ETH zürich

Exercise Session 11 – DP and Flow Algos

Data Structures and Algorithms

These slides are based on those of the lecture, but were adapted and extended by the teaching assistant Adel Gavranović

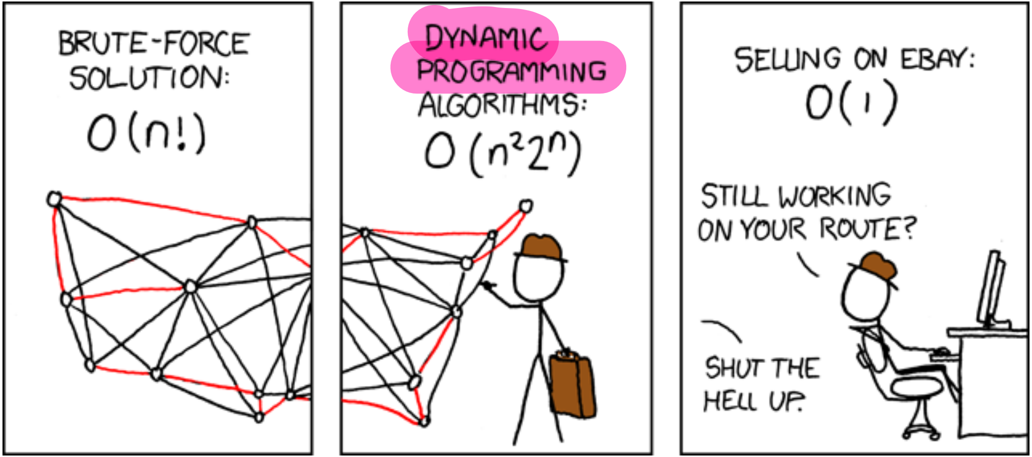
Today’s Schedule

Intro
Feedback for **code** expert
MaxFlow
Old Exam Questions (Max-Flow)
Dynamic Programming
Overlap of Convex Polygons
In-Class Code-Example
Outro

`n.ethz.ch/~agavranovic`

- Exercise Session Material
- Adel’s Webpage
- Mail to Adel

Comic of the Week



xkcd

1. Intro

Intro

Intro

- Often explaining stuff via email is suboptimal
- Consider going to the Study Center (especially if it’s related to exercises!)
  - Thursdays
  - 08:15 - 10:00
  - ML H 41.1

2. Feedback for **code** expert

General things regarding **code** expert

[1] your boi has his own exams

General things regarding **code** expert

- You can submit your partial solutions too!

[1]your boi has his own exams

General things regarding **code** expert

- You can submit your partial solutions too!
- If you want feedback, please make sure to indicate *what part* of your code you want me to have a closer look at or what *you think* the problem is

[1] your boi has his own exams

General things regarding **code** expert

- You can submit your partial solutions too!
- If you want feedback, please make sure to indicate *what part* of your code you want me to have a closer look at or what *you think* the problem is
- I'm not going to be very responsive in the Lernphase[1] so better ask now

[1] your boi has his own exams

General things regarding **code** expert

- You can submit your partial solutions too!
- If you want feedback, please make sure to indicate *what part* of your code you want me to have a closer look at or what *you think* the problem is
- I'm not going to be very responsive in the Lernphase[1] so better ask now
- Scores for exercises with (pseudo)random stuff can vary. So occasionally, it makes sense to just re-test the same code

[1] your boi has his own exams

Questions regarding **code** expert from your side?

3. MaxFlow

Flow

A **Flow** $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

Flow

A **Flow** $f : V \times V \rightarrow \mathbb{R}$ fulfills the following conditions:

- **Bounded Capacity**:

Flow

A **Flow** $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- **Bounded Capacity**:
  For all $u, v \in V$: $f(u, v) \leq c(u, v)$.

Flow

A **Flow** $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- **Bounded Capacity**:
  For all $u, v \in V$: $f(u, v) \leq c(u, v)$.
- **Skew Symmetry**:

Flow

A **Flow** $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- **Bounded Capacity**:
  For all $u, v \in V$: $f(u, v) \leq c(u, v)$.
- **Skew Symmetry**:
  For all $u, v \in V$: $f(u, v) = -f(v, u)$.

Flow

A **Flow** $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- **Bounded Capacity**:
  For all $u, v \in V$: $f(u, v) \leq c(u, v)$.
- **Skew Symmetry**:
  For all $u, v \in V$: $f(u, v) = -f(v, u)$.
- **Conservation of flow**:

Flow

A **Flow** $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- **Bounded Capacity**:
  For all $u, v \in V$: $f(u, v) \leq c(u, v)$.
- **Skew Symmetry**:
  For all $u, v \in V$: $f(u, v) = -f(v, u)$.
- **Conservation of flow**:
  For all $u \in V \setminus \{s, t\}$:

Flow

A **Flow** $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- **Bounded Capacity**:
  For all $u, v \in V$: $f(u,v) \leq c(u,v)$.
- **Skew Symmetry**:
  For all $u, v \in V$: $f(u,v) = -f(v,u)$.
- **Conservation of flow**:
  For all $u \in V \setminus \{s,t\}$:

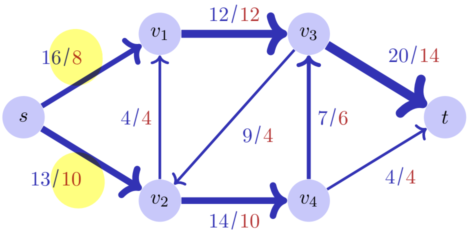
$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v).$$

Flow

A **Flow** $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- **Bounded Capacity**:
  For all $u, v \in V$: $f(u,v) \leq c(u,v)$.
- **Skew Symmetry**:
  For all $u, v \in V$: $f(u,v) = -f(v,u)$.
- **Conservation of flow**:
  For all $u \in V \setminus \{s,t\}$:

