backtracking.

- Finally, from all such subsequences we pick the longest one.

Add WeChat {edu_assist_pro}

- The end point of such a sequence can be obtained as

$$\text{end} = \arg \max\{\ell_i : i \leq n\}$$

More Dynamic Programming Problems

$$\ell_i = \max\{\ell_m : m < i \text{ \& } A[m] < A[i]\} + 1$$

$$\pi(i) = \arg \max\{\ell_m : m < i \text{ \& } A[m] < A[i]\}$$

Assignment Project Exam Help

- We store in the i^{th} slot of the table the length ℓ_i of the longest increasing subsequence ending with $A[i]$ and $\pi(i) = m$ such that the optimal solution

- So, we can find the solution by using the above formula.

- Finally, from all such subsequences we pick the longest one.

Add WeChat {edu_assist_pro}

- The end point of such a sequence can be obtained as

$$\text{end} = \arg \max\{\ell_i : i \leq n\}$$

More Dynamic Programming Problems

$$\ell_i = \max\{\ell_m : m < i \text{ \& } A[m] < A[i]\} + 1$$

$$\pi(i) = \arg \max\{\ell_m : m < i \text{ \& } A[m] < A[i]\}$$

Assignment Project Exam Help

- We store in the i^{th} slot of the table the length ℓ_i of the longest increasing subsequence ending with $A[i]$ and $\pi(i) = m$ such that the optimal solution

- So, we can find the end point of the sequence by traversing the table from right to left.

- Finally, from all such subsequences we pick the longest one.

Add WeChat {edu_assist_pro}

- The end point of such a sequence can be obtained as

$$\text{end} = \arg \max\{\ell_i : i \leq n\}$$

More Dynamic Programming Problems

$$\ell_i = \max\{\ell_m : m < i \text{ \& } A[m] < A[i]\} + 1$$

$$\pi(i) = \arg \max\{\ell_m : m < i \text{ \& } A[m] < A[i]\}$$

Assignment Project Exam Help

- We store in the i^{th} slot of the table the length ℓ_i of the longest increasing subsequence ending with $A[i]$ and $\pi(i) = m$ such that the optimal solution

- So, we can find the end of the sequence by looking at the last element of the table.

- Finally, from all such subsequences we pick the longest one.

Add WeChat `edu_assist_pro`

- The end point of such a sequence can be obtained as

$$\text{end} = \arg \max\{\ell_i : i \leq n\}$$

More Dynamic Programming Problems

$$\ell_i = \max\{\ell_m : m < i \text{ \& } A[m] < A[i]\} + 1$$

$$\pi(i) = \arg \max\{\ell_m : m < i \text{ \& } A[m] < A[i]\}$$

Assignment Project Exam Help

- We store in the i^{th} slot of the table the length ℓ_i of the longest increasing subsequence ending with $A[i]$ and $\pi(i) = m$ such that the optimal solution

- So, we can find the end of the sequence by looking at the last element of the table.

- Finally, from all such subsequences we pick the longest one.

Add WeChat `edu_assist_pro`

- The end point of such a sequence can be obtained as

$$\text{end} = \arg \max\{\ell_i : i \leq n\}$$

• We can now reconstruct the longest monotonically increasing sequence by backtracking (thus getting it in the reverse order):

$\text{end}, \pi(\text{end}), \pi(\pi(\text{end})), \dots$

- Assignment Project Exam Help
<https://eduassistpro.github.io>
- Add WeChat edu_assist_pro
- Time complexity: $O(n^2)$
- Exercise: (somewhat tough, but very useful) Design an algorithm for solving this problem which runs in time $n \log n$.

- We can now reconstruct the longest monotonically increasing sequence by backtracking (thus getting it in the reverse order):

$\text{end}, \pi(\text{end}), \pi(\pi(\text{end})), \dots$

- Again, we 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 $n \log n$.

- We can now reconstruct the longest monotonically increasing sequence by backtracking (thus getting it in the reverse order):

$\text{end}, \pi(\text{end}), \pi(\pi(\text{end})), \dots$

- Again, we 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 $n \log n$.

- We can now reconstruct the longest monotonically increasing sequence by backtracking (thus getting it in the reverse order):

$\text{end}, \pi(\text{end}), \pi(\pi(\text{end})), \dots$

- Again, we have been constructed as the solution for $P(m)$.
- Time complexity: $\mathcal{O}(n^2)$.
- Exercise: (somewhat tough, but very useful) Design an algorithm for solving this problem which runs in time $n \log n$.

More Dynamic Programming Problems

- **Making Change.** You are given n types of coin denominations of values $v(1) < v(2) < \dots < v(n)$ (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 den

- <https://eduassistpro.github.io>
- Solution: Consider the problem of finding the minimum number of coins required to make change for amount C . We can use dynamic programming to solve this problem. Let $\text{min_coins}(C)$ be the minimum number of coins required to make change for amount C . Then, we can define the following recurrence relation:
$$\text{min_coins}(C) = \min_{1 \leq i \leq n} (\text{min_coins}(C - v(i)) + 1)$$
where $v(i)$ is the value of the i -th coin denomination. This recurrence relation states that the minimum number of coins required to make change for amount C is the minimum of the number of coins required to make change for amounts $C - v(1), C - v(2), \dots, C - v(n)$, plus one coin for the i -th coin.
slot i .
 - If $C = 0$, the solution is trivial: no coins are needed.
 - 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

- **Making Change.** You are given n types of coin denominations of values $v(1) < v(2) < \dots < v(n)$ (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 den

- Solution: <https://eduassistpro.github.io> stored in slot i .

- If $C = 0$, the solution is trivial: no coins. If $C > 0$ and $v(1) = 1$,
- Assume we have found optimal solutions for every amount $j < i$ and now want to find an optimal solution for amount i .

- **Making Change.** You are given n types of coin denominations of values $v(1) < v(2) < \dots < v(n)$ (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 den

- Solution: Consider the amount C 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 .

- **Making Change.** You are given n types of coin denominations of values $v(1) < v(2) < \dots < v(n)$ (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 den

- Solution: Consider a slot i containing C coins. If we can fill slot i with C coins, then we can fill slot $i-1$ with $C - v(i)$ coins. This leads to the following recurrence relation:
$$f(i, C) = \min_{k=1}^n f(i-1, C - k)$$
where $f(i, C)$ is the minimum number of coins needed to fill slot i with C coins. The base case is $f(0, 0) = 0$.
Add WeChat edu_assist_pro
- 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 $\text{opt}(i - v(k))$ 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.)

Assignment Project Exam Help

- 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 t

- We can now write a recurrence relation for $\text{opt}(i)$:
$$\text{opt}(i) = \min\{\text{opt}(i - v(k)) + 1 \mid k = 1, \dots, n\}$$

$$\text{opt}(i) = \min\{\text{opt}(i - v(k)) + 1 \mid k = 1, \dots, n\}$$

Add WeChat edu_assist_pro

- Why does this produce an optimal solution for amount i ?
- 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 $\text{opt}(i - v(k))$ 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.)

Assignment Project Exam Help

- 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 t

<https://eduassistpro.github.io>

$$\text{opt}(i) = \min\{\text{opt}(i - v(k)) + 1 \mid v(k) \in \{v(1), \dots, v(n)\}\}$$

Add WeChat edu_assist_pro

- Why does this produce an optimal solution for amount i ?
- 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 $\text{opt}(i - v(k))$ 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.)

Assignment Project Exam Help

- 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 t

- We can now write $\text{opt}(i) = \min\{\text{opt}(i - v(k)) + 1 \mid k \in \{1, \dots, n\}\}$

$$\text{opt}(i) = \min\{\text{opt}(i - v(k)) + 1 \mid k \in \{1, \dots, n\}\}$$

Add WeChat edu_assist_pro

- Why does this produce an optimal solution for amount i ?
- 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 $\text{opt}(i - v(k))$ 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.)

Assignment Project Exam Help

- 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 t

- We can now write the recurrence relation for $\text{opt}(i)$:

$$\text{opt}(i) = \min\{\text{opt}(i - v(k)) + 1 \mid k = 1, \dots, n\}$$

Add WeChat edu_assist_pro

- Why does this produce an optimal solution for amount i ?
- 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 $\text{opt}(i - v(k))$ 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.)

Assignment Project Exam Help

- 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 t

- We can now write $\text{opt}(i) = \min\{\text{opt}(i - v(k)) + 1 \mid k \in \{1, \dots, n\}\}$

$$\text{opt}(i) = \min\{\text{opt}(i - v(k)) + 1 \mid k \in \{1, \dots, n\}\}$$

Add WeChat edu_assist_pro

- Why does this produce an optimal solution for amount i ?
- 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 $\text{opt}(i - v(k))$ 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.)

Assignment Project Exam Help

- 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 t

- We can now write $\text{opt}(i) = \min\{\text{opt}(i - v(k)) + 1 \mid v(k) \in \{v(1), \dots, v(n)\}\}$

$$\text{opt}(i) = \min\{\text{opt}(i - v(k)) + 1 \mid v(k) \in \{v(1), \dots, v(n)\}\}$$

Add WeChat edu_assist_pro

- Why does this produce an optimal solution for amount i ?
- 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

- 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.

Assignment Project Exam Help

- 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 solution by looking at the previous slot, $i-1$, and the value stored in the i^{th} slot.

<https://eduassistpro.github.io>

- $opt(C)$ is the solution we need.

Time complexity of our algorithm is nC .
Add WeChat edu_assist_pro

- Note: Our algorithm is NOT a polynomial algorithm. The running time is nC , where C is the length of the input, because the length of a representation of C is only $\log C$, while the length of the output is nC .
- But this is the best what we can do...

More Dynamic Programming Problems

- 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 $i - v_i(m)$ uses the fewest number of coins.

Assignment Project Exam Help

- It is enough to store in the i^{th} slot of the table such m and $opt(i)$ because this allows us to find the minimum number of coins needed in the i^{th} slot, stored in the i^{th} slot.

<https://eduassistpro.github.io>

- $opt(C)$ is the solution we need.

Time complexity of our algorithm is nC .
Add WeChat edu_assist_pro

- Note: Our algorithm is NOT a polynomial algorithm. The running time is nC , where C is the length of the input, because the length of a representation of C is only $\log C$, while the length of the output is nC .
- But this is the best what we can do...

More Dynamic Programming Problems

- 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 $i - v_i(m)$ uses the fewest number of coins.

Assignment Project Exam Help

- It is enough to store in the i^{th} slot of the table such m and $opt(i)$ because this allows us to find the minimum number of coins needed in the i^{th} slot, stored in the i^{th} slot.
- $opt(C)$ is the solution we need.

<https://eduassistpro.github.io>
Add WeChat edu_assist_pro

- Note: Our algorithm is NOT a polynomial algorithm. The time complexity of our algorithm is nC , where n is 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

- 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 $i - v_i(m)$ uses the fewest number of coins.

