CSCI-SHU 220: Algorithms
Dynamic Programming I

NYU Shanghai
Spring 2025

Weakness of greedy algorithms

- We have seen that many optimization problems can be efficiently solved using greedy algorithms.

Weakness of greedy algorithms

- We have seen that many optimization problems can be efficiently solved using greedy algorithms.
- However, on the other hand, greedy algorithms do not always work: For some problems, greedy algorithms can't give optimal solutions.

Weakness of greedy algorithms

- We have seen that many optimization problems can be efficiently solved using greedy algorithms.
- However, on the other hand, greedy algorithms do not always work: For some problems, greedy algorithms can't give optimal solutions.
- The main reason for why a greedy algorithm fails is that it considers optimality locally instead of globally when making a decision.

Weakness of greedy algorithms

- We have seen that many optimization problems can be efficiently solved using greedy algorithms.
- However, on the other hand, greedy algorithms do not always work: For some problems, greedy algorithms can't give optimal solutions.
- The main reason for why a greedy algorithm fails is that it considers optimality locally instead of globally when making a decision.
    - If locally optimal = globally optimal, the greedy algorithm works.

Weakness of greedy algorithms

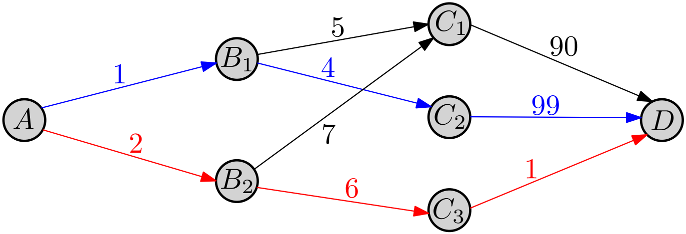
- We have seen that many optimization problems can be efficiently solved using greedy algorithms.
- However, on the other hand, greedy algorithms do not always work: For some problems, greedy algorithms can't give optimal solutions.
- The main reason for why a greedy algorithm fails is that it considers optimality locally instead of globally when making a decision.
  - If locally optimal $=$ globally optimal, the greedy algorithm works.
  - If locally optimal $\neq$ globally optimal, the greedy algorithm fails.

Weakness of greedy algorithms

- We have seen that many optimization problems can be efficiently solved using greedy algorithms.
- However, on the other hand, greedy algorithms do not always work: For some problems, greedy algorithms can't give optimal solutions.
- The main reason for why a greedy algorithm fails is that it considers optimality locally instead of globally when making a decision.
  - If locally optimal $=$ globally optimal, the greedy algorithm works.
  - If locally optimal $\neq$ globally optimal, the greedy algorithm fails.
- So for some problems, we have to consider optimality globally.

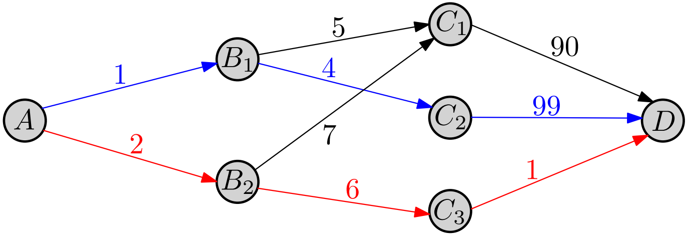
Weakness of greedy algorithms

- **Finding the shortest path from left to right**



Weakness of greedy algorithms

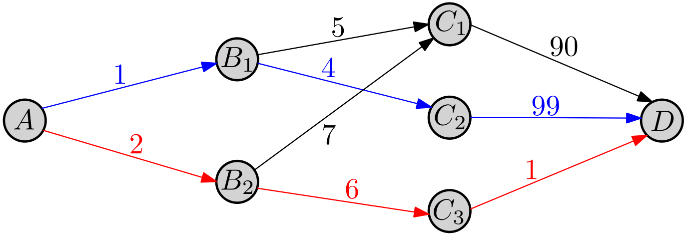
- **Finding the shortest path from left to right**



- Moving $A \rightarrow B_1$ only costs 1 but brings us to a "bad" position B_1. Moving $A \rightarrow B_2$ costs 2 but brings us to a "good" position B_2.

Weakness of greedy algorithms

- **Finding the shortest path from left to right**



- Moving $A \to B_1$ only costs 1 but brings us to a "bad" position B_1. Moving $A \to B_2$ costs 2 but brings us to a "good" position B_2.
- Should consider the cost and the position simultaneously.

- **How to make a decision**

Dynamic programming

- **How to make a decision**
- If we move $A \to B_1$ in the first step, what's the minimum total cost to reach D?

- **How to make a decision**
- If we move $A \to B_1$ in the first step, what's the minimum total cost to reach D? $\text{cost}(A \to B_1) + \textsc{ShortestPath}(B_1 \to D)$

Dynamic programming

- **How to make a decision**
- If we move $A \to B_1$ in the first step, what's the minimum total cost to reach D? $\text{cost}(A \to B_1) + \textsc{ShortestPath}(B_1 \to D)$
- If we move $A \to B_2$ in the first step, what's the minimum total cost to reach D?

Dynamic programming

- **How to make a decision**
- If we move $A \to B_1$ in the first step, what's the minimum total cost to reach D? $\text{cost}(A \to B_1) + \textsc{ShortestPath}(B_1 \to D)$
- If we move $A \to B_2$ in the first step, what's the minimum total cost to reach D? $\text{cost}(A \to B_2) + \textsc{ShortestPath}(B_2 \to D)$

Dynamic programming

- **How to make a decision**
- If we move $A \to B_1$ in the first step, what's the minimum total cost to reach D? $\text{cost}(A \to B_1) + \textsc{ShortestPath}(B_1 \to D)$
- If we move $A \to B_2$ in the first step, what's the minimum total cost to reach D? $\text{cost}(A \to B_2) + \textsc{ShortestPath}(B_2 \to D)$
- $A \to B_1$ or $A \to B_2$?

- **How to make a decision**
- If we move $A \to B_1$ in the first step, what's the minimum total cost to reach D? $\text{cost}(A \to B_1) + \textsc{ShortestPath}(B_1 \to D)$
- If we move $A \to B_2$ in the first step, what's the minimum total cost to reach D? $\text{cost}(A \to B_2) + \textsc{ShortestPath}(B_2 \to D)$
- $A \to B_1$ or $A \to B_2$? Depend on which of the above two is smaller.

Dynamic programming

- **How to make a decision**
- If we move $A \to B_1$ in the first step, what's the minimum total cost to reach D? $\text{cost}(A \to B_1) + \textsc{ShortestPath}(B_1 \to D)$
- If we move $A \to B_2$ in the first step, what's the minimum total cost to reach D? $\text{cost}(A \to B_2) + \textsc{ShortestPath}(B_2 \to D)$
- $A \to B_1$ or $A \to B_2$? Depend on which of the above two is smaller.
- Knowing the shortest path from B_1 to D and the shortest path from B_2 to D, we can make a globally optimal choice at A, thus directly compute the shortest path from A to D.

Dynamic programming

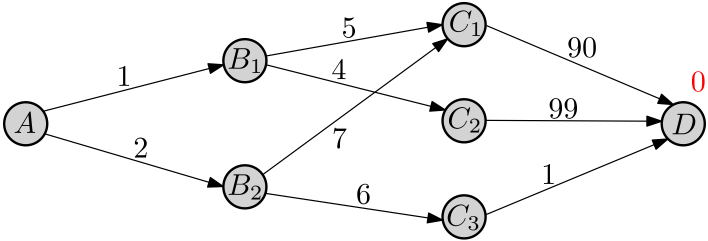
- Any idea to solve the problem?

Dynamic programming

- Any idea to solve the problem?
- Compute the shortest paths to D from right to left!

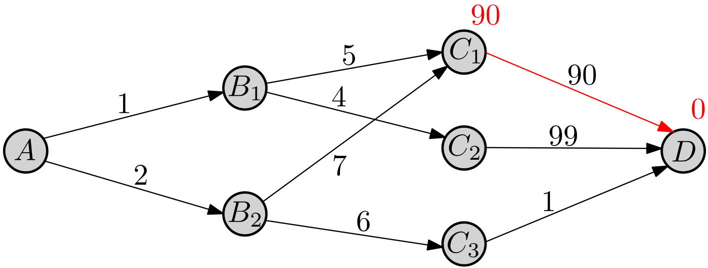
Dynamic programming

- Any idea to solve the problem?
- Compute the shortest paths to D from right to left!



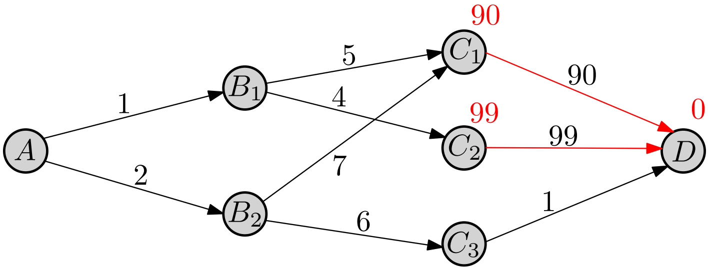
Dynamic programming

- Any idea to solve the problem?
- Compute the shortest paths to D from right to left!