$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v).$$



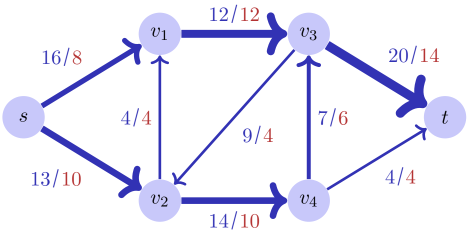
Value of the flow:

Flow

A **Flow** $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- **Bounded Capacity**:
  For all $u, v \in V$: $f(u,v) \leq c(u,v)$.
- **Skew Symmetry**:
  For all $u, v \in V$: $f(u,v) = -f(v,u)$.
- **Conservation of flow**:
  For all $u \in V \setminus \{s,t\}$:

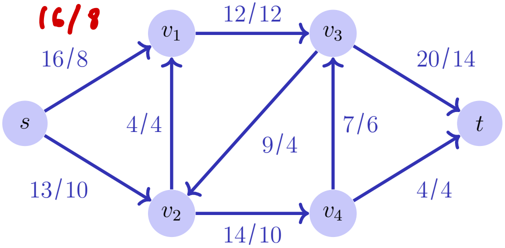
$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v).$$

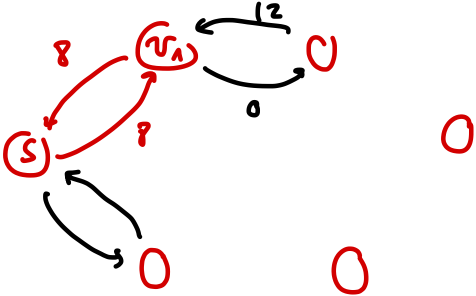


Value of the flow:
$|f| = \sum_{v \in V} f(s,v)$.
Here $|f| = 18$.

Residual Network

Residual network G_f provided by the edges with positive residual capacity:

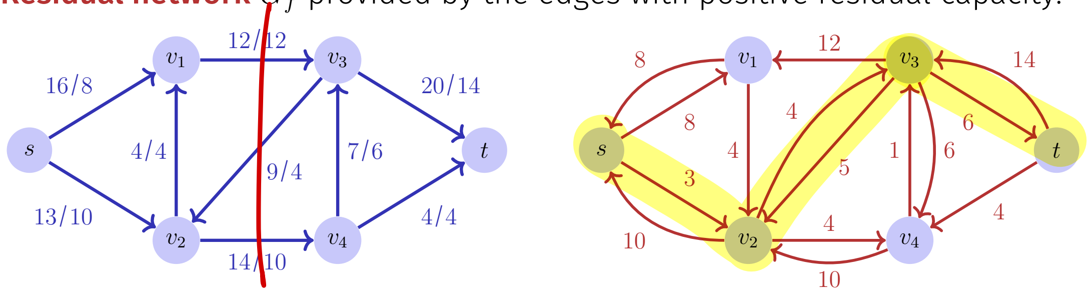




Residual networks provide the same kind of properties as flow networks with the exception of permitting antiparallel edges

Residual Network

Residual network G_f provided by the edges with positive residual capacity:



Residual networks provide the same kind of properties as flow networks with the exception of permitting antiparallel edges

Augmenting Paths

Expansion Path p: simple path from s to t in the residual network G_f.
Residual Capacity $c_f(p)$: the least capacity along the expansion path p

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ edge in } p\}$$

Algorithm Ford-Fulkerson(G, s, t)

Input: Flow network $G = (V, E, c)$
Output: Maximal flow f.

for $(u, v) \in E$ **do**
  $f(u, v) \leftarrow 0$

while Exists path $p : s \rightsquigarrow t$ in residual network G_f **do**
  $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$
  foreach $(u, v) \in p$ **do**
    if $(u, v) \in E$ **then**
      $f(u, v) \leftarrow f(u, v) + c_f(p)$
    else
      $f(v, u) \leftarrow f(u, v) - c_f(p)$

|| ↯ missmatch w/ concept

Edmonds-Karp Algorithm

Choose in the Ford-Fulkerson-Method for finding a path in G_f the expansion path of shortest possible length (e.g. with BFS)

Theorem 1

When the Edmonds-Karp algorithm is applied to some integer valued flow network $G = (V, E)$ with source s and sink t then the number of flow increases applied by the algorithm is in $\mathcal{O}(|V| \cdot |E|)$

$\Rightarrow$ ***Overall asymptotic runtime:*** $\mathcal{O}(|V| \cdot |E|^2)$

Max-Flow Min-Cut Theorem

Theorem 2

Let f be a flow in a flow network $G = (V, E, c)$ with source s and sink t. The following statements are equivalent: ("iff all around")

1. *f is a maximal flow in G*
2. *The residual network G_f does not provide any expansion paths*
3. *It holds that $|f| = c(S, T)$ for a cut (S, T) of G.*

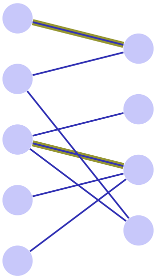
(**Hint:** This one is *really* important)

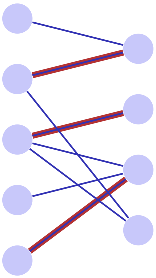
Application: maximal bipartite matching

Given: bipartite undirected graph $G = (V, E)$.

Matching M: $M \subseteq E$ such that $|\{m \in M : v \in m\}| \leq 1$ for all $v \in V$.