Assignment Project Exam Help

- It is enough to store in the i^{th} slot of the table such m and $opt(i)$ because this allocation is stored in the i^{th} slot, the value is stored in the i^{th} slot.

<https://eduassistpro.github.io>

- $opt(C)$ is the solution we need.

- Time complexity of our algorithm is nC

Add WeChat edu_assist_pro

- Note: Our algorithm is NOT a polynomial algorithm. The running time is exponential in the length of the input, because the length of a representation of C is only $\log C$, while the length of the output is nC .

- But this is the best what we can do...

More Dynamic Programming Problems

- 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 $i - v_i(m)$ uses the fewest number of coins.

Assignment Project Exam Help

- It is enough to store in the i^{th} slot of the table such m and $opt(i)$ because this allows us to calculate the value stored in the i^{th} slot, $v_i(opt(i))$, by looking at the value stored in the i^{th} slot.

<https://eduassistpro.github.io>

- $opt(C)$ is the solution we need.

- Time complexity of our algorithm is nC

Add WeChat edu_assist_pro

- **Note:** Our algorithm is NOT a polynomial algorithm. The running time is nC , where C is the length of the input, because the length of a representation of C is only $\log C$, while the length of the output is nC .

- But this is the best what we can do...

More Dynamic Programming Problems

- 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 $i - v_i(m)$ uses the fewest number of coins.

Assignment Project Exam Help

- It is enough to store in the i^{th} slot of the table such m and $opt(i)$ because this allows us to calculate the value stored in the i^{th} slot, $v_i(opt(i))$, by looking at the value stored in the i^{th} slot.

<https://eduassistpro.github.io>

- $opt(C)$ is the solution we need.

- Time complexity of our algorithm is nC

Add WeChat edu_assist_pro

- **Note:** Our algorithm is NOT a polynomial algorithm. The running time is nC , where C is the length of the input, because the length of a representation of C is only $\log C$, while the length of the output 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.

Assignment Project Exam Help

- Solution

- We begin by defining

$\text{opt}(i, C)$ as the maximum value obtainable by filling a knapsack of capacity C using only items of kinds $1, 2, \dots, i$.

- Assume we have solved the problem for all knapsacks of capacities $j < i$.

- We now look at optimal solution $\text{opt}(i, C)$ for $i = u, u + 1, \dots, n$ and $0 \leq C \leq w_i$.

Add WeChat edu_assist_pro

- Chose the one for which $\text{opt}(i - w_m) + v_m$ 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

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.

Assignment Project Exam Help

- Solu

- We b

<https://eduassistpro.github.io>

- Assume we have solved the problem for all knapsacks of capacities $j < i$.

- We now look at optimal solution $\text{opt}(i)$

$$i - w_m \leq j \leq n$$

Add WeChat edu_assist_pro

- Chose the one for which $\text{opt}(i - w_m) + v_m$ 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

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.

Assignment Project Exam Help

- Solu

- We b

<https://eduassistpro.github.io>

- Assume we have solved the problem for all knapsacks of capacities $j < i$.

- We now look at optimal solution $\text{opt}(i)$

$$i - w_m \leq j \leq n$$

Add WeChat edu_assist_pro

- Chose the one for which $\text{opt}(i - w_m) + v_m$ 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

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.

Assignment Project Exam Help

- Solu

- We b
<https://eduassistpro.github.io>

- Assume we have solved the problem for all knapsacks of capacities $j < i$.

- We now look at optimal solution $\text{opt}(i - w_m, v_m)$ for $i - w_m \leq m \leq n$

Add WeChat edu_assist_pro

- Chose the one for which $\text{opt}(i - w_m) + v_m$ 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

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.

Assignment Project Exam Help

- Solu

- We begin by defining the optimal value of a knapsack of capacity $i \leq C$.

- Assume we have solved the problem for all knapsacks of capacities $j < i$.
- We now look at optimal solutions $opt(i - w_m) + v_m$ 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

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.

Assignment Project Exam Help

- Solu

- We b

<https://eduassistpro.github.io>

- 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 $1 \leq m \leq n$,
- 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

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.

Assignment Project Exam Help

- Solu

- We b

<https://eduassistpro.github.io>

- 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 $1 \leq m \leq n$,
- 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,

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

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

- After

<https://eduassistpro.github.io>

- Whi

backtracking: if $\pi(C) = k$ then the first object is

$opt(C)$.

alue v_k ;

if $\pi(C - w_k) = m$ then the second object is

Add WeChat edu_assist_p

- Note that $\pi(i)$ might not be uniquely determined; in good solutions we pick arbitrarily among them.

- Again, our algorithm is NOT polynomial in the length of the input.

More Dynamic Programming Problems

- Thus,

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

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

- After

<https://eduassistpro.github.io/dp/AssignmentProjectExamHelp.html>

- Whi

backtracking: if $\pi(C) = k$ then the first object is

if $\pi(C - w_k) = m$ then the second object is

$opt(C)$.

alue v_k ;

- Note that $\pi(i)$ might not be uniquely determined; in good solutions we pick arbitrarily among them.

Add WeChat edu_assist_pro

- Again, our algorithm is NOT polynomial in the length of the input.

More Dynamic Programming Problems

- Thus,

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

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

- After

<https://eduassistpro.github.io>^{opt(C)}.

- Whi

backtracking: if $\pi(C) = k$ then the first object is
if $\pi(C - w_k) = m$ then the second object is

alue v_k ;

- Note that $\pi(i)$ might not be uniquely determined; in good solutions we pick arbitrarily among them.

Add WeChat edu_assist_pro

- Again, our algorithm is NOT polynomial in the length of the input.

More Dynamic Programming Problems

- Thus,

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

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

- After

<https://eduassistpro.github.io>^{opt(C)}.

- Whi

backtracking: if $\pi(C) = k$ then the first object is
if $\pi(C - w_k) = m$ then the second object is

alue v_k ;

- Note that $\pi(i)$ might not be uniquely determined; in good solutions we pick arbitrarily among them.

Add WeChat edu_assist_pro

- Again, our algorithm is NOT polynomial in the length of the input.

More Dynamic Programming Problems

- Thus,

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

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

- After

<https://eduassistpro.github.io>^{opt(C)}.

- Whi

backtracking: if $\pi(C) = k$ then the first object is

if $\pi(C - w_k) = m$ then the second object is

alue v_k ;

- Note that $\pi(i)$ might not be uniquely determined; in good solutions we pick arbitrarily among them.

Add WeChat edu_assist_pro

- Again, our algorithm is **NOT** polynomial in the **length** of the input.

More Dynamic Programming Problems

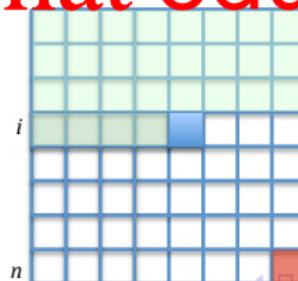
- **Integer Knapsack Problem (Duplicate Items NOT Allowed)** You have n items (some of which can be identical); item I_i is of weight w_i and value v_i . You also have a knapsack of capacity C . Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.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

chos
and s

<https://eduassistpro.github.io>

- 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 0 to c
 - ② for $i = j$ have solved the problem for all knapsacks of capacities from 0 to c

Add WeChat edu_assist_pro



More Dynamic Programming Problems

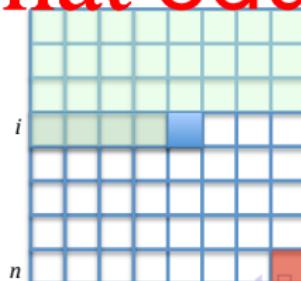
- **Integer Knapsack Problem (Duplicate Items NOT Allowed)** You have n items (some of which can be identical); item I_i is of weight w_i and value v_i . You also have a knapsack of capacity C . Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.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

choose
and is

<https://eduassistpro.github.io>

- 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 0 to c
 - ② for $i > j$ have solved the problem for knapsacks of capacities from 0 to $c - w_j$

Add WeChat edu_assist_pro



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:

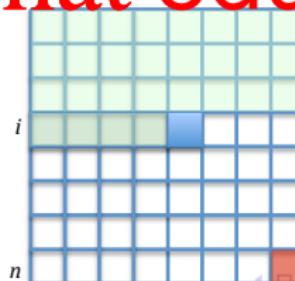
choose
and is

<https://eduassistpro.github.io>

- 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 0 to c have already been solved
- ② for $i = j$ have already been solved

Add WeChat edu_assist_pro



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.

Assignment Project Exam Help

- 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:

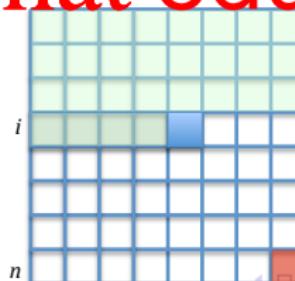
choose
and is

<https://eduassistpro.github.io>

- 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 0 to c
- ② if $i = n$ have solved the problem for $c = 0$

Add WeChat edu_assist_pro



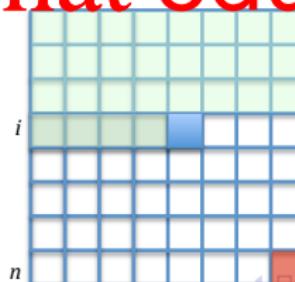
More Dynamic Programming Problems

- **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:

choose
and is

<https://eduassistpro.github.io>

- 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 0 to c
 - ② for i we have solved the problem for all capacities from 0 to C



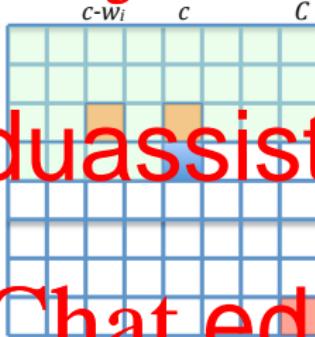
Add WeChat edu_assist_pro

More Dynamic Programming Problems

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

- so we look at optimal solutions $opt(i - 1, c - w_i)$ and $opt(i - 1, c)$:

Assignment Project Exam Help



<https://eduassistpro.github.io>

Add WeChat edu_assist_pro

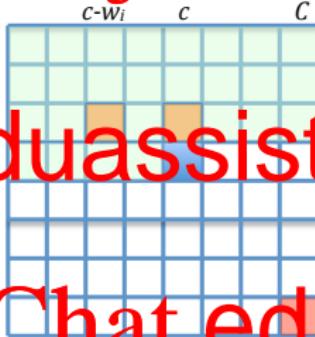
- if $opt(i - 1, c - w_i) + v_i > opt(i - 1, c)$
then $opt(i, c) = opt(i - 1, c - w_i) + v_i$;
else $opt(i, c) = opt(i - 1, c)$.
- Final solution will be given by $opt(n, C)$.