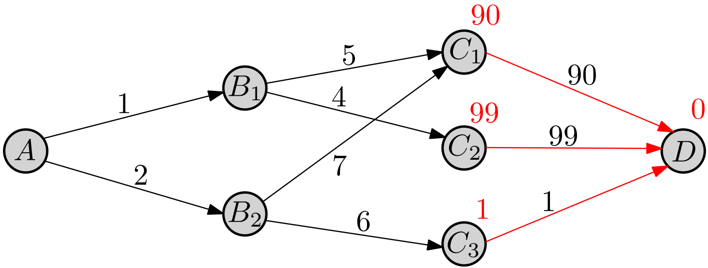


Dynamic programming

- Any idea to solve the problem?
- Compute the shortest paths to D from right to left!

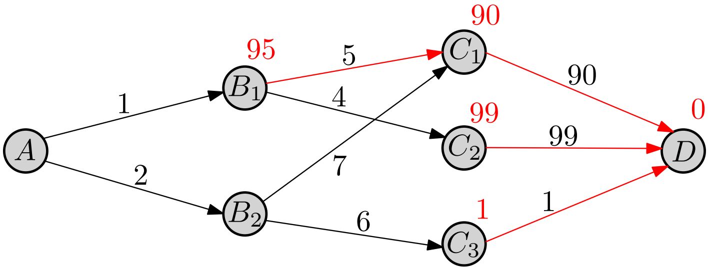


- Any idea to solve the problem?
- Compute the shortest paths to D from right to left!



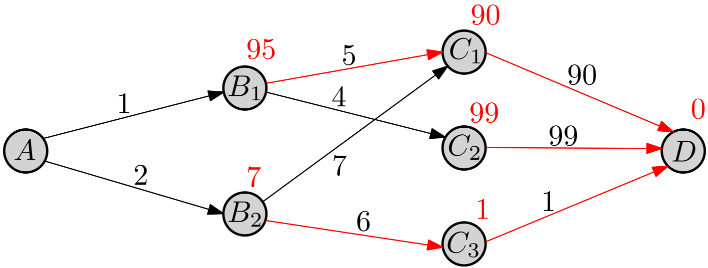
Dynamic programming

- Any idea to solve the problem?
- Compute the shortest paths to D from right to left!



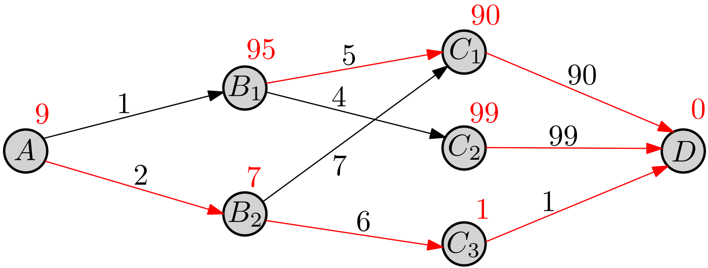
Dynamic programming

- Any idea to solve the problem?
- Compute the shortest paths to D from right to left!



Dynamic programming

- Any idea to solve the problem?
- Compute the shortest paths to D from right to left!



Dynamic programming

- **Abstract the idea**

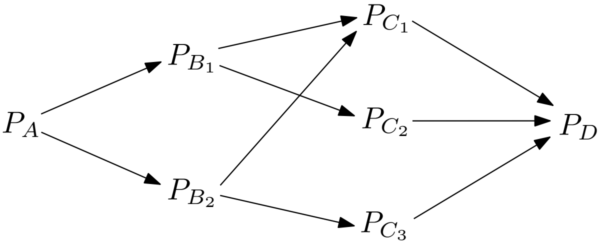
Dynamic programming

- **Abstract the idea**
- P_X = the subproblem of computing the shortest path $X \to D$

Dynamic programming

- **Abstract the idea**
- P_X = the subproblem of computing the shortest path $X \to D$
- The dependency among these subproblems is shown as follows.

- **Abstract the idea**
- P_X = the subproblem of computing the shortest path $X \to D$
- The dependency among these subproblems is shown as follows.



Dynamic programming

- **The framework of dynamic programming (DP)**

Dynamic programming

- **The framework of dynamic programming (DP)**
  1. Define a set $\mathcal{S}$ of subproblems with an acyclic dependency graph such that the optimal solution of a subproblem can be efficiently computed given the optimal solutions of all the subproblems it depends on.

- **The framework of dynamic programming (DP)**
  1. Define a set $\mathcal{S}$ of subproblems with an acyclic dependency graph such that the optimal solution of a subproblem can be efficiently computed given the optimal solutions of all the subproblems it depends on.
  2. Sort the subproblems in $\mathcal{S}$ as $P_1, \ldots, P_N$ (where $N = |\mathcal{S}|$) so that each subproblem P_i only depends on P_j for $j < i$.

Dynamic programming

- **The framework of dynamic programming (DP)**
  1. Define a set $\mathcal{S}$ of subproblems with an acyclic dependency graph such that the optimal solution of a subproblem can be efficiently computed given the optimal solutions of all the subproblems it depends on.
  2. Sort the subproblems in $\mathcal{S}$ as $P_1, \dots, P_N$ (where $N = |\mathcal{S}|$) so that each subproblem P_i only depends on P_j for $j < i$.
  3. Solve the subproblems $P_1, \dots, P_N$ iteratively, and then recover the final optimal solution from the optimal solutions of $P_1, \dots, P_N$.

Dynamic programming

- **The framework of dynamic programming (DP)**
  1. Define a set $\mathcal{S}$ of subproblems with an acyclic dependency graph such that the optimal solution of a subproblem can be efficiently computed given the optimal solutions of all the subproblems it depends on.
  2. Sort the subproblems in $\mathcal{S}$ as $P_1, \ldots, P_N$ (where $N = |\mathcal{S}|$) so that each subproblem P_i only depends on P_j for $j < i$.
  3. Solve the subproblems $P_1, \ldots, P_N$ iteratively, and then recover the final optimal solution from the optimal solutions of $P_1, \ldots, P_N$.
- Step 1 is the most technical and important step.

Dynamic programming

- **Dependency among the subproblems**
- Let $P \in \mathcal{S}$ and suppose P depends on $Q_1, \ldots, Q_r \in \mathcal{S}$.

- **Dependency among the subproblems**
- Let $P \in \mathcal{S}$ and suppose P depends on $Q_1, \ldots, Q_r \in \mathcal{S}$.
- Usually, the dependency looks like this...

- **Dependency among the subproblems**
- Let $P \in \mathcal{S}$ and suppose P depends on $Q_1, \ldots, Q_r \in \mathcal{S}$.
- Usually, the dependency looks like this...

  For each choice c of the first (or last) step to construct a solution of P, the optimal cost of a solution taking this choice can be expressed as a function $f_c(\text{opt}(Q_1), \ldots, \text{opt}(Q_r))$.

Dynamic programming

- **Dependency among the subproblems**
- Let $P \in \mathcal{S}$ and suppose P depends on $Q_1, \ldots, Q_r \in \mathcal{S}$.
- Usually, the dependency looks like this...

  For each choice c of the first (or last) step to construct a solution of P, the optimal cost of a solution taking this choice can be expressed as a function $f_c(\text{opt}(Q_1), \ldots, \text{opt}(Q_r))$.
- $C(P)$ = the set of choices of the first (or last) step for P

Dynamic programming

- **Dependency among the subproblems**
- Let $P \in \mathcal{S}$ and suppose P depends on $Q_1, \ldots, Q_r \in \mathcal{S}$.
- Usually, the dependency looks like this...

  For each choice c of the first (or last) step to construct a solution of P, the optimal cost of a solution taking this choice can be expressed as a function $f_c(\text{opt}(Q_1), \ldots, \text{opt}(Q_r))$.
- $C(P)$ = the set of choices of the first (or last) step for P
- $\text{opt}(P) = \min_{c \in C(P)} f_c(\text{opt}(Q_1), \ldots, \text{opt}(Q_r))$ for minimization.
  $\text{opt}(P) = \max_{c \in C(P)} f_c(\text{opt}(Q_1), \ldots, \text{opt}(Q_r))$ for maximization.

Dynamic programming

- **A generalized version of the shortest-path problem**

- **A generalized version of the shortest-path problem**

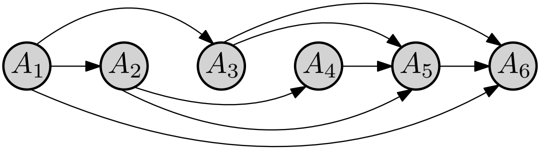
Problem (shortest path)

Suppose there are n cities $A_1, \ldots, A_n$ and a set E of m edges connecting these cities. Each path directs from a city A_i to another city A_j for $j > i$, and has a length. Our goal is to find a shortest path from A_1 to A_n.

- **A generalized version of the shortest-path problem**

Problem (shortest path)

Suppose there are n cities $A_1, \ldots, A_n$ and a set E of m edges connecting these cities. Each path directs from a city A_i to another city A_j for $j > i$, and has a length. Our goal is to find a shortest path from A_1 to A_n.



Dynamic programming

- We can solve the problem using DP as before.