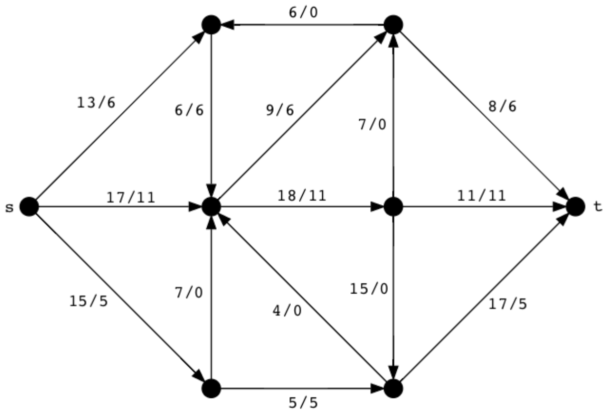
Maximal Matching M: Matching M, such that $|M| \geq |M'|$ for each matching M'.



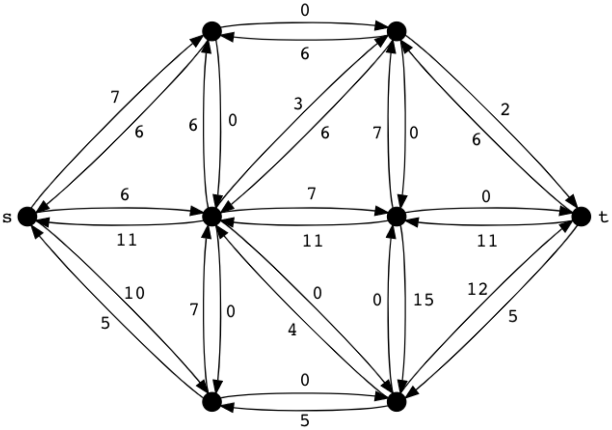


Manual Max Flow Exercise

This graph shows a flow chart that is not at maximum capacity. Run one iteration of the Ford-Fulkerson algorithm to find the max flow.



Manual Max Flow Solution



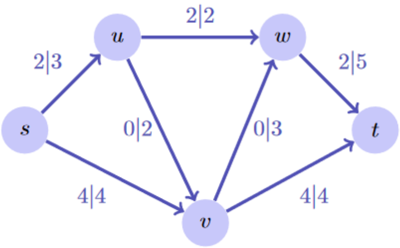
update not shown since it is not unique!

4. Old Exam Questions (Max-Flow)

Exam Question Example

Gegeben ist das folgende Flussnetzwerk G mit Quelle s und Senke t. Die einzelnen Kapazitäten c_i und Flüsse ϕ_i sind an den Kanten angegeben als $\phi_i|c_i$. Geben Sie den Wert des Flusses f an.

Provided in the following is a flow network G with source s and sink t. Capacities c_i and flows ϕ_i are provided at the edges as $\phi_i|c_i$. Provide the value of the flow f.

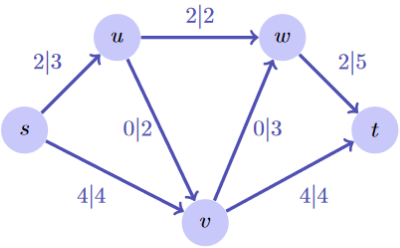


$|f| =$ ☐

Exam Question Example

Gegeben ist das folgende Flussnetzwerk G mit Quelle s und Senke t. Die einzelnen Kapazitäten c_i und Flüsse ϕ_i sind an den Kanten angegeben als $\phi_i|c_i$. Geben Sie den Wert des Flusses f an.

Provided in the following is a flow network G with source s and sink t. Capacities c_i and flows ϕ_i are provided at the edges as $\phi_i|c_i$. Provide the value of the flow f.

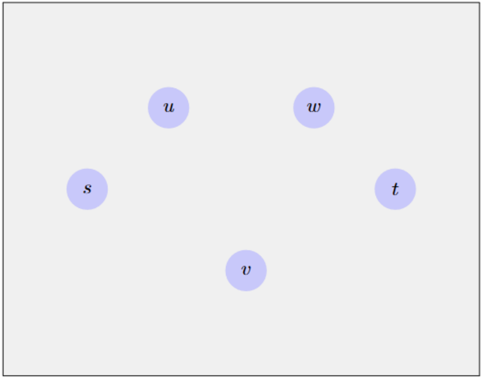


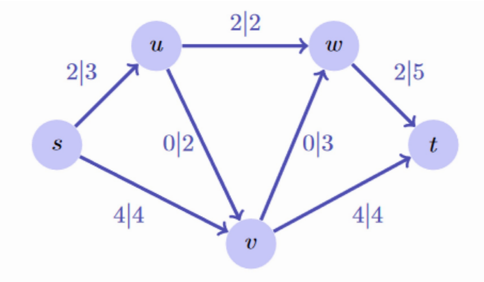
$|f| =$ 6

Exam Question Example

Zeichnen Sie nun das Restnetzwerk G_f zu obigem Fluss und markieren Sie darin einen Erweiterungspfad p. Geben Sie den Wert $c_f(p)$ der Restkapazität des Erweiterungspfades p im Restnetzwerk G_f an.

Draw the residual network G_f to the flow above and mark an augmenting path p. Provide the rest capacity $c_f(p)$ of the path p in the rest network G_f.

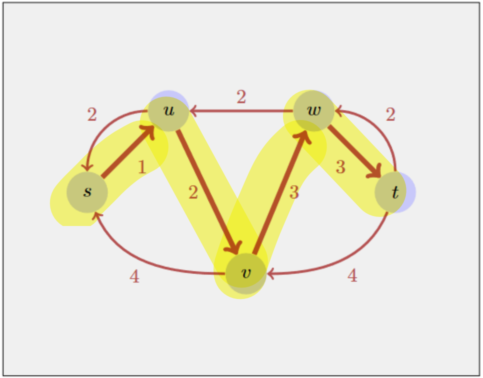




Exam Question Example

Zeichnen Sie nun das Restnetzwerk G_f zu obigem Fluss und markieren Sie darin einen Erweiterungspfad p. Geben Sie den Wert $c_f(p)$ der Restkapazität des Erweiterungspfades p im Restnetzwerk G_f an.