More Dynamic Programming Problems

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

- so we look at optimal solutions $opt(i - 1, c - w_i)$ and $opt(i - 1, c)$:

Assignment Project Exam Help



<https://eduassistpro.github.io>

Add WeChat edu_assist_pro

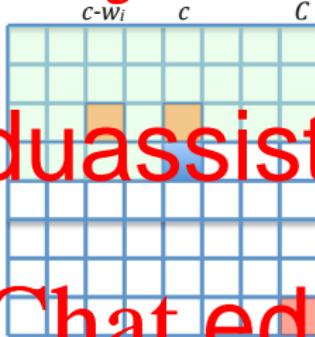
- if $opt(i - 1, c - w_i) + v_i > opt(i - 1, c)$
then $opt(i, c) = opt(i - 1, c - w_i) + v_i$;
else $opt(i, c) = opt(i - 1, c)$.
- Final solution will be given by $opt(n, C)$.

More Dynamic Programming Problems

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

- so we look at optimal solutions $opt(i - 1, c - w_i)$ and $opt(i - 1, c)$:

Assignment Project Exam Help



<https://eduassistpro.github.io>

Add WeChat edu_assist_pro

- 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 $opt(i, c) = opt(i - 1, c)$.

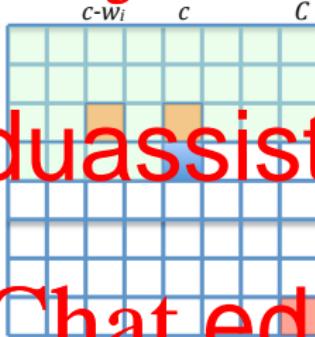
- Final solution will be given by $opt(n, C)$.

More Dynamic Programming Problems

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

- so we look at optimal solutions $opt(i - 1, c - w_i)$ and $opt(i - 1, c)$:

Assignment Project Exam Help



<https://eduassistpro.github.io>

Add WeChat edu_assist_pro

- 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.

Assignment Project Exam Help
solution. Let S be the total sum of integers in the set considered. This is equivalent to 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 .

- Classification:
partitions
into

<https://eduassistpro.github.io>

- Why? Since $S = S_1 + S_2$ we obtain

Add WeChat edu_assist_pro

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 $S/2$.

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.

Assignment Project Exam Help

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 .

- Cl
part
inte

<https://eduassistpro.github.io>

- Why? Since $S = S_1 + S_2$ we obtain

Add WeChat edu_assist_pro

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

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 $S/2$.

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.

Assignment Project Exam Help
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 .

- Class part
introduction

<https://eduassistpro.github.io>

- Why? Since $S = S_1 + S_2$ we obtain

Add WeChat edu_assist_pro

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 $S/2$.

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.

Assignment Project Exam Help

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 .

- Class part
integers
- Why? Since $S = S_1 + S_2$ we obtain

Add WeChat edu_assist_pro

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 $S/2$.

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.

Assignment Project Exam Help
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 .

- Classification:
part of the
integer
- Why? Since $S = S_1 + S_2$ we obtain

Add WeChat edu_assist_pro

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

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 $S/2$.

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.

Assignment Project Exam Help
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 .

- Classification
part of the
integer

- Why? Since $S = S_1 + S_2$ we obtain

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

Add WeChat edu_assist_pro

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 $S/2$.

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.

Assignment Project Exam Help
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 .

- Classification
part of the
integer

- Why? Since $S = S_1 + S_2$ we obtain

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

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 $S/2$.

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.

Assignment Project Exam Help
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 .

- Classification
part of the
integer

<https://eduassistpro.github.io>

- Why? Since $S = S_1 + S_2$ we obtain

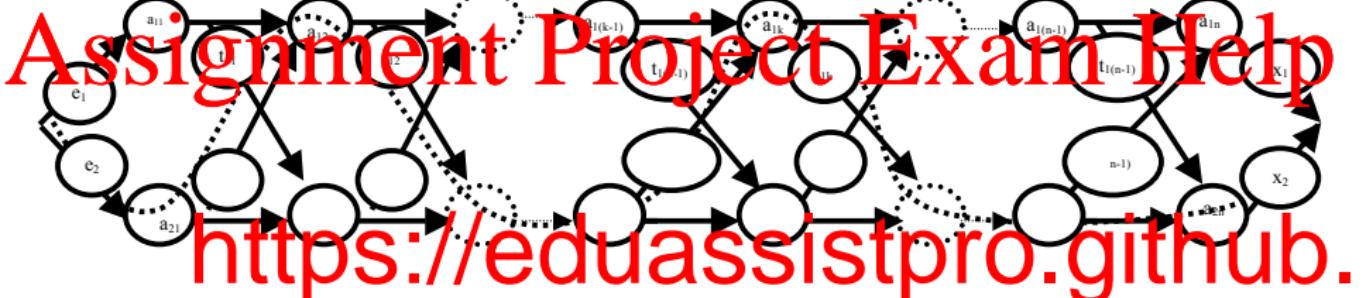
Add WeChat edu_assist_pro

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

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

- On the first assembly line the $t_{1,k}^{(t)}$ it takes to move a product from station $k - 1$ to station k is complete; on the second assembly line the same job takes $t_{2,k}$.
- 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.

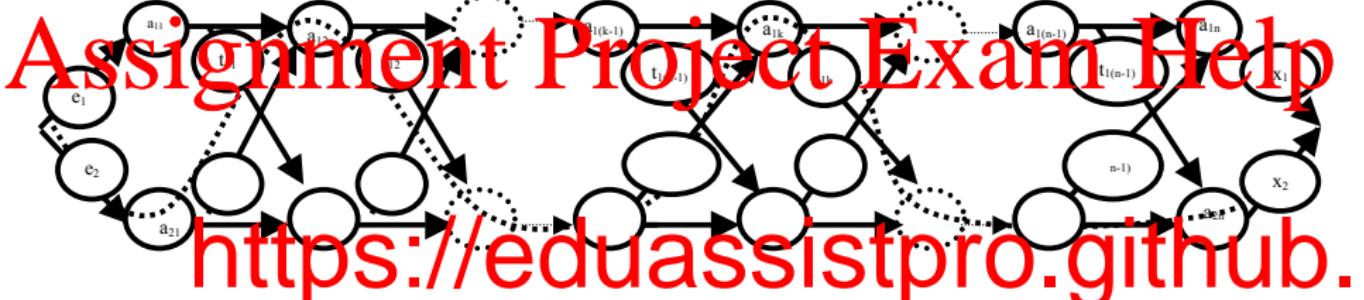
Dynamic Programming: Assembly line scheduling



Instance: Two assembly lines with workstations for

- On the first assembly line the k^{th} job takes $t_{1,k}$ units of time to complete; on the second assembly line the same job takes $t_{2,k}$ 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.

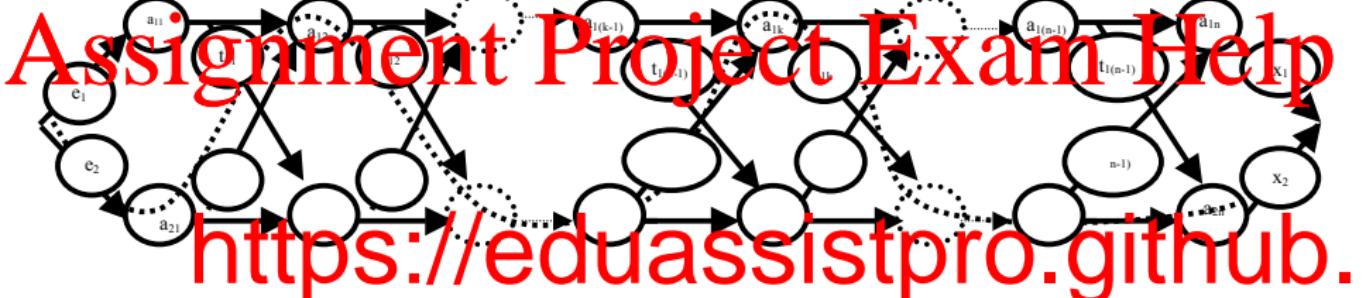
Dynamic Programming: Assembly line scheduling



Instance: Two assembly lines with workstations for

- On the first assembly line the k^{th} job takes $t_{1,k}$ units of time to complete; on the second assembly line the same job takes $t_{2,k}$ 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.

Dynamic Programming: Assembly line scheduling

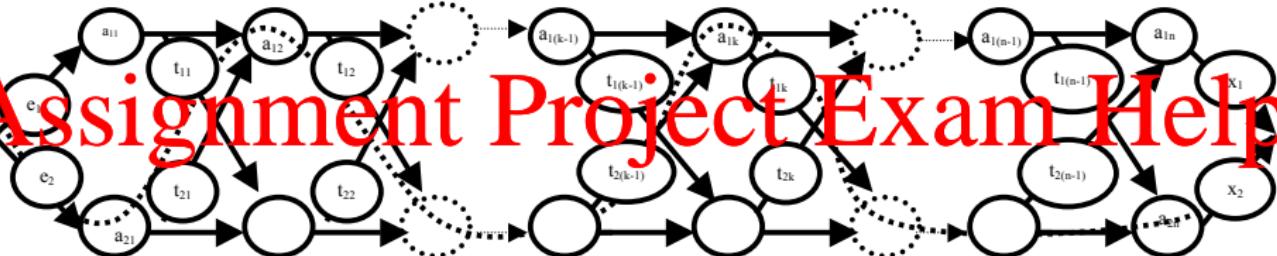


Instance: Two assembly lines with workstations for

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

Dynamic Programming: Assembly line scheduling

Assignment Project Exam Help

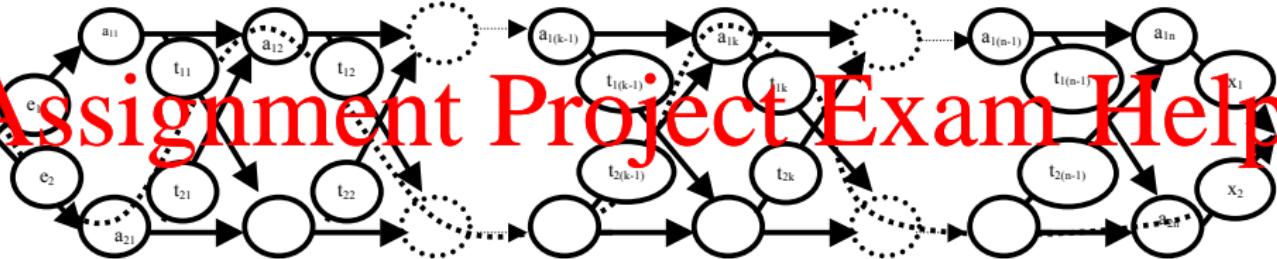


<https://eduassistpro.github.io>

- 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

Assignment Project Exam Help



<https://eduassistpro.github.io>

- 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

Assignment Project Exam Help

<https://eduassistpro.github.io/>

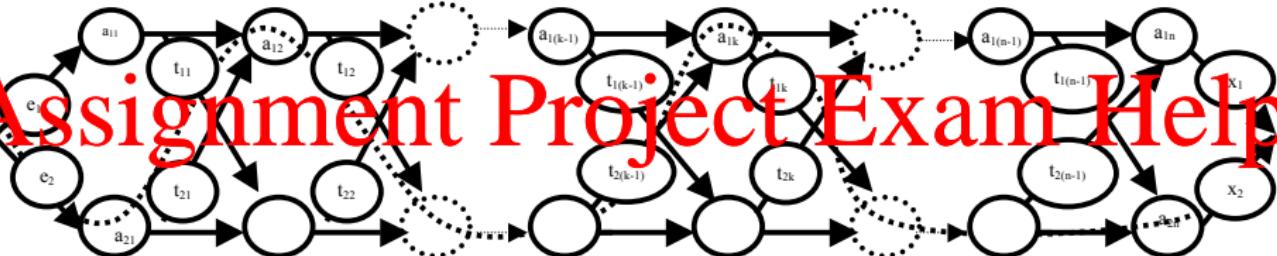
- To bring an unfinished product to the first assembly line it takes e_1 units of time.
 - To bring an unfinished product to the second assembly line it takes e_2 units of time.