Dynamic programming

- We can solve the problem using DP as before.
- P_i = computing the shortest path $A_1 \to A_i$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$

Dynamic programming

- We can solve the problem using DP as before.
- P_i = computing the shortest path $A_1 \to A_i$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$
- Note that P_i only depends on P_j for $j < i$.

Dynamic programming

- We can solve the problem using DP as before.
- P_i = computing the shortest path $A_1 \to A_i$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$
- Note that P_i only depends on P_j for $j < i$.
- $\textsc{ShortestPath}(n, A_1, \ldots, A_n, E)$

  opt[1] $\leftarrow 0$
  for $i = 2, \ldots, n$ **do**
  $\quad E_i \leftarrow \{e \in E : \text{target}(e) = i\}$
  $\quad$ **if** $E_i = \emptyset$ **then** opt[i] $\leftarrow +\infty$
  $\quad$ **else** opt[i] $\leftarrow \min_{e \in E_i}(\text{length}(e) + \text{opt}[\text{source}(e)])$
  return opt[n]

Dynamic programming

- We can solve the problem using DP as before.
- P_i = computing the shortest path $A_1 \to A_i$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$
- Note that P_i only depends on P_j for $j < i$.
- $\textsc{ShortestPath}(n, A_1, \ldots, A_n, E)$

  $\text{opt}[1] \leftarrow 0$
  for $i = 2, \ldots, n$ **do**
  $\quad E_i \leftarrow \{e \in E : \text{target}(e) = i\}$
  $\quad$ **if** $E_i = \emptyset$ **then** $\text{opt}[i] \leftarrow +\infty$
  $\quad$ **else** $\text{opt}[i] \leftarrow \min_{e \in E_i}(\text{length}(e) + \text{opt}[\text{source}(e)])$
  return $\text{opt}[n]$
- Time complexity $= O(n^2)$ or $O(n + |E|)$ if implemented carefully

Dynamic programming

- What if we use a recursive implementation?

Dynamic programming

- What if we use a recursive implementation?
- $\textsc{ShortestPath}(n, A_1, \ldots, A_n, E)$
  return $\textsc{Solve}(n, E)$
- $\textsc{Solve}(i, E)$
  if $i = 1$ **then return** 0
  $E_i \leftarrow \{e \in E : \text{target}(e) = i\}$
  return $\min_{e \in E_i}(\text{length}(e) + \textsc{Solve}(\text{source}(e), E))$

Dynamic programming

- What if we use a recursive implementation?
- $\textsc{ShortestPath}(n, A_1, \ldots, A_n, E)$
  return $\textsc{Solve}(n, E)$
- $\textsc{Solve}(i, E)$
  if $i = 1$ **then return** 0
  $E_i \leftarrow \{e \in E : \text{target}(e) = i\}$
  return $\min_{e \in E_i}(\text{length}(e) + \textsc{Solve}(\text{source}(e), E))$
- If we have edges between all $\binom{n}{2}$ pairs of cities...
  $T(n) \geq \sum_{i=1}^{n-1} T(i) \implies T(n) = \Omega(2^n)$

Dynamic programming

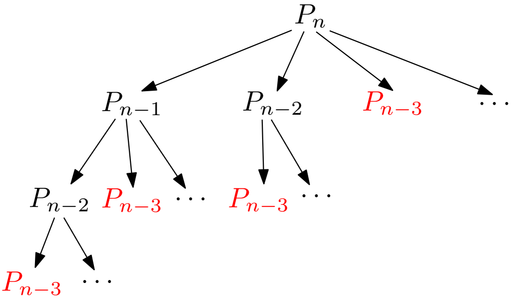
- Why is recursion much slower than iteration?

Dynamic programming

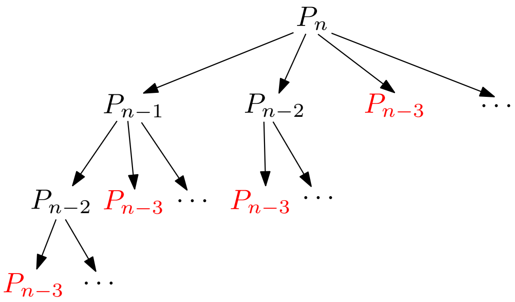
- Why is recursion much slower than iteration?
- Recursion may solve one subproblem in $\mathcal{S}$ many times!

Dynamic programming

- Why is recursion much slower than iteration?
- Recursion may solve one subproblem in $\mathcal{S}$ many times!



- Why is recursion much slower than iteration?
- Recursion may solve one subproblem in $\mathcal{S}$ many times!



- When we have $\binom{n}{2}$ edges, P_{n-3} is solved 4 times.

- **How to retrieve an optimal solution**

Dynamic programming

- **How to retrieve an optimal solution**
- $\textsc{ShortestPath}(n, A_1, \ldots, A_n, E)$

  $\text{opt}[1] \leftarrow 0$
  for $i = 2, \ldots, n$ **do**
  $\quad E_i \leftarrow \{e \in E : \text{target}(e) = i\}$
  $\quad$ **if** $E_i = \emptyset$ **then** $\text{opt}[i] \leftarrow +\infty$
  $\quad$ **else**
  $\quad\quad \text{opt}[i] \leftarrow \min_{e \in E_i}(\text{length}(e) + \text{opt}[\text{source}(e)])$
  $\quad\quad \text{last}[i] \leftarrow \arg\min_{e \in E_i}(\text{length}(e) + \text{opt}[\text{source}(e)])$
  $\text{path} \leftarrow [\,]$ and $i \leftarrow n$
  while $i > 1$ **do**
  $\quad \text{path} \leftarrow [\text{last}[i]] + \text{path}$
  $\quad i \leftarrow \text{source}(\text{last}[i])$
  return $(\text{opt}[n], \text{path})$

Maximum subarray

- A subarray of $A[1 \ldots n]$ is an array of the form $A[i \ldots j]$ for $i \leq j$.

Maximum subarray

- A subarray of $A[1 \dots n]$ is an array of the form $A[i \dots j]$ for $i \leq j$.

Problem (maximum subarray)

Given an array $A[1 \dots n]$, we want to design an algorithm that computes a subarray A' of A which maximizes $\mathrm{sum}(A')$, the sum of numbers in A'.

Maximum subarray

- A subarray of $A[1 \dots n]$ is an array of the form $A[i \dots j]$ for $i \leq j$.

Problem (maximum subarray)

Given an array $A[1 \dots n]$, we want to design an algorithm that computes a subarray A' of A which maximizes $\text{sum}(A')$, the sum of numbers in A'.

- Solve the problem by brute-force: $O(n^3)$ time

Maximum subarray

- A subarray of $A[1 \dots n]$ is an array of the form $A[i \dots j]$ for $i \leq j$.

Problem (maximum subarray)

Given an array $A[1 \dots n]$, we want to design an algorithm that computes a subarray A' of A which maximizes $\text{sum}(A')$, the sum of numbers in A'.

- Solve the problem by brute-force: $O(n^3)$ time
- Solve the problem by (improved) brute-force: $O(n^2)$ time

Maximum subarray

- A subarray of $A[1 \ldots n]$ is an array of the form $A[i \ldots j]$ for $i \leq j$.

Problem (maximum subarray)

Given an array $A[1 \ldots n]$, we want to design an algorithm that computes a subarray A' of A which maximizes $\mathrm{sum}(A')$, the sum of numbers in A'.

- Solve the problem by brute-force: $O(n^3)$ time
- Solve the problem by (improved) brute-force: $O(n^2)$ time
- Solve the problem by divide & conquer: $O(n \log n)$ time

Maximum subarray

- A subarray of $A[1 \ldots n]$ is an array of the form $A[i \ldots j]$ for $i \leq j$.

Problem (maximum subarray)

Given an array $A[1 \ldots n]$, we want to design an algorithm that computes a subarray A' of A which maximizes $\text{sum}(A')$, the sum of numbers in A'.

- Solve the problem by brute-force: $O(n^3)$ time
- Solve the problem by (improved) brute-force: $O(n^2)$ time
- Solve the problem by divide & conquer: $O(n \log n)$ time
- Can we do even better?

Maximum subarray

- How about using DP?

Maximum subarray

- How about using DP?
- In order to use DP, we need to define a set $\mathcal{S}$ of subproblems.

Maximum subarray

- How about using DP?
- In order to use DP, we need to define a set $\mathcal{S}$ of subproblems.
- **First attempt**

  P_i = computing the maximum subarray of $A[1 \ldots i]$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$

Maximum subarray

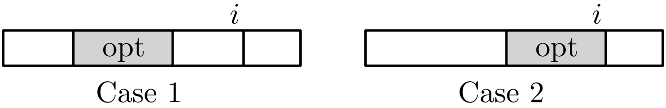
- How about using DP?
- In order to use DP, we need to define a set $\mathcal{S}$ of subproblems.
- **First attempt**

  $P_i =$ computing the maximum subarray of $A[1 \dots i]$
  $\mathcal{S} = \{P_1, \dots, P_n\}$
- Try to solve P_i using the optimal solutions of $P_1, \dots, P_{i-1}$?