Draw the residual network G_f to the flow above and mark an augmenting path p. Provide the rest capacity $c_f(p)$ of the path p in the rest network G_f.



$|c_f(p)| = $ 1

Exam Question Example

Woran erkennen Sie, ob Sie den maximalen Fluss gefunden haben?

How do you see if you have found the maximum flow?

Exam Question Example

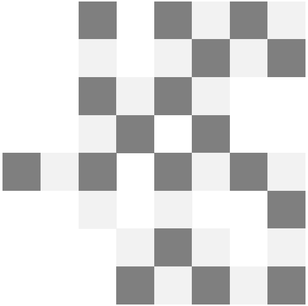
Woran erkennen Sie, ob Sie den maximalen Fluss gefunden haben?

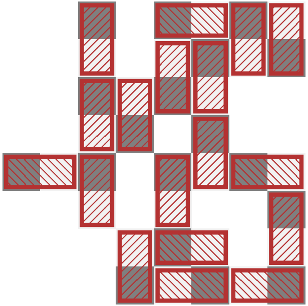
How do you see if you have found the maximum flow?

Found the maximum flow if:
The residual network does not have any more augmenting path.
Alternative: Identify a cut with $|c(S,T)| = |f|$.

Max Flow Question

which exam? □





Let an $n \times n$ chessboard be given without some squares. Describe an efficient algorithm to find out if the board can be completely covered with dominoes. Then model the problem as a flow problem.

5. Dynamic Programming

Dynamic Programming: Idea

1. Divide a complex problem into a reasonable number of sub-problems;
   Partial solutions are combined to more complex ones
   = Top-down recursion ("assume the subproblems")

Dynamic Programming: Idea

1. Divide a complex problem into a reasonable number of sub-problems;
   Partial solutions are combined to more complex ones
   = Top-down recursion ("assume the subproblems")
2. Identical problems will be computed only once
   = Memoization

Dynamic Programming: Idea

1. Divide a complex problem into a reasonable number of sub-problems; Partial solutions are combined to more complex ones
   = Top-down recursion ("assume the subproblems")
2. Identical problems will be computed only once
   = Memoization

   - The idea is to simply **store the results of subproblems** so that we do not have to re-compute them when needed later.

Dynamic Programming: Idea

1. Divide a complex problem into a reasonable number of sub-problems; Partial solutions are combined to more complex ones
   = Top-down recursion ("assume the subproblems")
2. Identical problems will be computed only once
   = Memoization
   - The idea is to simply **store the results of subproblems** so that we do not have to re-compute them when needed later.
3. Eliminate recursion
   = Bottom-up algorithms ("combine the subproblems")

Dynamic Programming: Idea

1. Divide a complex problem into a reasonable number of sub-problems; Partial solutions are combined to more complex ones
   = Top-down recursion ("assume the subproblems")
2. Identical problems will be computed only once
   = Memoization
   - The idea is to simply **store the results of subproblems** so that we do not have to re-compute them when needed later.
3. Eliminate recursion
   = Bottom-up algorithms ("combine the subproblems")

- Optionally, not always possible: Save space by storing as little as possible in the DP table

Dynamic Programming: Idea

Question: Which of the following Fibonacci implementations would perform better?

Dynamic Programming: Idea

Question: Which of the following Fibonacci implementations would perform better?

```cpp
int fib(int n) {
  if (n <= 1) {
    return n;
  }

  return fib(n - 1) +
         fib(n - 2);
}
```

```cpp
int fib2(int n) {
  std::vector<int> f(n+1);
  f[0] = 0;
  f[1] = 1;

  for(int i=2;i<=n;++i){
    f[i] = f[i-1]+f[i-2];
  }

  return f[n];
}
```

```cpp
int fib3(int n) {
  if (n <= 1) {
    return n;
  }

  int a = 0;
  int b = 1;
  for(int i=2;i<=n;++i){
    int a_old = a;
    a = b;
    b += a_old;
  }

  return b;
}
```

Dynamic Programming = Divide-And-Conquer ?

Dynamic Programming = Divide-And-Conquer ?

- In both cases the original problem can be solved (more easily) by utilizing the solutions of sub-problems. The problem provides **optimal substructure**.

Dynamic Programming = Divide-And-Conquer ?

- In both cases the original problem can be solved (more easily) by utilizing the solutions of sub-problems. The problem provides **optimal substructure**.
- Divide-And-Conquer algorithms (such as Mergesort): *sub-problems are independent*; their solutions are required only once in the algorithm.

Dynamic Programming = Divide-And-Conquer ?

- In both cases the original problem can be solved (more easily) by utilizing the solutions of sub-problems. The problem provides **optimal substructure**.
- Divide-And-Conquer algorithms (such as Mergesort): *sub-problems are independent*; their solutions are required only once in the algorithm.
- Dynamic Programming: *sub-problems are dependent*. The problem is said to have **overlapping sub-problems** that are required multiple-times in the algorithm.

Dynamic Programming = Divide-And-Conquer ?

- In both cases the original problem can be solved (more easily) by utilizing the solutions of sub-problems. The problem provides **optimal substructure**.
- Divide-And-Conquer algorithms (such as Mergesort): *sub-problems are independent*; their solutions are required only once in the algorithm.
- Dynamic Programming: *sub-problems are dependent*. The problem is said to have **overlapping sub-problems** that are required multiple-times in the algorithm.
- In order to avoid redundant computations, results are tabulated. For **sub-problems there must not be any circular dependencies**.