Add WeChat edu_assist_p

 - 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

Assignment Project Exam Help

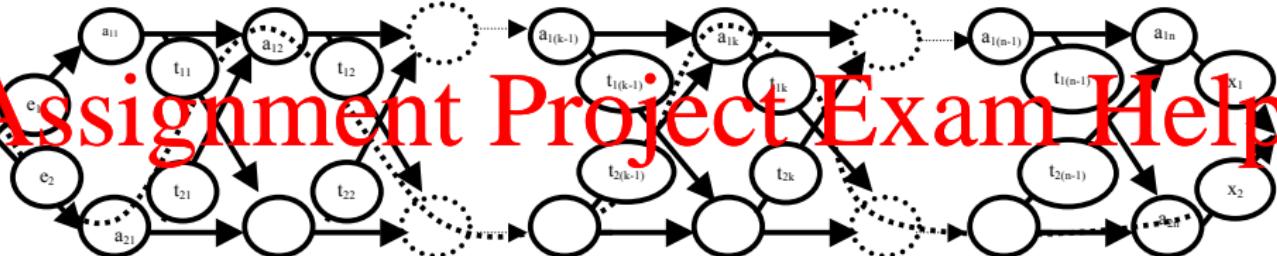


<https://eduassistpro.github.io>

- 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

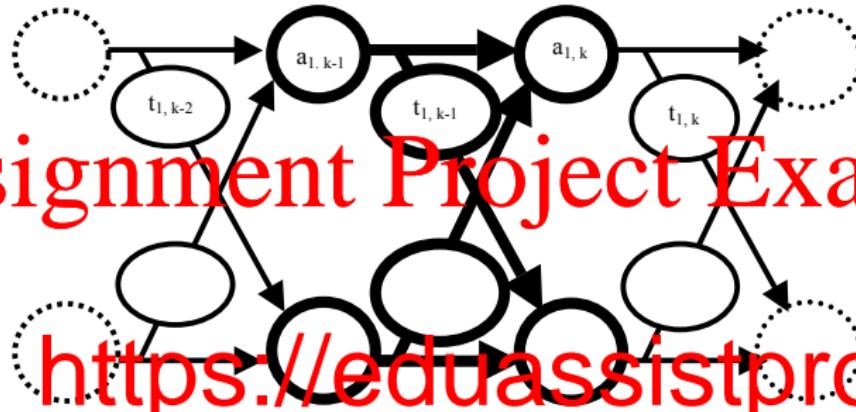
Assignment Project Exam Help



<https://eduassistpro.github.io>

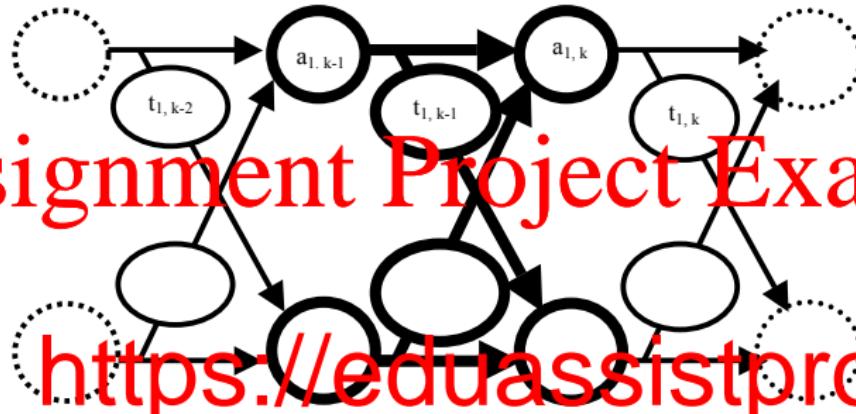
- To bring an unfinished product to the first assembly line it takes e_1 units of time.
- To bring an unfinished product to the second assembly line it takes e_2 units of time.
- To get a finished product from the first assembly line to the warehouse it takes x_1 units of time;
- To get a finished product 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



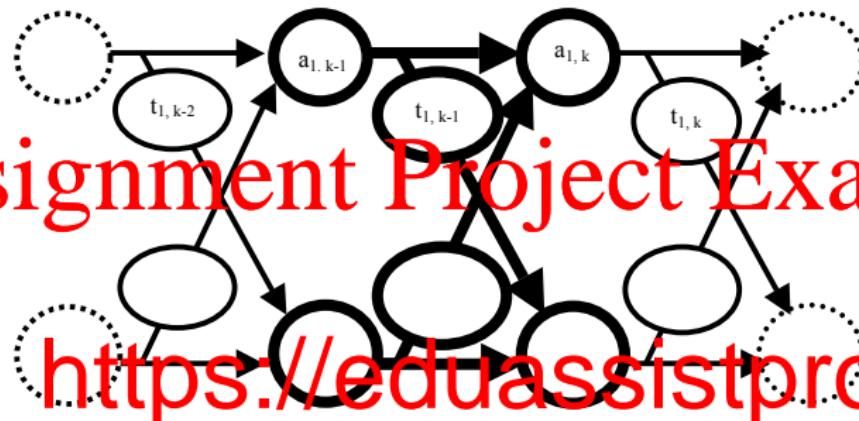
- For each $k \leq n$, we solve subproblems $P(1)$ recursively on k .
- $P(1, k)$: find the minimal amount of time $m(1, k)$ needed to finish the first k jobs, such the k^{th} job is finished on the k^{th} workstation on the first assembly line;
- $P(2, k)$: find the minimal amount of time $m(2, k)$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

Dynamic Programming: Assembly line scheduling



- For each $k \leq n$, we solve subproblems $P(1)$ recursively 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.

Dynamic Programming: Assembly line scheduling



- For each $k \leq n$, we solve subproblems $P(1)$ recursively 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.

Add WeChat **edu_assist_pro**

Assignment Project Exam Help

- We s
- Initi
- Rec

<https://eduassistpro.github.io>

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

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

Add WeChat edu_assist_pro

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

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

- This problem is important because it has the same design logic as the Viterbi algorithm, an extremely important algorithm for many fields such as speech recognition, decoding convolutional codes in telecommunications etc., covered in COMP3121/3821/9101/9801

Assignment Project Exam Help

- We s
- Initi
- Rec

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

Add WeChat edu_assist_pro

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

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

- This problem is important because it has the same design logic as the Viterbi algorithm, an extremely important algorithm for many fields such as speech recognition, decoding convolutional codes in telecommunications etc., covered in COMP3121/3821/9101/9801

Dynamic Programming

Assignment Project Exam Help

- We s
- Initi
- Rec

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

Add WeChat edu_assist_pro

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

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

- This problem is important because it has the same design logic as the Viterbi algorithm, an extremely important algorithm for many fields such as speech recognition, decoding convolutional codes in telecommunications etc., covered in COMP3121/3821/9101/9801

Dynamic Programming

Assignment Project Exam Help

- We s
- Initi
- Rec

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

Add WeChat edu_assist_pro

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

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

- This problem is important because it has the same design logic as the Viterbi algorithm, an extremely important algorithm for many fields such as speech recognition, decoding convolutional codes in telecommunications etc., covered in

Dynamic Programming

Assignment Project Exam Help

- We s
- Initi
- Rec

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

Add WeChat edu_assist_pro

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

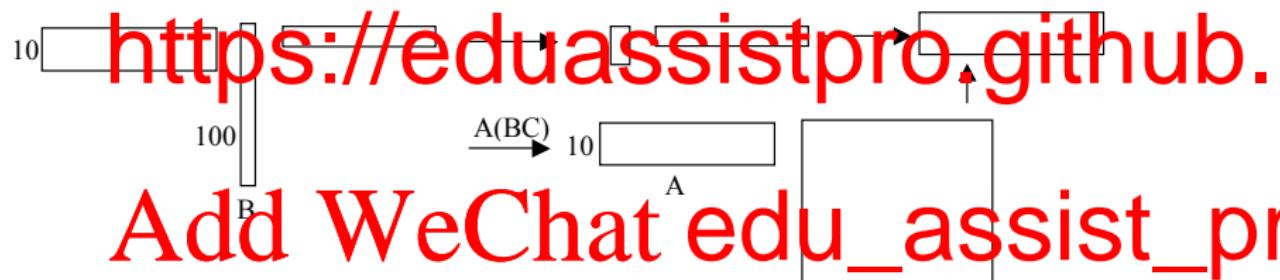
$$opt = \min\{m(1, n) + x_1, \quad m(2, n) + x_2\}.$$

- This problem is important because it has the same design logic as the Viterbi algorithm, an extremely important algorithm for many fields such as speech recognition, decoding convolutional codes in telecommunications etc, covered

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:

Assignment Project Exam Help

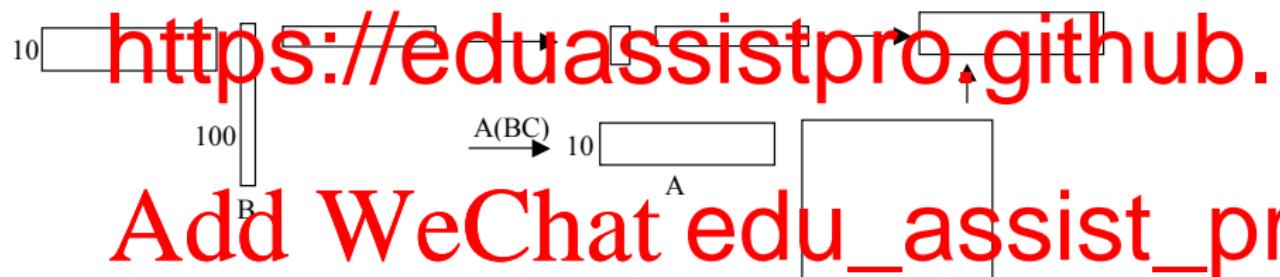


- To evaluate $(AB)C$ we need $(10 \times 5) \times 100 + (10 \times 50) \times 5 = 5000 + 2500 = 7500$ multiplications;
- To evaluate $A(BC)$ we need $(100 \times 50) \times 5 + (10 \times 50) \times 100 = 25000 + 50000 = 75000$ multiplications!