Maximum subarray

- How about using DP?
- In order to use DP, we need to define a set $\mathcal{S}$ of subproblems.
- **First attempt**

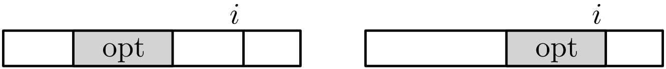
  P_i = computing the maximum subarray of $A[1 \ldots i]$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$
- Try to solve P_i using the optimal solutions of $P_1, \ldots, P_{i-1}$?



Maximum subarray

- How about using DP?
- In order to use DP, we need to define a set $\mathcal{S}$ of subproblems.
- **First attempt**

  P_i = computing the maximum subarray of $A[1 \ldots i]$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$
- Try to solve P_i using the optimal solutions of $P_1, \ldots, P_{i-1}$?



Case 1 Case 2

- Seems impossible... An optimal solution of P_i may end at $A[i]$, and in this case the optimal solutions of $P_1, \ldots, P_{i-1}$ are not helpful.

Maximum subarray

- **Second attempt**

  P_i = computing the maximum subarray of A ending at $A[i]$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$

Maximum subarray

- **Second attempt**

  P_i = computing the maximum subarray of A ending at $A[i]$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$

- From the optimal solutions of $P_1, \ldots, P_n$, we can obtain the optimal solution of the entire problem.

Maximum subarray

- **Second attempt**

  P_i = computing the maximum subarray of A ending at $A[i]$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$
- From the optimal solutions of $P_1, \ldots, P_n$, we can obtain the optimal solution of the entire problem.
- Dependency among $P_1, \ldots, P_n$?

Maximum subarray

- **Second attempt**

  P_i = computing the maximum subarray of A ending at $A[i]$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$

- From the optimal solutions of $P_1, \ldots, P_n$, we can obtain the optimal solution of the entire problem.

- Dependency among $P_1, \ldots, P_n$?

- Consider a subproblem P_i. Construct a solution of P_i step by step?

Maximum subarray

- **Second attempt**

  P_i = computing the maximum subarray of A ending at $A[i]$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$

- From the optimal solutions of $P_1, \ldots, P_n$, we can obtain the optimal solution of the entire problem.

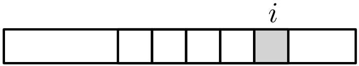
- Dependency among $P_1, \ldots, P_n$?

- Consider a subproblem P_i. Construct a solution of P_i step by step?
  Start at $A[i]$, choose "go left" or "stop" in each step.

- **Second attempt**

  P_i = computing the maximum subarray of A ending at $A[i]$
  $\mathcal{S} = \{P_1, \ldots, P_n\}$
- From the optimal solutions of $P_1, \ldots, P_n$, we can obtain the optimal solution of the entire problem.
- Dependency among $P_1, \ldots, P_n$?
- Consider a subproblem P_i. Construct a solution of P_i step by step?

  Start at $A[i]$, choose "go left" or "stop" in each step.



Maximum subarray

- Consider the first step of constructing a solution of P_i.

Maximum subarray

- Consider the first step of constructing a solution of P_i.
- $C(P_i) = \{\text{“go left”}, \text{“stop”}\}$

Maximum subarray

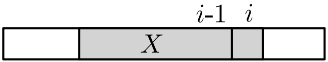
- Consider the first step of constructing a solution of P_i.
- $C(P_i) = \{\text{“go left”}, \text{“stop”}\}$
- If we choose “stop”, then the optimal score we can get is $A[i]$.

Maximum subarray

- Consider the first step of constructing a solution of P_i.
- $C(P_i) = \{\text{"go left"}, \text{"stop"}\}$
- If we choose "stop", then the optimal score we can get is $A[i]$.
- What if we choose "go left"? Our solution must contain $A[i-1]$.

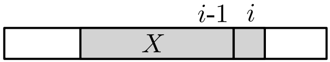
Maximum subarray

- Consider the first step of constructing a solution of P_i.
- $C(P_i) = \{$"go left", "stop"$\}$
- If we choose "stop", then the optimal score we can get is $A[i]$.
- What if we choose "go left"? Our solution must contain $A[i-1]$.



Maximum subarray

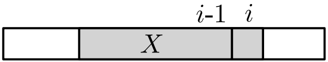
- Consider the first step of constructing a solution of P_i.
- $C(P_i) = \{$ "go left", "stop" $\}$
- If we choose "stop", then the optimal score we can get is $A[i]$.
- What if we choose "go left"? Our solution must contain $A[i-1]$.



- Our solution consists of $A[i]$ and a subarray X ending at $A[i-1]$.

Maximum subarray

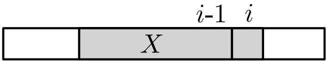
- Consider the first step of constructing a solution of P_i.
- $C(P_i) = \{$"go left", "stop"$\}$
- If we choose "stop", then the optimal score we can get is $A[i]$.
- What if we choose "go left"? Our solution must contain $A[i-1]$.



- Our solution consists of $A[i]$ and a subarray X ending at $A[i-1]$. Score $= \text{sum}(X) + A[i]$, maximized when $\text{sum}(X) = \text{opt}(P_{i-1})$.

Maximum subarray

- Consider the first step of constructing a solution of P_i.
- $C(P_i) = \{$"go left", "stop"$\}$
- If we choose "stop", then the optimal score we can get is $A[i]$.
- What if we choose "go left"? Our solution must contain $A[i-1]$.



- Our solution consists of $A[i]$ and a subarray X ending at $A[i-1]$. Score $= \mathrm{sum}(X) + A[i]$, maximized when $\mathrm{sum}(X) = \mathrm{opt}(P_{i-1})$.
- $\mathrm{opt}(P_i) = \max\{A[i], \mathrm{opt}(P_{i-1}) + A[i]\} = \max\{0, \mathrm{opt}(P_{i-1})\} + A[i]$

Maximum subarray

- $\textsc{MaxSubarray}(n, A)$

  $\text{opt}[1] \leftarrow A[1]$ and $C[1] \leftarrow$ "stop"
  for $i = 2, \ldots, n$ **do**
  $\quad \text{opt}[i] \leftarrow \max\{0, \text{opt}[i-1]\} + A[i]$
  $\quad$ **if** $0 < \text{opt}[i-1]$ **then** $C[i] \leftarrow$ "go left"
  $\quad$ **else** $C[i] \leftarrow$ "stop"
  $i^* \leftarrow \arg\max_{i \in \{1,\ldots,n\}} \text{opt}[i]$
  $j \leftarrow i^*$
  while $C[j] =$ "go left" **do**
  $\quad j \leftarrow j - 1$
  return $A[j \ldots i^*]$

Maximum subarray

- $\textsc{MaxSubarray}(n, A)$

  opt[1] $\leftarrow A[1]$ and $C[1] \leftarrow$ "stop"
  for $i = 2, \ldots, n$ **do**
      $\text{opt}[i] \leftarrow \max\{0, \text{opt}[i-1]\} + A[i]$
      if $0 < \text{opt}[i-1]$ **then** $C[i] \leftarrow$ "go left"
      else $C[i] \leftarrow$ "stop"
  $i^* \leftarrow \arg\max_{i \in \{1,\ldots,n\}} \text{opt}[i]$
  $j \leftarrow i^*$
  while $C[j] =$ "go left" **do**
      $j \leftarrow j - 1$
  return $A[j \ldots i^*]$

- Time complexity $= O(n)$

Weighted activity selection

Problem (weighted activity selection)

In NYUSH, there are n proposed activities $a_1, \ldots, a_n$, which wish to use the same room. Each activity a_i has a start time s_i, a finish time f_i, and a weight $w_i > 0$. We want to select a subset of compatible activities with maximum total weight.

Weighted activity selection

Problem (weighted activity selection)

In NYUSH, there are n proposed activities $a_1, \ldots, a_n$, which wish to use the same room. Each activity a_i has a start time s_i, a finish time f_i, and a weight $w_i > 0$. We want to select a subset of compatible activities with maximum total weight.

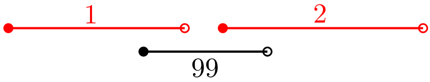
- We have seen the greedy algorithm for the unweighted version of activity selection fails for the weighted version.

Weighted activity selection

Problem (weighted activity selection)

In NYUSH, there are n proposed activities $a_1, \ldots, a_n$, which wish to use the same room. Each activity a_i has a start time s_i, a finish time f_i, and a weight $w_i > 0$. We want to select a subset of compatible activities with maximum total weight.

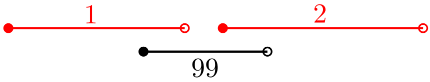
- We have seen the greedy algorithm for the unweighted version of activity selection fails for the weighted version.



Problem (weighted activity selection)

In NYUSH, there are n proposed activities $a_1, \ldots, a_n$, which wish to use the same room. Each activity a_i has a start time s_i, a finish time f_i, and a weight $w_i > 0$. We want to select a subset of compatible activities with maximum total weight.

- We have seen the greedy algorithm for the unweighted version of activity selection fails for the weighted version.



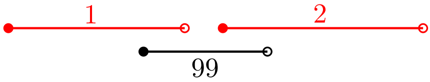
- Another greedy strategy: keep picking the maximum-weighted activity

Weighted activity selection

Problem (weighted activity selection)

In NYUSH, there are n proposed activities $a_1, \ldots, a_n$, which wish to use the same room. Each activity a_i has a start time s_i, a finish time f_i, and a weight $w_i > 0$. We want to select a subset of compatible activities with maximum total weight.