Memoization vs. Dynamic Programming

Memoization vs. Dynamic Programming

- **Memoization:**

Memoization vs. Dynamic Programming

- **Memoization:**
  - Top-down approach
  - Recursion with caching of results
  - Lazily computes values on-demand
  - Can be more efficient if only a few values are needed
- **Dynamic Programming:**

Memoization vs. Dynamic Programming

- **Memoization:**
  - Top-down approach
  - Recursion with caching of results
  - Lazily computes values on-demand
  - Can be more efficient if only a few values are needed
- **Dynamic Programming:**
  - Iterative bottom-up approach
  - Builds solutions from smaller subproblems
  - Computes all values in a predefined order
  - Can be more efficient if all values are needed

Problem Without Optimal Substructure

Question: Problem Without Optimal Substructure?

Problem Without Optimal Substructure

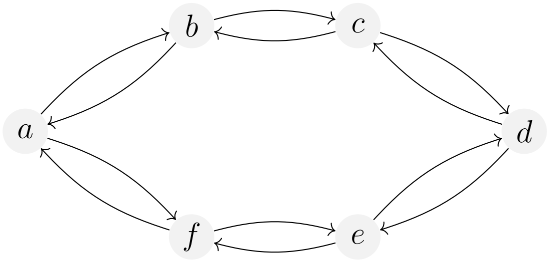
Question: Problem Without Optimal Substructure?

Example: Longest (simple) path

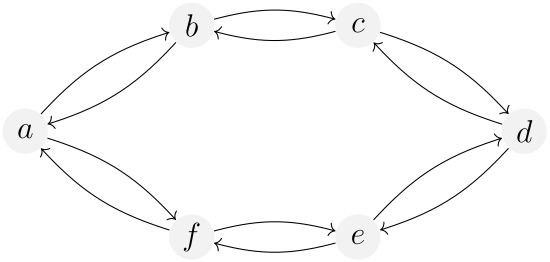
Problem Without Optimal Substructure

Question: Problem Without Optimal Substructure?

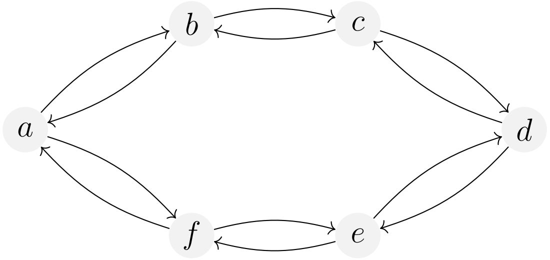
Example: Longest (simple) path



Problem Without Optimal Substructure: Longest Path

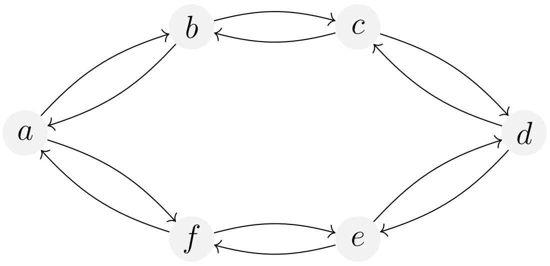


Problem Without Optimal Substructure: Longest Path



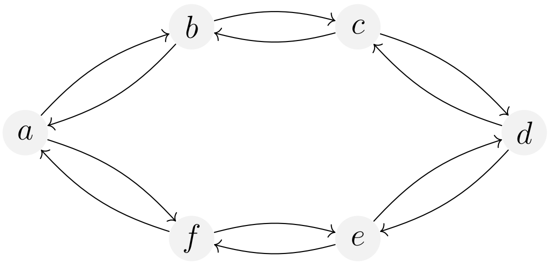
- Longest path from, e.g. a to e is a, b, c, d, e, i.e. via c

Problem Without Optimal Substructure: Longest Path



- Longest path from, e.g. a to e is a, b, c, d, e, i.e. via c
- But the longest path from a to c is *not* a, b, c (and analogously for c to e)

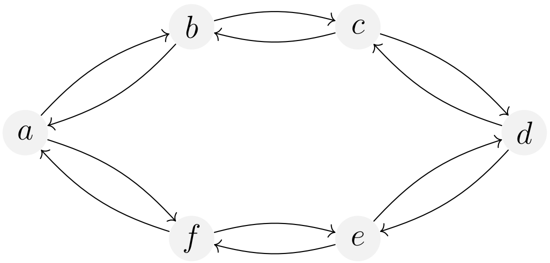
Problem Without Optimal Substructure: Longest Path



- Longest path from, e.g. a to e is a, b, c, d, e, i.e. via c
- But the longest path from a to c is *not* a, b, c (and analogously for c to e)

$\Rightarrow$ Combining optimal subsolutions does not yield an optimal overall solution

Problem Without Optimal Substructure: Longest Path



- Longest path from, e.g. a to e is a, b, c, d, e, i.e. via c
- But the longest path from a to c is *not* a, b, c (and analogously for c to e)

$\Rightarrow$ Combining optimal subsolutions does not yield an optimal overall solution

$\Rightarrow$ This problem does not have optimal substructure

Memoization vs. Dynamic Programming

Question

In which of the following cases might memoization be significantly more efficient than dynamic programming?

1. When all values are required for the final result
2. When only a few values are required for the final result
3. When the problem has overlapping subproblems
4. When the problem can be solved iteratively

Memoization vs. Dynamic Programming

Answer

Memoization might be significantly more efficient than dynamic programming when only a few values are required for the final result (option 2).

Dynamic Programming

A complete description of a dynamic program **always** consists of:

Dynamic Programming

A complete description of a dynamic program **always** consists of:

- **Definition of the subproblems / the DP table**:

Dynamic Programming

A complete description of a dynamic program **always** consists of:

- **Definition of the subproblems / the DP table**:
  What are the dimensions of the table? What is the meaning of each entry?