Dynamic Programming: Matrix chain multiplication

- 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:

Assignment Project Exam Help

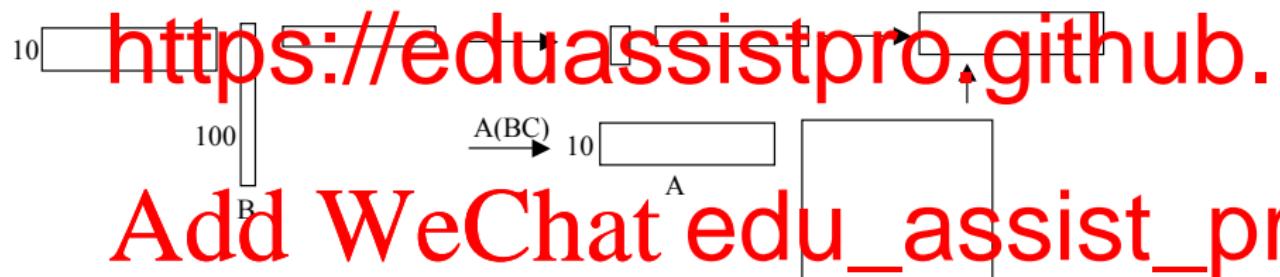


- To evaluate $(AB)C$ we need $(10 \times 5) \times 100 + (10 \times 50) \times 5 = 5000 + 2500 = 7500$ multiplications;
- To evaluate $A(BC)$ we need $(100 \times 5) \times 5 + (10 \times 50) \times 100 = 25000 + 50000 = 75000$ multiplications!

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:

Assignment Project Exam Help

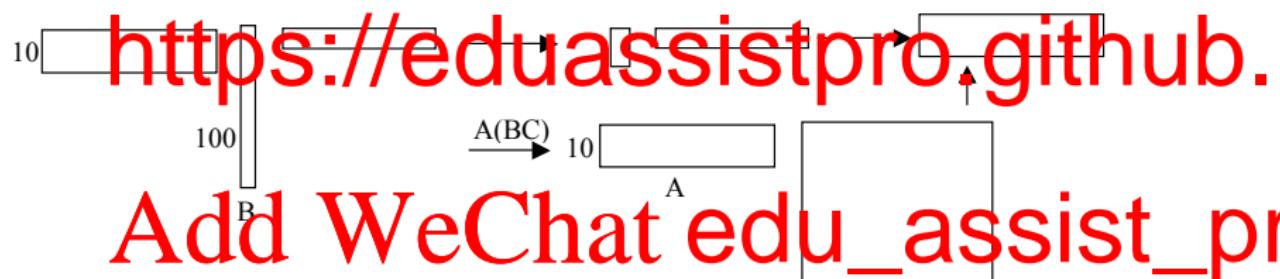


- To evaluate $(AB)C$ we need $(10 \times 5) \times 100 + (10 \times 50) \times 5 = 5000 + 2500 = 7500$ multiplications;
- To evaluate $A(BC)$ we need $(100 \times 5) \times 5 + (10 \times 50) \times 100 = 25000 + 50000 = 75000$ multiplications!

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:

Assignment Project Exam Help

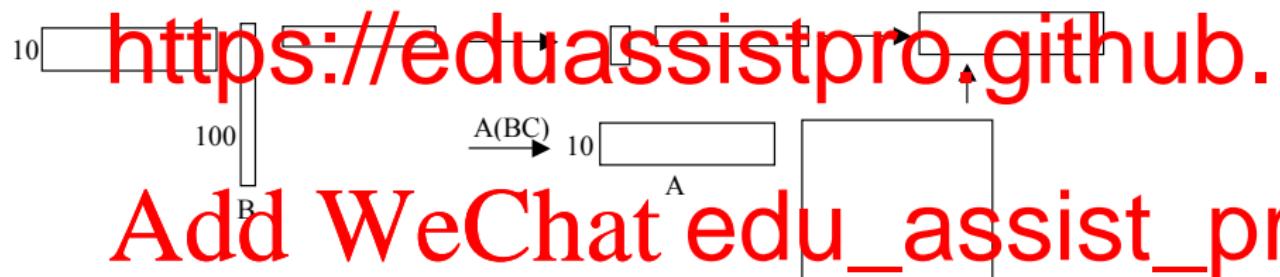


- To evaluate $(AB)C$ we need $(10 \times 5) \times 100 + (10 \times 50) \times 5 = 5000 + 2500 = 7500$ multiplications;
- To evaluate $A(BC)$ we need $(100 \times 5) \times 5 + (10 \times 50) \times 100 = 25000 + 50000 = 75000$ multiplications!

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:

Assignment Project Exam Help



- 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!

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

Assignment Project Exam Help

- The time complexity of this approach is exponential.
- The time complexity can be reduced using recursion (why?):

<https://eduassistpro.github.io>

- 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.

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

Assignment Project Exam Help

- The total number of brackets ($\Theta(n)$)
- The total number of ways to group them ($\Omega(2^n)$)
- The total number of ways to group them using recursion (why?):

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

Add WeChat edu_assist_pro

- 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.

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

Assignment Project Exam Help

- Task: Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The time complexity of binomial coefficient calculation.
- The time complexity of recursion (why?):

<https://eduassistpro.github.io>

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

Add WeChat edu_assist_pro

- 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.

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

Assignment Project Exam Help

- Task: Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The time complexity of binomial coefficient calculation.
- The time complexity of recursion (why?):

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

Add WeChat edu_assist_pro

- 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.

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

Assignment Project Exam Help

- Task: Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The time complexity of binomial coefficient calculation.
- The time complexity of recursion (why?):

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

Add WeChat edu_assist_pro

- 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.

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

Assignment Project Exam Help

- Task: Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.
- The time complexity of binomial coefficient calculation.
- The time complexity of recursion (why?):

<https://eduassistpro.github.io>

Add WeChat edu_assist_pro

- 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.

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

Assignment Project Exam Help

- The s

^{"gro}
_{num}<https://eduassistpro.github.io>

- Note: this looks like it is a case of a “2D recursion, but we can ac 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 $j - i = m$.

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

Task: Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

- The s

“grouping”
number of multiplications

<https://eduassistpro.github.io>

- Note: this looks like it is a case of a “2D recursion, but we can actually do better 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 $j - i = m$.

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

Assignment Project Exam Help

- Task: Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

- The s

^{“gro}
_{num}<https://eduassistpro.github.io>

- Note: this looks like it is a case of a “2D recursion, but we can achieve this 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 $j - i = m$.

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

Assignment Project Exam Help

Task: Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

- The s

“gro
num” <https://eduassistpro.github.io>

- Note: this looks like it is a case of a “2D recursion, but we can ac 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 $j - i = m$.

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

Task: Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

- The s

“grouping”
number of multiplications

- Note: this looks like it is a case of a “2D recursion, but we can actually solve it with a simple “linear” recursion.

Add WeChat edu_assist_pro

- 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 $j - i = m$.

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

Task: Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

- The s

“grouping”
number of multiplications

- Note: this looks like it is a case of a “2D recursion, but we can actually solve 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 $j - i = m$.

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

Task: Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

- The s

“grouping”
number of multiplications

- Note: this looks like it is a case of a “2D recursion, but we can actually solve 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 $j - i = m$.

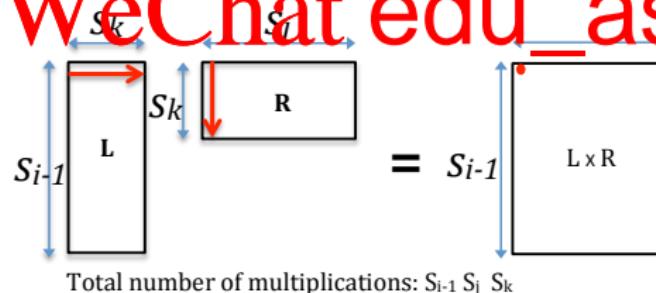
Dynamic Programming: Matrix chain multiplication

- 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 $s_{i-1} \times s_i$.

- Recursion:** We examine all possible ways to place the principal (outermost) multiplication, partitioning the chain into a product $(A_1 \dots A_k)(A_{k+1} \dots A_j)$.

Assignment Project Exam Help

- Note that both $k - i < j - i$ and $j - (k + 1) < j - i$; thus we have the solutions of the subproblems $m(k-i, j)$ and $m(j-k-1, i)$.
- Note that $m(k-i, j)$ 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 $s_{i-1} \cdot s_k \cdot s_j$ multiplications.



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 $s_{i-1} \times s_i$.

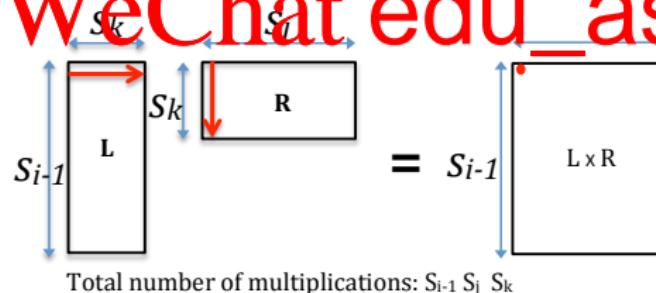
- Recursion:** We examine all possible ways to place the principal (outermost) multiplication, splitting the chain into the product $(A_i \dots A_k)(A_{k+1} \dots A_j)$.

- Note that both $k - i < j - i$ and $j - (k + 1) < j - i$; thus we have the solutions of the subp

$j - ($

- Note 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 , we need $i-1$ many multiplications.



Dynamic Programming: Matrix chain multiplication

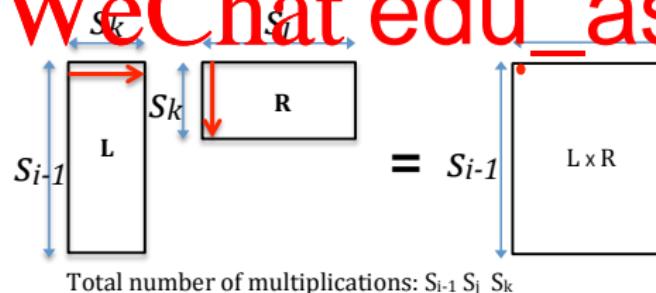
- Let $m(i, j)$ denote the minimal number of multiplications needed to compute the product $A_i A_{i+1} \dots A_{j-1} A_j$; let also the size of matrix A_i be $s_{i-1} \times s_i$.

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

- Note that both $k - i < j - i$ and $j - (k + 1) < j - i$; thus we have the solutions of the subp
 $k - i$ and
 $j - ($

- Note
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 , we need $s_{i-1} \times s_k \times s_j$ multiplications.