- We have seen the greedy algorithm for the unweighted version of activity selection fails for the weighted version.



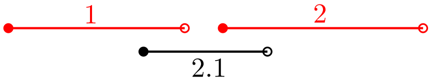
- Another greedy strategy: keep picking the maximum-weighted activity Does this strategy work?

Weighted activity selection

Problem (weighted activity selection)

In NYUSH, there are n proposed activities $a_1, \ldots, a_n$, which wish to use the same room. Each activity a_i has a start time s_i, a finish time f_i, and a weight $w_i > 0$. We want to select a subset of compatible activities with maximum total weight.

- We have seen the greedy algorithm for the unweighted version of activity selection fails for the weighted version.



- Another greedy strategy: keep picking the maximum-weighted activity Does this strategy work?

Weighted activity selection

- Now we try to solve this problem using DP.

Weighted activity selection

- Now we try to solve this problem using DP.
- First, let's look at a DP algorithm that is very inefficient.

Weighted activity selection

- Now we try to solve this problem using DP.
- First, let's look at a DP algorithm that is very inefficient.
- Let $A = \{a_1, \ldots, a_n\}$ be the set of activities.
  Define $A_i = \{a \in A : a \text{ is compatible with } a_i\}$.

Weighted activity selection

- Now we try to solve this problem using DP.
- First, let's look at a DP algorithm that is very inefficient.
- Let $A = \{a_1, \ldots, a_n\}$ be the set of activities.
  Define $A_i = \{a \in A : a \text{ is compatible with } a_i\}$.
- For each $S \subseteq A$, we define a subproblem P_S as selecting a subset of compatible activities in S with maximum total weight.

Weighted activity selection

- Now we try to solve this problem using DP.
- First, let's look at a DP algorithm that is very inefficient.
- Let $A = \{a_1, \ldots, a_n\}$ be the set of activities.
  Define $A_i = \{a \in A : a \text{ is compatible with } a_i\}$.
- For each $S \subseteq A$, we define a subproblem P_S as selecting a subset of compatible activities in S with maximum total weight.
- **How to solve** P_S (given the optimal solutions of $P_{S'}$ for all $S' \subsetneq S$)

Weighted activity selection

- Now we try to solve this problem using DP.
- First, let's look at a DP algorithm that is very inefficient.
- Let $A = \{a_1, \ldots, a_n\}$ be the set of activities.
  Define $A_i = \{a \in A : a \text{ is compatible with } a_i\}$.
- For each $S \subseteq A$, we define a subproblem P_S as selecting a subset of compatible activities in S with maximum total weight.
- **How to solve** P_S (given the optimal solutions of $P_{S'}$ for all $S' \subsetneq S$)
- $\text{opt}(P_S) = \max_{a_i \in S}(w_i + \text{opt}(P_{S \cap A_i}))$

Weighted activity selection

- Now we try to solve this problem using DP.
- First, let's look at a DP algorithm that is very inefficient.
- Let $A = \{a_1, \ldots, a_n\}$ be the set of activities.
  Define $A_i = \{a \in A : a \text{ is compatible with } a_i\}$.
- For each $S \subseteq A$, we define a subproblem P_S as selecting a subset of compatible activities in S with maximum total weight.
- **How to solve** P_S (given the optimal solutions of $P_{S'}$ for all $S' \subsetneq S$)
- $\text{opt}(P_S) = \max_{a_i \in S}(w_i + \text{opt}(P_{S \cap A_i}))$
- Time complexity? At least $\Omega(2^n)$.

Weighted activity selection

- Improve the time complexity?

Weighted activity selection

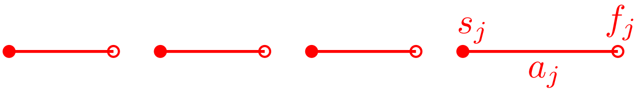
- Improve the time complexity?
- We consider the rightmost activity in our solution.

Weighted activity selection

- Improve the time complexity?
- We consider the rightmost activity in our solution.
- Let's sort the activities such that $f_1 \leq \cdots \leq f_n$.

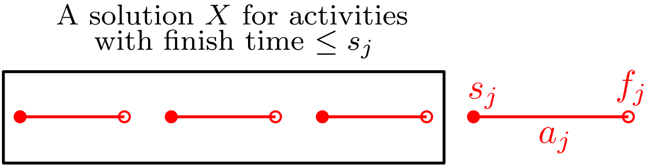
Weighted activity selection

- Improve the time complexity?
- We consider the rightmost activity in our solution.
- Let's sort the activities such that $f_1 \leq \cdots \leq f_n$.



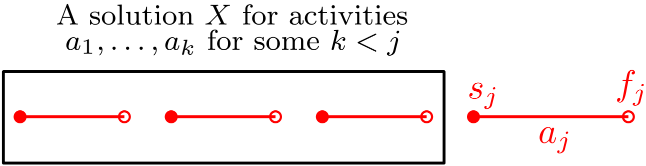
Weighted activity selection

- Improve the time complexity?
- We consider the rightmost activity in our solution.
- Let's sort the activities such that $f_1 \le \cdots \le f_n$.



Weighted activity selection

- Improve the time complexity?
- We consider the rightmost activity in our solution.
- Let's sort the activities such that $f_1 \leq \cdots \leq f_n$.



Weighted activity selection

- Let P_i be the subproblem of selecting a subset of compatible activities in $\{a_1, \ldots, a_i\}$ with maximum total weight, and $\mathcal{S} = \{P_1, \ldots, P_n\}$.

Weighted activity selection

- Let P_i be the subproblem of selecting a subset of compatible activities in $\{a_1, \ldots, a_i\}$ with maximum total weight, and $\mathcal{S} = \{P_1, \ldots, P_n\}$.
- P_i only depends on P_j for $j < i$.

Weighted activity selection

- Let P_i be the subproblem of selecting a subset of compatible activities in $\{a_1, \ldots, a_i\}$ with maximum total weight, and $\mathcal{S} = \{P_1, \ldots, P_n\}$.
- P_i only depends on P_j for $j < i$.
- $\text{pre}(j) = \max\{k : f_k \leq s_j\}$, then the activities in A compatible with a_j and to the left of a_j are exactly $a_1, \ldots, a_{\text{pre}(j)}$.

Weighted activity selection

- Let P_i be the subproblem of selecting a subset of compatible activities in $\{a_1, \ldots, a_i\}$ with maximum total weight, and $\mathcal{S} = \{P_1, \ldots, P_n\}$.
- P_i only depends on P_j for $j < i$.
- $\text{pre}(j) = \max\{k : f_k \leq s_j\}$, then the activities in A compatible with a_j and to the left of a_j are exactly $a_1, \ldots, a_{\text{pre}(j)}$.
- **How to solve** P_i

  Determine the rightmost activity in the solution.

Weighted activity selection

- Let P_i be the subproblem of selecting a subset of compatible activities in $\{a_1, \ldots, a_i\}$ with maximum total weight, and $\mathcal{S} = \{P_1, \ldots, P_n\}$.
- P_i only depends on P_j for $j < i$.
- $\text{pre}(j) = \max\{k : f_k \leq s_j\}$, then the activities in A compatible with a_j and to the left of a_j are exactly $a_1, \ldots, a_{\text{pre}(j)}$.
- **How to solve** P_i
  Determine the rightmost activity in the solution.
- $C(P_i) = \{1, \ldots, i\}$
  Choosing $j \in C(P_i)$ means a_j is the rightmost activity in the solution.

Weighted activity selection

- If we choose a_j as the rightmost activity, then our solution consists of a_j and a solution X of $P_{\text{pre}(j)}$, and its weight is $w_j + \text{weight}(X)$.

Weighted activity selection

- If we choose a_j as the rightmost activity, then our solution consists of a_j and a solution X of $P_{\text{pre}(j)}$, and its weight is $w_j + \text{weight}(X)$.
- To maximize this weight, X must be an optimal solution of $P_{\text{pre}(j)}$.

Weighted activity selection

- If we choose a_j as the rightmost activity, then our solution consists of a_j and a solution X of $P_{\mathrm{pre}(j)}$, and its weight is $w_j + \mathrm{weight}(X)$.
- To maximize this weight, X must be an optimal solution of $P_{\mathrm{pre}(j)}$.
- $\mathrm{opt}(P_i) = \max_{j \in C(P_i)}(w_j + \mathrm{opt}(P_{\mathrm{pre}(j)}))$

Weighted activity selection

- If we choose a_j as the rightmost activity, then our solution consists of a_j and a solution X of $P_{\text{pre}(j)}$, and its weight is $w_j + \text{weight}(X)$.
- To maximize this weight, X must be an optimal solution of $P_{\text{pre}(j)}$.
- $\text{opt}(P_i) = \max_{j \in C(P_i)}(w_j + \text{opt}(P_{\text{pre}(j)}))$
- Time for computing $\text{opt}(P_1), \ldots, \text{opt}(P_n)$?
  Straightforward way takes $O(n^2)$ time.