Dynamic Programming

A complete description of a dynamic program **always** consists of:

- **Definition of the subproblems / the DP table**:
  What are the dimensions of the table? What is the meaning of each entry?
- **Recursion: Computation of an entry**:

Dynamic Programming

A complete description of a dynamic program **always** consists of:

- **Definition of the subproblems / the DP table**:
  What are the dimensions of the table? What is the meaning of each entry?
- **Recursion: Computation of an entry**:
  How can an entry be computed from the values of other entries? Which entries do not depend on others?

Dynamic Programming

A complete description of a dynamic program **always** consists of:

- **Definition of the subproblems / the DP table**:
  What are the dimensions of the table? What is the meaning of each entry?
- **Recursion: Computation of an entry**:
  How can an entry be computed from the values of other entries? Which entries do not depend on others?
- **Computation order (topological order)**:

Dynamic Programming

A complete description of a dynamic program **always** consists of:

- **Definition of the subproblems / the DP table**:
  What are the dimensions of the table? What is the meaning of each entry?
- **Recursion: Computation of an entry**:
  How can an entry be computed from the values of other entries? Which entries do not depend on others?
- **Computation order (topological order)**:
  In which order can entries be computed so that values needed for each entry have been determined in previous steps?

Dynamic Programming

A complete description of a dynamic program **always** consists of:

- **Definition of the subproblems / the DP table**:
  What are the dimensions of the table? What is the meaning of each entry?
- **Recursion: Computation of an entry**:
  How can an entry be computed from the values of other entries? Which entries do not depend on others?
- **Computation order (topological order)**:
  In which order can entries be computed so that values needed for each entry have been determined in previous steps?
- **Solution and Running Time**:

Dynamic Programming

A complete description of a dynamic program **always** consists of:

- **Definition of the subproblems / the DP table**:
  What are the dimensions of the table? What is the meaning of each entry?
- **Recursion: Computation of an entry**:
  How can an entry be computed from the values of other entries? Which entries do not depend on others?
- **Computation order (topological order)**:
  In which order can entries be computed so that values needed for each entry have been determined in previous steps?
- **Solution and Running Time**:
  How can the final solution be extracted once the table has been filled? Running time of the DP algorithm.

Review

Choose which characteristics a problem needs to have for a dynamic programming approach to be appropriate:

- Optimal substructure
- Real-time problem-solving
- Independent sub-problems
- Memory-efficient solution
- Recursive structure
- Overlapping sub-problems
- Circular dependencies
- Tabulation or memoization potential
- Small state space

Answers

Choose which characteristics a problem needs to have for a dynamic programming approach to be appropriate:

- **Optimal substructure**
- Real-time problem-solving
- Independent sub-problems
- (Memory-efficient solution *)
- **Recursive structure**
- **Overlapping sub-problems**
- Circular dependencies
- **Tabulation or memoization potential**
- Small state space

Example: Coin Change Problem

Definition

Given a set of coin denominations and a target amount, find the minimum number of coins needed to make the target amount. Note that the same coin denomination can be used more than once.

Example: Coin Change Problem

Definition

Given a set of coin denominations and a target amount, find the minimum number of coins needed to make the target amount. Note that the same coin denomination can be used more than once.

Example

Given coins = [1, 2, 4] and target amount = 8, the solution is 2 (4 + 4).

Example: Coin Change Problem

Definition

Given a set of coin denominations and a target amount, find the minimum number of coins needed to make the target amount. Note that the same coin denomination can be used more than once.

Example

Given coins = [1, 2, 4] and target amount = 8, the solution is 2 (4 + 4).

Remark

When the problem does not have a solution, the algorithm returns -1.

Coin Change Problem

Task

Design a recursive algorithm to solve the task.

Coin Change: Recursive Solution

```cpp
int coinChange(const std::vector<int>& coins, int amount) {
    if (amount == 0) {
        return 0;
    }
    int minCoins = INT_MAX;
    for (int coin : coins) {
        if (amount - coin >= 0) {
            int temp = coinChange(coins, amount - coin);
            if (temp != -1) {
                minCoins = std::min(minCoins, temp + 1);
            }
        }
    }
    return minCoins == INT_MAX ? -1 : minCoins;
}
```

Coin Change: Recursive Solution

```cpp
int coinChange(const std::vector<int>& coins, int amount) {
    if (amount == 0) {
        return 0;
    }
    int minCoins = INT_MAX;
    for (int coin : coins) {
        if (amount - coin >= 0) {
            int temp = coinChange(coins, amount - coin);
            if (temp != -1) {
                minCoins = std::min(minCoins, temp + 1);
            }
        }
    }
    return minCoins == INT_MAX ? -1 : minCoins;
}
```

Coin Change Problem

Task

Design a DP algorithm to solve the task.

Coin Change: Dynamic Programming

We can use dynamic programming to solve this problem by building a one-dimensional array where `dp[i]` represents the minimum number of coins required to make the amount i:

- Set each element in `dp` to a value larger than the maximum possible number of coins.
- Set `dp[0] = 0`.
- For each coin `c`, iterate through the array and update `dp[i]` if `dp[i-c]+1` has a lower value.

Indep of all other entries!

Coin Change: DP Solution

```
int coinChange(const std::vector<int>& coins, int amount) {
    std::vector<int> dp(amount + 1, amount + 1);
    dp[0] = 0;
    for (int coin : coins) {
        for (int i = coin; i <= amount; ++i) {
            dp[i] = std::min(dp[i], dp[i - coin] + 1);
        }
    }
    return dp[amount] <= amount ? dp[amount] : -1;
}
```

Coin Change: DP Visualisation

$i \in 0 \rightarrow 8$

```
dp[i] = std::min(dp[i], dp[i - coin] + 1);
```

Coins: [1, 2, 4]   Target: 8

i	0	1	2	3	4	5	6	7	8
dp[i]	0	~~∞~~ 1	∞	∞	∞	∞	∞	∞	∞

Initial state of the **dp** array. Note that we use ∞ instead of **amount+1**.