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 $s_{i-1} \times s_i$.

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

- Note that both $k - i < j - i$ and $j - (k + 1) < j - i$; thus we have the solutions of the subp

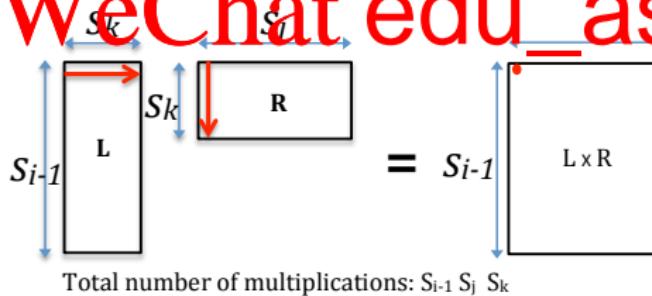
$j - ($

- Note

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 , we need $s_{i-1} \times s_j$ multiplications.

Add WeChat [edu_assist_pro](https://eduassistpro.github.io)



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 $s_{i-1} \times s_i$.

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

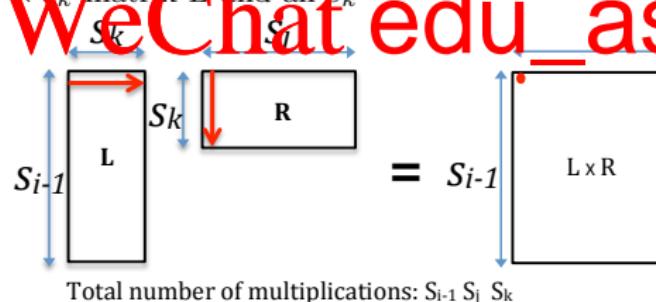
- Note that both $k - i < j - i$ and $j - (k + 1) < j - i$; thus we have the solutions of the subp

$j - ($

- Note

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 , we need $s_{i-1} \cdot s_k \cdot s_j$ multiplications.



- The recursion:

Assignment Project Exam Help

- Not algorithmically

<https://eduassistpro.github.io>

- k for which the minimum in the recursive definition can be obtained to derive the optimal placement of brackets. Achieved by dynamic programming.
- Thus, in the m^{th} slot of the table we are constructing we store all pairs $(m(i, j), k)$ for which $j - i = m$.

- The recursion:

$$m(i, j) = \min_k \{m(i, k) + m(k+1, j) + s_{i-1}s_j s_k\} : i \leq k \leq j - 1$$

- Not algorithmic

<https://eduassistpro.github.io/>

- k for which the minimum in the recursive definition can be obtained to derive the optimal placement of brackets. Add WeChat edu_assist_pro
- Thus, in the m^{th} slot of the table we are constructing we store all pairs $(m(i, j), k)$ for which $j - i = m$.

- The recursion:

Assignment Project Exam Help

- Not algorithmic

<https://eduassistpro.github.io>

- k for which the minimum in the recursive definition can be stored to retrieve the optimal placement of bracketed chain $A_1 \cdots A_n$.
- Thus, in the m^{th} slot of the table we are constructing we store all pairs $(m(i, j), k)$ for which $j - i = m$.

- The recursion:

$$m(i, j) = \min_k \{m(i, k) + m(k+1, j) + s_{i-1}s_j s_k\} : i \leq k \leq j - 1$$

- Not algorithmic
- k for which the minimum in the recursive definition can be stored to retrieve the optimal placement of bracketed chain $A_1 \cdots A_n$.
- Thus, in the m^{th} slot of the table we are constructing we store all pairs $(m(i, j), k)$ for which $j - i = m$.

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

Assignment Project Exam Help

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

- This can be done by finding the longest common subsequence (LCS) of the two sequences.

<https://eduassistpro.github.io/>

- A sequence s is a subsequence of a sequence t if s can be obtained by deleting some of the symbols of t (possibly none or all of them). The remaining symbols of t are called the gaps of the subsequence s .

Add WeChat `edu_assist_pro` to receive help.

- 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.

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

Assignment Project Exam Help

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

- This project can be found at <https://eduassistpro.github.io/>.

- A sequence s is a subsequence of a sequence t if s can be obtained by deleting some of the symbols of t (possibly none or all of them). The remaining symbols of t are called the gaps of the subsequence s .

Add WeChat `edu_assist_pro`

- 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.

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

Assignment Project Exam Help

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

- This <https://eduassistpro.github.io>

- A sequence s is a subsequence of a sequence t if s can be obtained by deleting some of the symbols of t (possibly none or all of them). The remaining symbols of t are called the gaps of s .

Add WeChat `edu_assist_pro`

- 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.

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

Assignment Project Exam Help

- Example: how similar are the genetic codes of two viruses.
- This <https://eduassistpro.github.io> can be used to find the longest common subsequence between two strings.
- A sequence s is a **subsequence** of a string t if s can be obtained by deleting some of the symbols of t (not changing 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.

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

Assignment Project Exam Help

- Example: how similar are the genetic codes of two viruses.
- This <https://eduassistpro.github.io> can be used to find the longest common subsequence between two strings.
- A sequence s is a **subsequence** of a string t if s can be obtained by deleting some of the symbols of t (not changing 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.

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

Assignment Project Exam Help

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

- “2D
the l
 $S_i =$

<https://eduassistpro.github.io>

- Recursion: we fill the table row by row, so the ordering of s
lexicographical order;

Add WeChat edu_assist_pro

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

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

Assignment Project Exam Help

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

- “2D

the l
 $S_i =$

j

- Recursion: we fill the table row by row, so the ordering of s

lexicographical order; <https://eduassistpro.github.io>

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

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

Assignment Project Exam Help

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

- “2D

the l
 $S_i =$

length of
<https://eduassistpro.github.io>

- Recursion: we fill the table row by row, so the ordering of s
lexicographical order; s

Add WeChat edu_assist_pro

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

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

Assignment Project Exam Help

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

- “2D”

the length of
 $S_i = \max_{j=1}^m c[i, j]$

- Recursion: we fill the table row by row, so the ordering of sequences is lexicographical order;

Add WeChat edu_assist_pro

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

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

Assignment Project Exam Help

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

- “2D”
the lo
 $S_i =$

<https://eduassistpro.github.io>

- Recursion: we fill the table row by row, so the ordering of the rows is lexicographical order; S^*

Add WeChat edu_assist_pro

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

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

Assignment Project Exam Help

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

- “2D DP”:
the length of
 $S_i = \max_{j=1}^m c[i, j]$

- Recursion: we fill the table row by row, so the ordering of rows is lexicographical ordering;

Add WeChat edu_assist_pro

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

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

Assignment Project Exam Help

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

- “2D DP”:
the length of
 $S_i = \max_{j=1}^m c[i, j]$

- Recursion: we fill the table row by row, so the ordering of rows is lexicographical ordering;

Add WeChat edu_assist_pro

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

Retrieving a longest common subsequence:

Assignment Project Exam Help

<https://eduassistpro.github.io>

Add WeChat edu_assist_pro

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

<https://eduassistpro.github.io>

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

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

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

Add WeChat `edu_assist_pro`

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

- 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

<https://eduassistpro.github.io>

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

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

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

Add WeChat `edu_assist_pro`

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

- 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

<https://eduassistpro.github.io>

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

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

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

Add WeChat `edu_assist_pro`

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

- 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

<https://eduassistpro.github.io>

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

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

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

Add WeChat `edu_assist_pro`

But

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

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

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

Assignment Project Exam Help

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

- We a

S, S^*, S^{**} .

- for all long

the
 $i \in \{1, 2, \dots, n\}$,
 $j \in \{1, 2, \dots, m\}$,
 $l \in \{1, 2, \dots, k\}$,

$S_j^* = \langle b_1, b_2, \dots, b_j \rangle$ and $S_l^{**} = \langle c_1, c_2, \dots, c_l \rangle$

- Recurrence

Add WeChat edu_assist_pro

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

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

Assignment Project Exam Help

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

- We a

S, S^*, S^{**} .

- for all long

the
 i
 $a_1, a_2, \dots, a_i \rangle$,

$S_j^* = \langle b_1, b_2, \dots, b_j \rangle$ and $S_l^{**} = \langle c_1, c_2, \dots, c_l \rangle$

- Recur

Add WeChat edu_assist_pro

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

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

Assignment Project Exam Help

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

- We are given three sequences S, S^*, S^{**} .

- for all $i \in \{1, 2, \dots, n\}$, $j \in \{1, 2, \dots, m\}$, $l \in \{1, 2, \dots, k\}$,
longest common subsequence of a_1, a_2, \dots, a_i , b_1, b_2, \dots, b_j , c_1, c_2, \dots, c_l

$$S_j^* = \langle b_1, b_2, \dots, b_j \rangle \text{ and } S_l^{**} = \langle c_1, c_2, \dots, c_l \rangle$$

- Recurrence relation

Add WeChat edu_assist_pro

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

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

Assignment Project Exam Help

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

- We aim to find a longest common subsequence of S, S^*, S^{**} .

- for all $i \in \{0, 1, 2, \dots, n\}$ and $j \in \{0, 1, 2, \dots, m\}$ and $l \in \{0, 1, 2, \dots, k\}$,
longest common subsequence of $\langle a_1, a_2, \dots, a_i \rangle$,
 $S_j^* = \langle b_1, b_2, \dots, b_j \rangle$ and $S_l^{**} = \langle c_1, c_2, \dots, c_l \rangle$

- Recurrence relation: Add WeChat edu_assist_pro

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

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

Assignment Project Exam Help

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

- We aim to find a longest common subsequence of S, S^*, S^{**} .

- for all $i \in \{1, 2, \dots, n\}$, $j \in \{1, 2, \dots, m\}$, $l \in \{1, 2, \dots, k\}$, if the longest common subsequence of $\langle a_1, a_2, \dots, a_i \rangle$, $S_j^* = \langle b_1, b_2, \dots, b_j \rangle$ and $S_l^{**} = \langle c_1, c_2, \dots, c_l \rangle$ is $d[i, j, l]$.

- Recursion: Add WeChat edu_assist_pro

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

Assignment Project Exam Help

- **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 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:
then
order

<https://eduassistpro.github.io>

Add WeChat `edu_assist_pro`

$$LCS(s, s^*)$$

shortest super-sequence $S = axbyacazda$

- **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 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 .

- Sol
then
orde

<https://eduassistpro.github.io>

Add WeChat `edu_assist_pro`

$$LCS(s, s^*)$$

shortest super-sequence $S = axbyacazda$

Assignment Project Exam Help

- **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 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 .

• Sol
then <https://eduassistpro.github.io>

Add WeChat `edu_assist_pro`

$$LCS(s, s^*)$$

shortest super-sequence $S = axbyacazda$

- **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 .

Assignment Project Exam Help

- Tas

- Not num
<https://eduassistpro.github.io>
called the *edit distance* between A and B.