Weighted activity selection

- If we choose a_j as the rightmost activity, then our solution consists of a_j and a solution X of $P_{\mathrm{pre}(j)}$, and its weight is $w_j + \mathrm{weight}(X)$.
- To maximize this weight, X must be an optimal solution of $P_{\mathrm{pre}(j)}$.
- $\mathrm{opt}(P_i) = \max_{j \in C(P_i)}(w_j + \mathrm{opt}(P_{\mathrm{pre}(j)}))$
- Time for computing $\mathrm{opt}(P_1), \ldots, \mathrm{opt}(P_n)$?
  Straightforward way takes $O(n^2)$ time.
- Can we do this more efficiently?

Weighted activity selection

- To this end, we first simplify the expression for $\text{opt}(P_i)$.

Weighted activity selection

- To this end, we first simplify the expression for $\text{opt}(P_i)$.
- $\text{opt}(P_i) = \max_{j \in C(P_i)}(w_j + \text{opt}(P_{\text{pre}(j)})) = \max_{j=1}^{i}(w_j + \text{opt}(P_{\text{pre}(j)}))$

Weighted activity selection

- To this end, we first simplify the expression for $\text{opt}(P_i)$.
- $\text{opt}(P_i) = \max_{j \in C(P_i)}(w_j + \text{opt}(P_{\text{pre}(j)})) = \max_{j=1}^{i}(w_j + \text{opt}(P_{\text{pre}(j)}))$
  $\text{opt}(P_{i-1}) = \max_{j=1}^{i-1}(w_j + \text{opt}(P_{\text{pre}(j)}))$

Weighted activity selection

- To this end, we first simplify the expression for $\mathrm{opt}(P_i)$.
- $\mathrm{opt}(P_i) = \max_{j \in C(P_i)}(w_j + \mathrm{opt}(P_{\mathrm{pre}(j)})) = \max_{j=1}^{i}(w_j + \mathrm{opt}(P_{\mathrm{pre}(j)}))$

  $\mathrm{opt}(P_{i-1}) = \max_{j=1}^{i-1}(w_j + \mathrm{opt}(P_{\mathrm{pre}(j)}))$

  $\Longrightarrow \mathrm{opt}(P_i) = \max\{\mathrm{opt}(P_{i-1}), w_i + \mathrm{opt}(P_{\mathrm{pre}(i)})\}$

Weighted activity selection

- To this end, we first simplify the expression for $\text{opt}(P_i)$.
- $\text{opt}(P_i) = \max_{j \in C(P_i)}(w_j + \text{opt}(P_{\text{pre}(j)})) = \max_{j=1}^{i}(w_j + \text{opt}(P_{\text{pre}(j)}))$
  $\text{opt}(P_{i-1}) = \max_{j=1}^{i-1}(w_j + \text{opt}(P_{\text{pre}(j)}))$
  $\Longrightarrow \text{opt}(P_i) = \max\{\text{opt}(P_{i-1}), w_i + \text{opt}(P_{\text{pre}(i)})\}$
- If we know $\text{pre}(1), \ldots, \text{pre}(n)$, we can compute $\text{opt}(P_1), \ldots, \text{opt}(P_n)$ in $O(n)$ time iteratively.

Weighted activity selection

- To this end, we first simplify the expression for $\mathrm{opt}(P_i)$.
- $\mathrm{opt}(P_i) = \max_{j \in C(P_i)}(w_j + \mathrm{opt}(P_{\mathrm{pre}(j)})) = \max_{j=1}^{i}(w_j + \mathrm{opt}(P_{\mathrm{pre}(j)}))$
  $\mathrm{opt}(P_{i-1}) = \max_{j=1}^{i-1}(w_j + \mathrm{opt}(P_{\mathrm{pre}(j)}))$
  $\Longrightarrow \mathrm{opt}(P_i) = \max\{\mathrm{opt}(P_{i-1}), w_i + \mathrm{opt}(P_{\mathrm{pre}(i)})\}$
- If we know $\mathrm{pre}(1), \ldots, \mathrm{pre}(n)$, we can compute $\mathrm{opt}(P_1), \ldots, \mathrm{opt}(P_n)$ in $O(n)$ time iteratively.
- **Last question:** how to compute $\mathrm{pre}(1), \ldots, \mathrm{pre}(n)$

Weighted activity selection

- To this end, we first simplify the expression for $\mathrm{opt}(P_i)$.
- $\mathrm{opt}(P_i) = \max_{j \in C(P_i)}(w_j + \mathrm{opt}(P_{\mathrm{pre}(j)})) = \max_{j=1}^{i}(w_j + \mathrm{opt}(P_{\mathrm{pre}(j)}))$
  $\mathrm{opt}(P_{i-1}) = \max_{j=1}^{i-1}(w_j + \mathrm{opt}(P_{\mathrm{pre}(j)}))$
  $\Longrightarrow \mathrm{opt}(P_i) = \max\{\mathrm{opt}(P_{i-1}), w_i + \mathrm{opt}(P_{\mathrm{pre}(i)})\}$
- If we know $\mathrm{pre}(1), \ldots, \mathrm{pre}(n)$, we can compute $\mathrm{opt}(P_1), \ldots, \mathrm{opt}(P_n)$ in $O(n)$ time iteratively.
- **Last question:** how to compute $\mathrm{pre}(1), \ldots, \mathrm{pre}(n)$
- Recall that $\mathrm{pre}(j) = \max\{k : f_k \leq s_j\}$.

Weighted activity selection

- To this end, we first simplify the expression for $\text{opt}(P_i)$.
- $\text{opt}(P_i) = \max_{j \in C(P_i)}(w_j + \text{opt}(P_{\text{pre}(j)})) = \max_{j=1}^{i}(w_j + \text{opt}(P_{\text{pre}(j)}))$

  $\text{opt}(P_{i-1}) = \max_{j=1}^{i-1}(w_j + \text{opt}(P_{\text{pre}(j)}))$

  $\implies \text{opt}(P_i) = \max\{\text{opt}(P_{i-1}), w_i + \text{opt}(P_{\text{pre}(i)})\}$
- If we know $\text{pre}(1), \ldots, \text{pre}(n)$, we can compute $\text{opt}(P_1), \ldots, \text{opt}(P_n)$ in $O(n)$ time iteratively.
- **Last question:** how to compute $\text{pre}(1), \ldots, \text{pre}(n)$
- Recall that $\text{pre}(j) = \max\{k : f_k \leq s_j\}$. Since $f_1 < \cdots < f_n$, we can use binary search to compute $\text{pre}(j)$ in $O(\log n)$ time.

Weighted activity selection

- $\textsc{WeightedActSelect}(n, S, F, W)$

  Sort the activities such that $F[1] \leq \cdots \leq F[n]$

  $F[0] \leftarrow -\infty$

  for $j = 1, \ldots, n$ **do**

      $\text{pre}[j] \leftarrow$ largest k s.t. $F[k] \leq S[j]$ $\quad\triangleright$ done in $O(\log n)$ time

  $\text{opt}[0] \leftarrow 0$

  for $i = 1, \ldots, n$ **do**

      $\text{opt}[i] \leftarrow \max\{\text{opt}[i-1], W[i] + \text{opt}[\text{pre}[i]]\}$

  return $\text{opt}[n]$

Weighted activity selection

- $\textsc{WeightedActSelect}(n, S, F, W)$

  Sort the activities such that $F[1] \leq \cdots \leq F[n]$

  $F[0] \leftarrow -\infty$

  for $j = 1, \ldots, n$ **do**

      $\text{pre}[j] \leftarrow$ largest k s.t. $F[k] \leq S[j]$ $\quad\triangleright$ done in $O(\log n)$ time

  $\text{opt}[0] \leftarrow 0$

  for $i = 1, \ldots, n$ **do**

      $\text{opt}[i] \leftarrow \max\{\text{opt}[i-1], W[i] + \text{opt}[\text{pre}[i]]\}$

  return $\text{opt}[n]$

- The above code only returns the optimum.
  Try to figure out by yourself how to retrieve an solution!

Weighted activity selection

- $\textsc{WeightedActSelect}(n, S, F, W)$

  Sort the activities such that $F[1] \leq \cdots \leq F[n]$

  $F[0] \leftarrow -\infty$

  for $j = 1, \ldots, n$ **do**

      $\text{pre}[j] \leftarrow$ largest k s.t. $F[k] \leq S[j]$ $\qquad \triangleright$ done in $O(\log n)$ time

  $\text{opt}[0] \leftarrow 0$

  for $i = 1, \ldots, n$ **do**

      $\text{opt}[i] \leftarrow \max\{\text{opt}[i-1], W[i] + \text{opt}[\text{pre}[i]]\}$

  return $\text{opt}[n]$

- The above code only returns the optimum.
  Try to figure out by yourself how to retrieve an solution!

- Time complexity $= O(n \log n)$

Longest increasing subsequence

- A subsequence of $A[1 \ldots n]$ is an array $B[1 \ldots m]$ for $m \leq n$ such that there exists $i_1 < \cdots < i_m$ satisfying $B[j] = A[i_j]$ for $j \in \{1, \ldots, m\}$.