$dp[1] = \min(\infty, \underbrace{dp[1-1]+1}_{1})$

Coin Change: DP Visualisation

```
dp[i] = std::min(dp[i], dp[i - coin] + 1);
```

Coins: [1, 2, 4]   Target: 8

i	0	1	2	3	4	5	6	7	8
dp[i]	0	1	2	3	4	5	6	7	8

1

After processing the first coin.

dp[2] ← min(dp[2] , dp[2 - 2] + 1) = 1

2   0 + 1

Coin Change: DP Visualisation

```
dp[i] = std::min(dp[i], dp[i - coin] + 1);
```

Coins: [1, 2, 4]   Target: 8

i	0	1	2	3	4	5	6	7	8
dp[i]	0	1	1	2	2	3	3	4	4

After processing the second coin.

Coin Change: DP Visualisation

A complete description of a dynamic program **always** consists of:

- **Definition of the subproblems / the DP table**: What are the dimensions of the table? What is the meaning of each entry?
- **Recursion: Computation of an entry**: How can an entry be computed from the values of other entries? Which entries do not depend on others?
- **Computation order (topological order)**: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
- **Solution and Running Time**: How can the final solution be extracted once the table has been filled? Running time of the DP algorithm.

```
dp[i] = std::min(dp[i], dp
```

Coins: [1, 2, 4]   Target: 8

i	0	1	2	3	4	5	6	7	8
dp[i]	0	1	1	2	1	2	2	3	2

dp[target]

After processing the third and last coin. Answer: `dp[8] = 2`.

Coin Change: Time Complexity

Task

Compare the time complexity of the DP algorithm with that of the naive recursive algorithm

Coin Change: Time Complexity

Task

Compare the time complexity of the DP algorithm with that of the naive recursive algorithm

Naive Algorithm

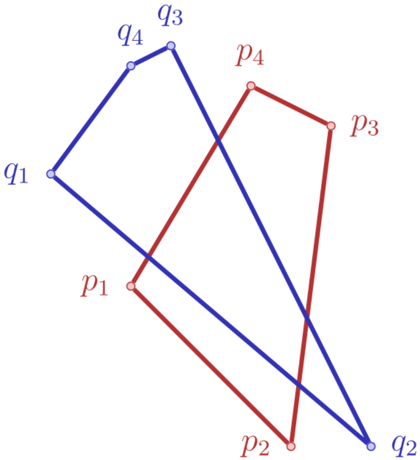
The naive algorithm has an exponential time complexity of $\mathcal{O}(c^n)$, where c is the number of coin denominations and n is the target amount.

Dynamic Programming Algorithm

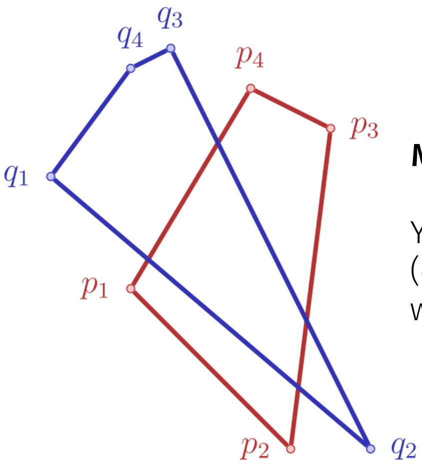
The dynamic programming algorithm has a polynomial time complexity of $\mathcal{O}(c \cdot n)$, where c is the number of coin denominations and n is the target amount.