- If the sequences are DNA bases, and the cost probability of the process of adding, deletion and mutation, then the probability that one sequence mutates into another sequence by DNA copying.
- 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

- **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 .

Assignment Project Exam Help

- Tas

- Not num
called the *edit distance* between A and B.

- If the sequences are DNA bases, and the cost probability of the process of adding, deletion, and the probability that one sequence mutates into another of DNA copying.
- 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 $j \leq m$.

- **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 .

Assignment Project Exam Help

- Tas
- Not num <https://eduassistpro.github.io> called *the edit distance* between A and B.
- If the sequences are DNA bases and the cost probability of the process of adding, deletion and mutation is given, calculate the probability that one sequence mutates into another sequence by the process of DNA copying.
- 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

- **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 .

Assignment Project Exam Help

- Tas
- Not num <https://eduassistpro.github.io> called *the edit distance* between A and B.
- If the sequences are sequences of DNA bases and the cost probabilities of the corresponding mutations, then it's the probability that one sequence mutates into another of DNA copying.
- 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 $j \leq m$.

- **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 .

Assignment Project Exam Help

- Tas
- Not num <https://eduassistpro.github.io> called *the edit distance* between A and B.
- If the sequences are sequences of DNA bases and the cost probabilities of the corresponding mutations, then it's the probability that one sequence mutates into another of DNA copying.
- 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 $j \leq m$.

- Recursion: we again fill the table of solutions $C(i, j)$ for subproblem $P(i, j)$.
We have three options:
 - 1. if $A[i] = B[j]$, then we can skip this character and move to $P(i-1, j-1)$;
 - 2. if $A[i] \neq B[j]$, then we can either replace $A[i]$ by $B[j]$ (cost c_R) or delete $A[i]$ (cost c_D) or insert $B[j]$ at the end (cost c_I);
 - 3. if $A[i] \neq B[j]$, then we can also do nothing (cost 0), if $A[i-1] = B[j-1]$.

<https://eduassistpro.github.io/>

- cost $c_D + C(i - 1, j)$ corresponds to the option if you delete $A[i..i]$ and then apply $P(i-1, j)$;
- cost $c_I + C(i, j - 1)$ corresponds to the option if you insert $B[j..j]$ and then apply $P(i, j-1)$;
- the third option corresponds to first transforming $A[1..i-1]$ to $B[1..j-1]$ and then:
 - ① if $A[i] = B[j]$ do nothing, thus incurring a cost of only $C(i-1, j-1)$;
 - ② if $A[i] \neq B[j]$ replace $A[i]$ by $B[j]$ with a total cost of $C(i-1, j-1) + c_R$.

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 subproblem $P(i, j)$ row by row (why is this OK?):

Assignment Project Exam Help

<https://eduassistpro.github.io/>

- cost $c_D + C(i - 1, j)$ corresponds to the option if $y_{A[1..i-1]}$ is mapped to $B[1..j-1]$ and then delete $A[i]$;
- cost $c_I + C(i - 1, j) - c_P$ corresponds to the option if $y_{A[1..i-1]}$ is mapped 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: 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?):

<https://eduassistpro.github.io/>

- cost $c_D + C(i - 1, j)$ corresponds to the option if you transform $A[1..i - 1]$ into $B[1..j]$ and then delete $A[i]$
- cost $c_I + C(i - 1, j) - c_P$ corresponds to the option if you transform $A[1..i - 1]$ 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: 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 subproblem $P(i, j)$ row by row (why is this OK?):

<https://eduassistpro.github.io/>

- cost $c_D + C(i - 1, j)$ corresponds to the option if you transform $A[1..i - 1]$ into $B[1..j]$ and then delete $A[i]$
- cost $C(i, j - 1) + c_I$ corresponds to the option if you 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: 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 subproblem $P(i, j)$ row by row (why is this OK?):

<https://eduassistpro.github.io/>

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

Assignment Project Exam Help

<https://eduassistpro.github.io/>

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

Assignment Project Exam Help

<https://eduassistpro.github.io/>

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

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

Assignment Project Exam Help

- What is the problem?
- May we consider the expression as a tree?

<https://eduassistpro.github.io/>

- maybe we could consider which the principal operations are $A[i..k] \oplus A[k+1..j]$. Where \oplus is whatever operation is used.
- But when would such expression be maximised if there could be 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: Maximizing an expression

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

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

Assignment Project Exam Help

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

- Wha

- May
expr

<https://eduassistpro.github.io/>

- maybe we could consider which the principal operations s $A[i..k] \circ A[k+1..j]$. Where \circ is whatever operation

Add WeChat edu_assist_pro

- But when would such expression be maximised if there could be 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: Maximizing an expression

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

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

Assignment Project Exam Help

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

- Wha

- May
expr

- maybe we could consider which the principal operations s
 $A[i..k] \circ A[k+1..j]$. Where \circ is whatever operation

Add WeChat edu_assist_pro

- But when would such expression be maximised if there could be 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: Maximizing an expression

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

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

Assignment Project Exam Help

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

- Wha
- May expr

<https://eduassistpro.github.io>

- maybe we could consider which the principal operations s $A[i..k] \circ A[k+1..j]$. Where \circ is whatever operation
- But when would such expression be maximised if there could be 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: Maximizing an expression

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

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

Assignment Project Exam Help

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

- Wha

- May
expr

- maybe we could consider which the principal operations s
 $A[i..k] \odot A[k+1..j]$. Here \odot is whatever operatio

Add WeChat [edu_assist_pro](https://eduassistpro.github.io)

- But when would such expression be maximised if there could be 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: Maximizing an expression

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

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

Assignment Project Exam Help

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

- Wha

- May
expr

- maybe we could consider which the principal operations s

$A[i..k] \odot A[k+1..j]$. Here \odot is whatever operation

Add WeChat [edu_assist_pro](https://eduassistpro.github.io)

- But when would such expression be maximised if there coul

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: Maximizing an expression

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

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

Assignment Project Exam Help

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

- Wha

- May
expr

- maybe we could consider which the principal operations s

$A[i..k] \odot A[k+1..j]$. Here \odot is whatever operation

Add WeChat [edu_assist_pro](https://eduassistpro.github.io)

- But when would such expression be maximised if there coul

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: Maximizing an expression

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

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

Assignment Project Exam Help

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

- Wha

- May
expr

<https://eduassistpro.github.io>

- maybe we could consider which the principal operations s

$A[i..k] \odot A[k + 1..j]$. Here \odot is whatever operatio

Add WeChat edu_assist_pro

- But when would such expression be maximised if there coul

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.

Assignment Project Exam Help

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:

<https://eduassistpro.github.io>

- Hint: Order turtles in an increasing order of the sum of their strength, and proceed by recursion.

Add WeChat `edu_assist_pro`

- You can find a solution to this problem and of another in the class website (class resources, file "More Dynamic Programming")

Assignment Project Exam Help

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: Top of the class website
- Hint: Order turtles in an increasing order of the sum of their strength, and proceed by recursion.
- You can find a solution to this problem and of another in the class website (class resources, file "More Dynamic Programming")

Add WeChat edu_assist_pro

Assignment Project Exam Help

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:** Top of the stack
- **Hint:** Order turtles in an increasing order of the sum of their strength, and proceed by recursion.
- You can find a solution to this problem and of another in the class website (class resources, file "More Dynamic Programming")
<https://eduassistpro.github.io>
Add WeChat edu_assist_pro

Assignment Project Exam Help

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: Top of the class website
- Hint: Order turtles in an increasing order of the sum of their strength, and proceed by recursion.
- Add WeChat `edu_assist_pro`
- You can find a solution to this problem and of another interview question on the class website (class resources, file “More Dynamic Programming”)

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 .

- Sol

cont

<https://eduassistpro.github.io>

- Thu

- Subproblems: For every $v \in V$ and every $n \geq 0$, let $\text{opt}(n, v)$ be the length of a shortest path from s to v using at most n edges.

Add WeChat `edu_assist_pro`

- 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.
- Note that if the shortest path from a vertex v to t is $(v, p_1, p_2, \dots, p_k, t)$ then $(p_1, p_2, \dots, p_k, t)$ must be the shortest path from p_1 to t , and $(v, p_1, p_2, \dots, p_k)$ must also be the shortest path from v to p_k .

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 .

- Sol

cont

<https://eduassistpro.github.io>

- Thu

- Subproblems: For every $v \in V$ and every $n \geq 0$, let $\text{opt}(n, v)$ be the length of a shortest path from s to v using at most n edges.

Add WeChat `edu_assist_pro`

- 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.
- Note that if the shortest path from a vertex v to t is $(v, p_1, p_2, \dots, p_k, t)$ then $(p_1, p_2, \dots, p_k, t)$ must be the shortest path from p_1 to t , and $(v, p_1, p_2, \dots, p_k)$ must also be the shortest path from v to p_k .

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 .

- Sol

cont

<https://eduassistpro.github.io>

- Thu

- Subproblems: For every $v \in V$ and every $n \geq 0$, let $\text{opt}(n, v)$ be the length of a shortest path from s to v using at most n edges.

Add WeChat `edu_assist_pro`

- 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.
- Note that if the shortest path from a vertex v to t is $(v, p_1, p_2, \dots, p_k, t)$ then $(p_1, p_2, \dots, p_k, t)$ must be the shortest path from p_1 to t , and $(v, p_1, p_2, \dots, p_k)$ must also be the shortest path from v to p_k .

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 .
- **Sol**

cont

<https://eduassistpro.github.io>

- Thu

- Subproblems: For every $v \in V$ and every $n \geq 0$ let $\text{opt}(n, v)$ be the length of a shortest path from s to v using at most n 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.
- Note that if the shortest path from a vertex v to t is $(v, p_1, p_2, \dots, p_k, t)$ then $(p_1, p_2, \dots, p_k, t)$ must be the shortest path from p_1 to t , and $(v, p_1, p_2, \dots, p_k)$ must also be the shortest path from v to p_k .

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 .
- **Sol**
cont
- Thu

<https://eduassistpro.github.io>

- Subproblems: For every $v \in V$ and every $n \geq 0$, let $\text{opt}(n, v)$ be the length of a shortest path from s to v using at most n 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.
- Note that if the shortest path from a vertex v to t is $(v, p_1, p_2, \dots, p_k, t)$ then $(p_1, p_2, \dots, p_k, t)$ must be the shortest path from p_1 to t , and $(v, p_1, p_2, \dots, p_k)$ must also be the shortest path from v to p_k .

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 .

• Sol

cont

<https://eduassistpro.github.io>

• Thu

• **Subproblems:** For every $v \in V$ and every $n \geq 0$ let $\text{opt}(n, v)$ be the length of a shortest path from s to v .

- 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.
- Note that if the shortest path from a vertex v to t is $(v, p_1, p_2, \dots, p_k, t)$ then $(p_1, p_2, \dots, p_k, t)$ must be the shortest path from p_1 to t , and $(v, p_1, p_2, \dots, p_k)$ must also be the shortest path from v to p_k .