Longest increasing subsequence

- A subsequence of $A[1 \dots n]$ is an array $B[1 \dots m]$ for $m \leq n$ such that there exists $i_1 < \cdots < i_m$ satisfying $B[j] = A[i_j]$ for $j \in \{1, \dots, m\}$.
- A sequence $B[1 \dots m]$ is increasing if $B[1] < \cdots < B[m]$.

Longest increasing subsequence

- A subsequence of $A[1 \dots n]$ is an array $B[1 \dots m]$ for $m \leq n$ such that there exists $i_1 < \cdots < i_m$ satisfying $B[j] = A[i_j]$ for $j \in \{1, \dots, m\}$.
- A sequence $B[1 \dots m]$ is increasing if $B[1] < \cdots < B[m]$.

Problem (longest increasing subsequence)

Given an array $A[1 \dots n]$, we want to design an algorithm that computes a longest subsequence B that is increasing.

Longest increasing subsequence

- A subsequence of $A[1 \dots n]$ is an array $B[1 \dots m]$ for $m \leq n$ such that there exists $i_1 < \cdots < i_m$ satisfying $B[j] = A[i_j]$ for $j \in \{1, \dots, m\}$.
- A sequence $B[1 \dots m]$ is increasing if $B[1] < \cdots < B[m]$.

Problem (longest increasing subsequence)

Given an array $A[1 \dots n]$, we want to design an algorithm that computes a longest subsequence B that is increasing.

- **Example**

6	3	7	5	6	1	9	18	14	4	15

Longest increasing subsequence

- Brute-force? A has 2^n subsequences!

Longest increasing subsequence

- Brute-force? A has 2^n subsequences!
- Greedy? Two reasonable (but incorrect) greedy strategies:
  - Always take the leftmost element that can be added to the end of B.
  - Always take the smallest element that can be added to the end of B.

Longest increasing subsequence

- Brute-force? A has 2^n subsequences!
- Greedy? Two reasonable (but incorrect) greedy strategies:
  - Always take the leftmost element that can be added to the end of B.
  - Always take the smallest element that can be added to the end of B.
- Now it's a good time to try DP.

Longest increasing subsequence

- Brute-force? A has 2^n subsequences!
- Greedy? Two reasonable (but incorrect) greedy strategies:
  - Always take the leftmost element that can be added to the end of B.
  - Always take the smallest element that can be added to the end of B.
- Now it's a good time to try DP.
- What if we define the subproblems as computing an LIS of $A[1 \ldots i]$? Not a good idea (same reason as in the maximum subarray problem).

Longest increasing subsequence

- Brute-force? A has 2^n subsequences!
- Greedy? Two reasonable (but incorrect) greedy strategies:
  - Always take the leftmost element that can be added to the end of B.
  - Always take the smallest element that can be added to the end of B.
- Now it's a good time to try DP.
- What if we define the subproblems as computing an LIS of $A[1 \dots i]$?
  Not a good idea (same reason as in the maximum subarray problem).
- P_i = computing an LIS of A ending at $A[i]$
  $\mathcal{S} = \{P_1, \dots, P_n\}$

Longest increasing subsequence

- Brute-force? A has 2^n subsequences!
- Greedy? Two reasonable (but incorrect) greedy strategies:
  - Always take the leftmost element that can be added to the end of B.
  - Always take the smallest element that can be added to the end of B.
- Now it's a good time to try DP.
- What if we define the subproblems as computing an LIS of $A[1 \dots i]$? Not a good idea (same reason as in the maximum subarray problem).
- $P_i =$ computing an LIS of A ending at $A[i]$
  $\mathcal{S} = \{P_1, \dots, P_n\}$
- Dependency among $P_1, \dots, P_n$?

Longest increasing subsequence

- Consider a subproblem P_i. Construct a solution of P_i step by step?

Longest increasing subsequence

- Consider a subproblem P_i. Construct a solution of P_i step by step? Start at $A[i]$, determine the solution from right to left.

Longest increasing subsequence

- Consider a subproblem P_i. Construct a solution of P_i step by step? Start at $A[i]$, determine the solution from right to left.
- The first step: determine the second rightmost number

Longest increasing subsequence

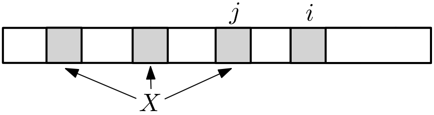
- Consider a subproblem P_i. Construct a solution of P_i step by step? Start at $A[i]$, determine the solution from right to left.
- The first step: determine the second rightmost number
- $C(P_i) = \{j < i : A[j] < A[i]\}$

Longest increasing subsequence

- Consider a subproblem P_i. Construct a solution of P_i step by step? Start at $A[i]$, determine the solution from right to left.
- The first step: determine the second rightmost number
- $C(P_i) = \{j < i : A[j] < A[i]\}$
- If we choose $A[j]$ for some $j \in C(P_i)$...

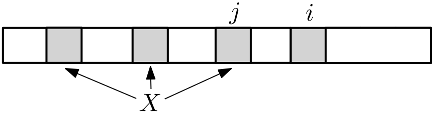
Longest increasing subsequence

- Consider a subproblem P_i. Construct a solution of P_i step by step? Start at $A[i]$, determine the solution from right to left.
- The first step: determine the second rightmost number
- $C(P_i) = \{j < i : A[j] < A[i]\}$
- If we choose $A[j]$ for some $j \in C(P_i)$...



Longest increasing subsequence

- Consider a subproblem P_i. Construct a solution of P_i step by step? Start at $A[i]$, determine the solution from right to left.
- The first step: determine the second rightmost number
- $C(P_i) = \{j < i : A[j] < A[i]\}$
- If we choose $A[j]$ for some $j \in C(P_i)$...



- The solution consists of $A[i]$ and an IS X of A ending at $A[j]$. Score $=$ lengt$(X) + 1$, maximized when lengt$(X) =$ opt(P_j).

Longest increasing subsequence

- $\mathrm{opt}(P_i) = \max_{j \in C(P_i)} (\mathrm{opt}(P_j) + 1) = \max_{j \in C(P_i)} \mathrm{opt}(P_j) + 1$

Longest increasing subsequence

- $\text{opt}(P_i) = \max_{j \in C(P_i)}(\text{opt}(P_j) + 1) = \max_{j \in C(P_i)} \text{opt}(P_j) + 1$
- $\text{LIS}(n, A)$

for $i = 1, \ldots, n$ **do**
  $m \leftarrow 0$ and $k \leftarrow 0$
  for $j = 1, \ldots, i-1$ **do**
    if $A[j] < A[i]$ and $\text{opt}[j] > m$ **then**
      $m \leftarrow \text{opt}[j]$ and $k \leftarrow j$
  $\text{opt}[i] \leftarrow m + 1$ and $\text{pos}[i] \leftarrow j$
$i^* = \arg\max_{i \in \{1,\ldots,n\}} \text{opt}[i]$
$i \leftarrow i^*$ and $B \leftarrow [A[i]]$
while $\text{pos}[i] > 0$ **do**
  $i \leftarrow \text{pos}[i]$ and $B \leftarrow [A[i]] + B$
return B

Longest common subsequence

- A sequence C is a common subsequence of two sequences A and B if C is a subsequence of both A and B.

Longest common subsequence

- A sequence C is a common subsequence of two sequences A and B if C is a subsequence of both A and B.

Problem (longest common subsequence)

Given arrays $A[1 \dots n_A]$ and $B[1 \dots n_B]$, we want to design an algorithm that computes a longest common subsequence C of A and B.

Longest common subsequence

- A sequence C is a common subsequence of two sequences A and B if C is a subsequence of both A and B.

Problem (longest common subsequence)

Given arrays $A[1 \ldots n_A]$ and $B[1 \ldots n_B]$, we want to design an algorithm that computes a longest common subsequence C of A and B.

- **Example**

Z	C	B	A	C	C	K	X	U	L	P

C	K	A	T	U	B	C	X	P	A	Z

Longest common subsequence

- Let's consider an easy case...
  Assume the elements in either of A and B are distinct.

Longest common subsequence

- Let's consider an easy case...
  Assume the elements in either of A and B are distinct.
- Solve the problem under this assumption?

Longest common subsequence

- Let's consider an easy case...
  Assume the elements in either of A and B are distinct.
- Solve the problem under this assumption?
  1. Remove the elements of A that do not appear in B.
     Remove the elements of B that do not appear in A.
     Now the elements of A one-to-one correspond to the elements of B.

Longest common subsequence

- Let's consider an easy case...
  Assume the elements in either of A and B are distinct.
- Solve the problem under this assumption?
  1. Remove the elements of A that do not appear in B.
     Remove the elements of B that do not appear in A.
     Now the elements of A one-to-one correspond to the elements of B.
  2. Create an array A' where $A'[i]$ stores the index of $A[i]$ in B.
     Consider a common subsequence $A[i_1, \ldots, i_m] = B[j_1, \ldots, j_m]$ of A, B and observe the subsequence $A'[i_1, \ldots, i_m]$ of A'. What do you find?