6. Overlap of Convex Polygons

Overlap of Convex Polygons – Issues



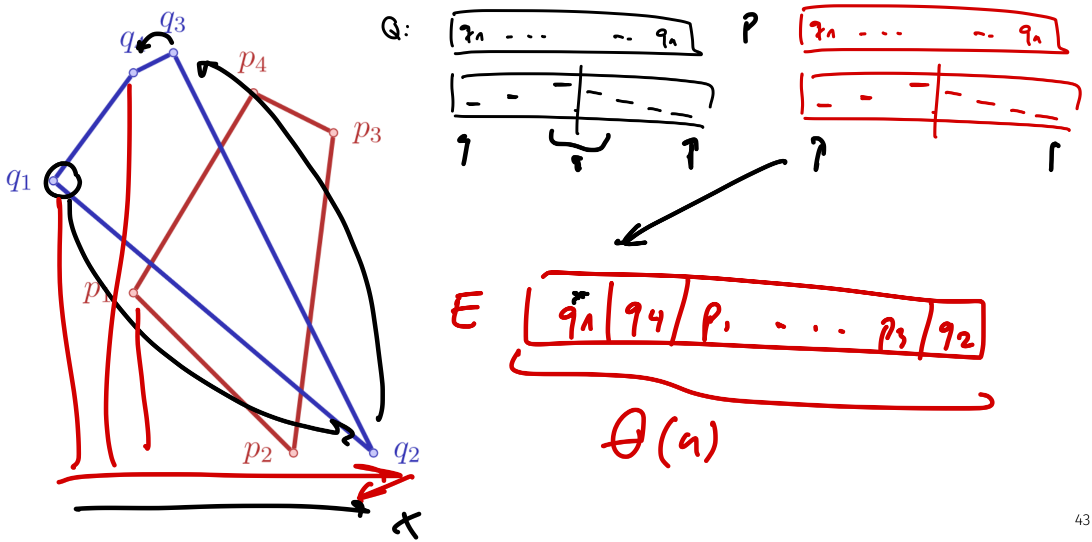
Overlap of Convex Polygons – Issues



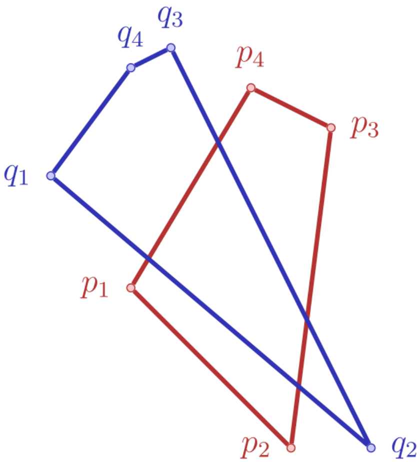
Main issue with most solutions

You sorted the given polygon points ($\mathcal{O}(n \log n)$) instead of using the fact that they were given in partly sorted order!

Overlap of Convex Polygons – Solution Sketch

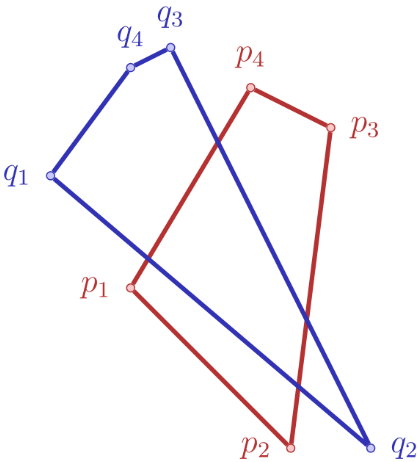


Overlap of Convex Polygons – Solution Sketch



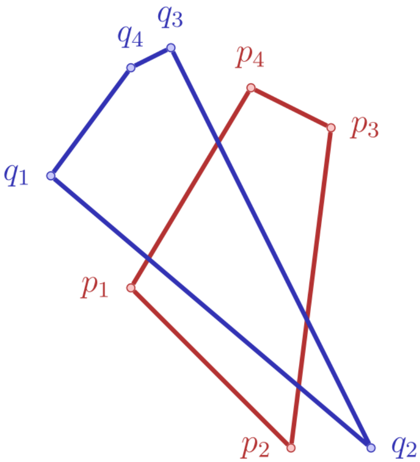
- The **Event Points** are the $2n$ points of the convex polygons, sorted by their x-coord

Overlap of Convex Polygons – Solution Sketch



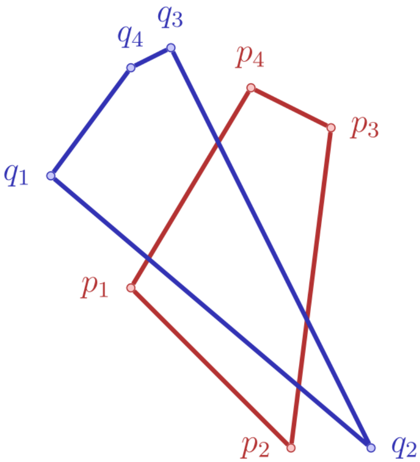
- The **Event Points** are the $2n$ points of the convex polygons, sorted by their x-coord
- They can be stored in a sorted array by merging the sequences $p_1, \ldots, p_n$ and $q_1, \ldots, q_n$ (given in counterclockwise sorting starting with the left-most point)

Overlap of Convex Polygons – Solution Sketch



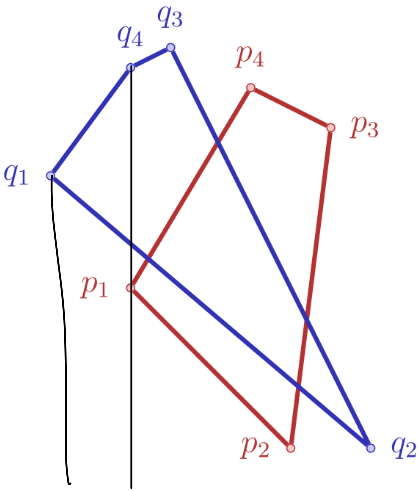
- The **Event Points** are the $2n$ points of the convex polygons, sorted by their x-coord
- They can be stored in a sorted array by merging the sequences $p_1, \ldots, p_n$ and $q_1, \ldots, q_n$ (given in counterclockwise sorting starting with the left-most point)
- Split each sequence into increasing and decreasing subsequences, then merge the increasing subsequences and the reversed decreasing subsequences

Overlap of Convex Polygons – Solution Sketch



- The **Event Points** are the $2n$ points of the convex polygons, sorted by their x-coord
- They can be stored in a sorted array by merging the sequences $p_1, \ldots, p_n$ and $q_1, \ldots, q_n$ (given in counterclockwise sorting starting with the left-most point)
- Split each sequence into increasing and decreasing subsequences, then merge the increasing subsequences and the reversed decreasing subsequences
- Store the polygon info and incident line segments for each point

Overlap of Convex Polygons – Solution Sketch



- The **Event Points** are the $2n$ points of the convex polygons, sorted by their x-coord
- They can be stored in a sorted array by merging the sequences $p_1, \ldots, p_n$ and $q_1, \ldots, q_n$ (given in counterclockwise sorting starting with the left-most point)
- Split each sequence into increasing and decreasing subsequences, then merge the increasing subsequences and the reversed decreasing subsequences
- Store the polygon info and incident line segments for each point
- **This step can be completed in $\Theta(n)$ time!**

7. In-Class Code-Example

Code-Examples: Memoization and DP

Memoization and DP: Maximum Sum of an Increasing Subsequence
$\longrightarrow$ **code** expert

8. Outro

General Questions?

See you next time

Have a nice week!