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 .

- Sol

cont

<https://eduassistpro.github.io>

- Thu

- **Subproblems:** For every $v \in V$ and every $n \geq 0$ let $\text{opt}(n, v)$ be the length of a shortest path from s to v using at most n 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.
- Note that if the shortest path from a vertex v to t is $(v, p_1, p_2, \dots, p_k, t)$ then $(p_1, p_2, \dots, p_k, t)$ must be the shortest path from p_1 to t , and $(v, p_1, p_2, \dots, p_k)$ must also be the shortest path from v to p_k .

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 .

• Sol

cont

<https://eduassistpro.github.io>

• Thu

• **Subproblems:** For every $v \in V$ and every $n \geq 0$, let $\text{opt}(n, v)$ be the length of a shortest path from s to v .

- 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.
- Note that if the shortest path from a vertex v to t is $(v, p_1, p_2, \dots, p_k, t)$ then $(p_1, p_2, \dots, p_k, t)$ must be the shortest path from p_1 to t , and $(v, p_1, p_2, \dots, p_k)$ must also be the shortest path from v to p_k .

Dynamic Programming: Bellman Ford algorithm

- Let us denote the length of the shortest path from s to v among all paths which contain at most i edges by $\text{opt}(i, v)$, and let $\text{pred}(i, v)$ be the immediate predecessor of vertex v on such shortest path.

Assignment Project Exam Help

$$\text{opt}(i, v) = \min_{p \in V} \{\text{opt}(i-1, p) + w(e(p, v))\};$$

$\text{pred}(i, v)$ is defined as follows:
Recursion step: $\text{opt}(i, v) = \min_{p \in V} \{\text{opt}(i-1, p) + w(e(p, v))\};$
Initial value: $\text{opt}(0, v) = 0$ for every vertex v .
Final solution: $p(n-1, v)$ for all $v \in G$.

<https://eduassistpro.github.io>

- Final solutions: $p(n-1, v)$ for all $v \in G$.
- Computing $\text{opt}(i, v)$ uses a tight loop over all vertices v for each i .
Each v , \min is taken over all edges $e(p, v)$ in G .
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

- Let us denote the length of the shortest path from s to v among all paths which contain at most i edges by $\text{opt}(i, v)$, and let $\text{pred}(i, v)$ be the immediate predecessor of vertex v on such shortest path.

Assignment Project Exam Help

Recursion:

$$\text{opt}(i, v) = \min_{p \in V} \{\text{opt}(i-1, p) + w(e(p, v))\};$$

$\text{pred}(i, v)$

$\text{opt}(i-1, v)$

(here vertex v)

- Final solutions: $p(n-1, v)$ for all $v \in G$.

- Computing $\text{opt}(i, v)$ uses a table for each v , min is taken over all edges $e(p, v)$ in 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

- Let us denote the length of the shortest path from s to v among all paths which contain at most i edges by $\text{opt}(i, v)$, and let $\text{pred}(i, v)$ be the immediate predecessor of vertex v on such shortest path.

Assignment Project Exam Help

Recursion:

$$\text{opt}(i, v) = \min_{p \in V} \{\text{opt}(i-1, p) + w(e(p, v))\};$$

$\text{pred}(i, v)$

$\text{opt}(i-1, v)$

(her vertex v)

- Final solutions: $p(n-1, v)$ for all $v \in G$.
- Computing $\text{opt}(i, v)$ uses a table for each v , min is taken over all edges $e(p, v)$ in 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

- Let us denote the length of the shortest path from s to v among all paths which contain at most i edges by $\text{opt}(i, v)$, and let $\text{pred}(i, v)$ be the immediate predecessor of vertex v on such shortest path.

Recursion:

$$\text{opt}(i, v) = \min(\text{opt}(i - 1, v), \min_{p \in V} \{\text{opt}(i - 1, p) + w(e(p, v))\});$$

$\text{pred}(i, v)$

(her <https://eduassistpro.github.io>)
vertex v .

- Final solutions: $p(n - 1, v)$ for all $v \in G$.
- Computation $\text{opt}(i, v)$ runs in time $O(|V| \times |E|)$ for each v , \min is taken over all edges $e(p, v)$ in 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

- Let us denote the length of the shortest path from s to v among all paths which contain at most i edges by $\text{opt}(i, v)$, and let $\text{pred}(i, v)$ be the immediate predecessor of vertex v on such shortest path.

Assignment Project Exam Help

Recursion:

$$\text{opt}(i, v) = \min(\text{opt}(i - 1, v), \min_{p \in V} \{\text{opt}(i - 1, p) + w(e(p, v))\});$$

$\text{pred}(i, v)$

$\text{opt}(i - 1, v)$

(her <https://eduassistpro.github.io>)
vertex v .

- Final solutions: $p(n - 1, v)$ for all $v \in G$.
- Computation $\text{opt}(i, v)$ runs in time $O(|V| \times |E|)$ for each v , \min is taken over all edges $e(p, v)$ in 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

- Let us denote the length of the shortest path from s to v among all paths which contain at most i edges by $\text{opt}(i, v)$, and let $\text{pred}(i, v)$ be the immediate predecessor of vertex v on such shortest path.

Recursion:

$$\text{opt}(i, v) = \min_{p \in V} \{\text{opt}(i-1, p) + w(e(p, v))\};$$

$\text{pred}(i, v)$

$\text{opt}(i-1, v)$

(her
vertex v)

- Final solutions: $p(n-1, v)$ for all $v \in G$.
- Computation $\text{opt}(i, v)$ runs in time $O(|V| \times |E|)$ for each v , \min is taken over all edges $e(p, v)$ in 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: Floyd Warshall algorithm

- Let again $G = (V, E)$ be a directed weighted graph where $V = \{v_1, v_2, \dots, v_n\}$ and where weights $w(e(v_p, v_q))$ 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 c_{v_p} be the shortest path from v_p to a vertex v_k .

<https://eduassistpro.github.io>

- Then

Add WeChat edu_assist_pro

- Thus, we gradually relax the constraint that the intermediary vertices have to belong to $\{v_1, v_2, \dots, v_k\}$.
- Algorithm runs in time $|V|^3$.

Dynamic Programming: Floyd Warshall algorithm

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

Assignment Project Exam Help

- 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 c_{v_p} be the shortest path from v_p to a vertex v_q in $\{v_1, v_2, \dots, v_k\}$.

<https://eduassistpro.github.io>

- Then

$c_{v_p} = \min_{v_q \in \{v_1, v_2, \dots, v_k\}} \{c_{v_p}(v_q)\}$

- Thus, we gradually **relax** the constraint that the intermediary vertices have to belong to $\{v_1, v_2, \dots, v_k\}$.
- Algorithm runs in time $|V|^3$.

Dynamic Programming: Floyd Warshall algorithm

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

Assignment Project Exam Help

- 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 a
vert
 $\{v_1,$

<https://eduassistpro.github.io>

- Then

$\text{opt}(v_p, v_q) = \min_{v_k \in V} \{ \text{opt}(v_p, v_k) + w(v_k, v_q) \}$

- Thus, we gradually **relax** the constraint that the intermediary vertices have to belong to $\{v_1, v_2, \dots, v_k\}$.
- Algorithm runs in time $|V|^3$.

Dynamic Programming: Floyd Warshall algorithm

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

Assignment Project Exam Help

- 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 a vertex $\{v_1, v_2, \dots, v_k, v_p\}$ to a

<https://eduassistpro.github.io>

- Then

$\text{opt}(k, v_p, v_q) := \min_{v_k} \{\text{opt}(k-1, v_p, v_q), \text{opt}(k-1, v_k, v_q)\}$

- Thus, we gradually **relax** the constraint that the intermediary vertices have to belong to $\{v_1, v_2, \dots, v_k\}$.
- Algorithm runs in time $|V|^3$.

Dynamic Programming: Floyd Warshall algorithm

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

Assignment Project Exam Help

- 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 a
vert
 $\{v_1,$

<https://eduassistpro.github.io>

- Then

$\text{opt}(k, v_p, v_q) := \min_{v_k} \{\text{opt}(k-1, v_p, v_k), \text{opt}(k-1, v_k, v_q)\}$

- Thus, we gradually **relax** the constraint that the intermediary vertices have to belong to $\{v_1, v_2, \dots, v_k\}$.
- Algorithm runs in time $|V|^3$.

Dynamic Programming: Floyd Warshall algorithm

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

Assignment Project Exam Help

- 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 a
vert
 $\{v_1,$

<https://eduassistpro.github.io>

- Then

$\text{opt}(k, v_p, v_q) := \min_{v_k} \{\text{opt}(k-1, v_p, v_k), \text{opt}(k-1, v_k, v_q)\}$

- Thus, we gradually **relax** the constraint that the intermediary vertices have to belong to $\{v_1, v_2, \dots, v_k\}$.
- Algorithm runs in time $|V|^3$.

Another example of relaxation:

- Compute the number of partitions of a positive integer n . That is to say the number of distinct multi-sets of positive integers $\{n_1, \dots, n_k\}$ which sum up to n , i.e., such that $n_1 + \dots + n_k = n$.

Assignment Project Exam Help

Hin

i.e.,

hav

<https://eduassistpro.github.io>

We are looking for $\text{nump}(n, n)$ but the recursion is based on relaxation of the allowed size i of the parts of j for all

definition of $\text{nump}(n, j)$ diminish the case

component value ≤ -1 and the new rule will

size i .

Add WeChat edu_assist_pro

Another example of relaxation:

- Compute the number of partitions of a positive integer n . That is to say the number of distinct multi-sets of positive integers $\{n_1, \dots, n_k\}$ which sum up to n , i.e., such that $n_1 + \dots + n_k = n$.

Note: multi-set means that the set can contain several copies of the same number, but all permutations of elements count as a single multi-set.

Hin

i.e.,

hav

<https://eduassistpro.github.io>

We are looking for $\text{nump}(n, n)$ but the recursion is based on relaxation of the allowed size i of the parts of j for all

definition of $\text{nump}(n, j)$ diminish the case

component value ≤ -1 and the new rule will

size i .

Add WeChat edu_assist_pro

Another example of relaxation:

- Compute the number of partitions of a positive integer n . That is to say the number of distinct multi-sets of positive integers $\{n_1, \dots, n_k\}$ which sum up to n , i.e., such that $n_1 + \dots + n_k = n$.

Note: multi-sets means that the set can contain several copies of the same number, but all permutations of elements count as a single multi-set.

Hin
i.e.,
hav

<https://eduassistpro.github.io>

We are looking for $\text{nump}(n, n)$ but the recursion is based on relaxation of the allowed size i of the parts of j for all definition of $\text{nump}(\cdot, j)$ distinguish the case where all components are $\leq i - 1$ and the case where at least one component has size i .

Add WeChat edu_assist_pro

Assignment Project Exam Help

You have
minute ea
at differen
thick ness

45 second interval?

Add WeChat edu_assist_pro