Longest common subsequence

- Let's consider an easy case...
  Assume the elements in either of A and B are distinct.
- Solve the problem under this assumption?
  1. Remove the elements of A that do not appear in B.
     Remove the elements of B that do not appear in A.
     Now the elements of A one-to-one correspond to the elements of B.
  2. Create an array A' where $A'[i]$ stores the index of $A[i]$ in B.
     Consider a common subsequence $A[i_1, \dots, i_m] = B[j_1, \dots, j_m]$ of A, B and observe the subsequence $A'[i_1, \dots, i_m]$ of A'. What do you find?
  3. Compute an LIS of A', which corresponds to an LCS of A and B.

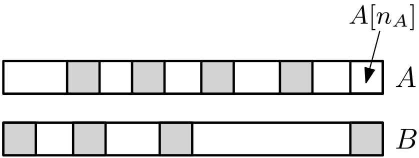
Longest common subsequence

- Now we consider the general case.

Longest common subsequence

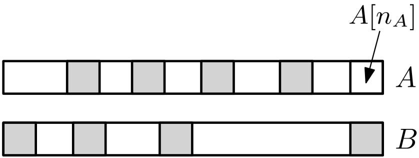
- Now we consider the general case.
- Imagine an LCS C of A and B. There can be three cases for C.

- Now we consider the general case.
- Imagine an LCS C of A and B. There can be three cases for C.
- **Case 1.** C doesn't contain $A[n_A]$.



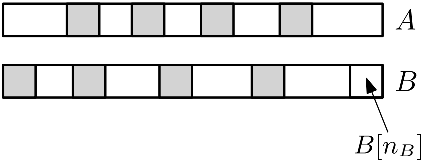
Longest common subsequence

- Now we consider the general case.
- Imagine an LCS C of A and B. There can be three cases for C.
- **Case 1.** C doesn't contain $A[n_A]$.



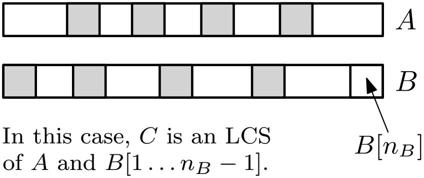
In this case, C is an LCS of $A[1 \ldots n_A - 1]$ and B.

- Now we consider the general case.
- Imagine an LCS C of A and B. There can be three cases for C.
- **Case 2.** C doesn't contain $B[n_B]$.



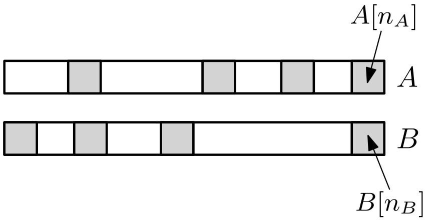
Longest common subsequence

- Now we consider the general case.
- Imagine an LCS C of A and B. There can be three cases for C.
- **Case 2.** C doesn't contain $B[n_B]$.



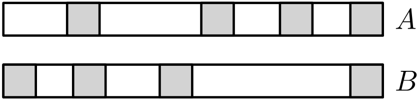
Longest common subsequence

- Now we consider the general case.
- Imagine an LCS C of A and B. There can be three cases for C.
- **Case 3.** C contains both $A[n_A]$ and $B[n_B]$.



Longest common subsequence

- Now we consider the general case.
- Imagine an LCS C of A and B. There can be three cases for C.
- **Case 3.** C contains both $A[n_A]$ and $B[n_B]$.



In this case, C consists of $A[n_A] = B[n_B]$
and an LCS of $A[1 \dots n_A\text{-}1]$ and $B[1 \dots n_B\text{-}1]$.

Longest common subsequence

- $P_{i,j}$ = computing an LCS of $A[1 \ldots i]$ and $B[1 \ldots j]$.
  $\mathcal{S} = \{P_{i,j} : i \in \{0, 1, \ldots, n_A\}$ and $j \in \{0, 1, \ldots, n_B\}\}$.

Longest common subsequence

- $P_{i,j}$ = computing an LCS of $A[1 \ldots i]$ and $B[1 \ldots j]$.
  $\mathcal{S} = \{P_{i,j} : i \in \{0, 1, \ldots, n_A\}$ and $j \in \{0, 1, \ldots, n_B\}\}$.
- $P_{i,j}$ depends on $P_{i-1,j}$, $P_{i,j-1}$, and $P_{i-1,j-1}$ (for $i, j \geq 1$).

Longest common subsequence

- $P_{i,j}$ = computing an LCS of $A[1 \ldots i]$ and $B[1 \ldots j]$.
  $\mathcal{S} = \{P_{i,j} : i \in \{0, 1, \ldots, n_A\} \text{ and } j \in \{0, 1, \ldots, n_B\}\}$.
- $P_{i,j}$ depends on $P_{i-1,j}$, $P_{i,j-1}$, and $P_{i-1,j-1}$ (for $i, j \geq 1$).
- If $A[i] = B[j]$, then we have
  $\text{opt}(P_{i,j}) = \max\{\text{opt}(P_{i-1,j}), \text{opt}(P_{i,j-1}), \text{opt}(P_{i-1,j-1}) + 1\}$.

- $P_{i,j}$ = computing an LCS of $A[1 \dots i]$ and $B[1 \dots j]$.
  $\mathcal{S} = \{P_{i,j} : i \in \{0, 1, \dots, n_A\}$ and $j \in \{0, 1, \dots, n_B\}\}$.
- $P_{i,j}$ depends on $P_{i-1,j}$, $P_{i,j-1}$, and $P_{i-1,j-1}$ (for $i, j \geq 1$).
- If $A[i] = B[j]$, then we have
  $\text{opt}(P_{i,j}) = \max\{\text{opt}(P_{i-1,j}), \text{opt}(P_{i,j-1}), \text{opt}(P_{i-1,j-1}) + 1\}$.
- If $A[i] \neq B[j]$, then we have
  $\text{opt}(P_{i,j}) = \max\{\text{opt}(P_{i-1,j}), \text{opt}(P_{i,j-1})\}$.

Longest common subsequence

- $P_{i,j}$ = computing an LCS of $A[1 \ldots i]$ and $B[1 \ldots j]$.
  $\mathcal{S} = \{P_{i,j} : i \in \{0, 1, \ldots, n_A\}$ and $j \in \{0, 1, \ldots, n_B\}\}$.
- $P_{i,j}$ depends on $P_{i-1,j}$, $P_{i,j-1}$, and $P_{i-1,j-1}$ (for $i, j \geq 1$).
- If $A[i] = B[j]$, then we have
  $\text{opt}(P_{i,j}) = \max\{\text{opt}(P_{i-1,j}), \text{opt}(P_{i,j-1}), \text{opt}(P_{i-1,j-1}) + 1\}$.
- If $A[i] \neq B[j]$, then we have
  $\text{opt}(P_{i,j}) = \max\{\text{opt}(P_{i-1,j}), \text{opt}(P_{i,j-1})\}$.
- **Boundary case**
  $\text{opt}(P_{i,0}) = \text{opt}(P_{0,j}) = 0$ for all i and all j

Longest common subsequence

- $\mathrm{LCS}(n_A, A, n_B, B)$

  opt$[i, 0] \leftarrow 0$ and opt$[0, j] \leftarrow 0$ for $i, j \in \{0, 1, \ldots, n\}$
  for $i = 1, \ldots, n_A$ **do**
      for $j = 1, \ldots, n_B$ **do**
          opt$[i, j] \leftarrow$ max$\{$opt$[i-1, j]$, opt$[i, j-1]\}$
          if $A[i] = B[j]$ and opt$[i, j] <$ opt$[i-1, j-1] + 1$ **then**
              opt$[i, j] \leftarrow$ opt$[i-1, j-1] + 1$
  return opt$[n_A, n_B]$

Longest common subsequence

- $\mathrm{LCS}(n_A, A, n_B, B)$

  opt$[i, 0] \leftarrow 0$ and opt$[0, j] \leftarrow 0$ for $i, j \in \{0, 1, \ldots, n\}$
  for $i = 1, \ldots, n_A$ **do**
      for $j = 1, \ldots, n_B$ **do**
          opt$[i, j] \leftarrow \max\{\text{opt}[i-1, j], \text{opt}[i, j-1]\}$
          if $A[i] = B[j]$ and opt$[i, j] <$ opt$[i-1, j-1] + 1$ **then**
              opt$[i, j] \leftarrow$ opt$[i-1, j-1] + 1$
  return opt$[n_A, n_B]$

- The above code only returns the optimum, i.e., the length of an LCS. Try to figure out by yourself how to retrieve an LCS!

Longest common subsequence

- $\mathrm{LCS}(n_A, A, n_B, B)$

  opt$[i, 0] \leftarrow 0$ and opt$[0, j] \leftarrow 0$ for $i, j \in \{0, 1, \ldots, n\}$
  for $i = 1, \ldots, n_A$ **do**
      for $j = 1, \ldots, n_B$ **do**
          opt$[i, j] \leftarrow \max\{\text{opt}[i-1, j], \text{opt}[i, j-1]\}$
          if $A[i] = B[j]$ and opt$[i, j] <$ opt$[i-1, j-1] + 1$ **then**
              opt$[i, j] \leftarrow$ opt$[i-1, j-1] + 1$
  return opt$[n_A, n_B]$

- The above code only returns the optimum, i.e., the length of an LCS. Try to figure out by yourself how to retrieve an LCS!
- Time complexity $= O(n_A n_B) = O(n^2)$ where $n = n_A + n_B$