Announcements

- Reminder: HW1 due next Wednesday 6 pm
  - You can use at most 2 late days

CS6161: Design and Analysis of Algorithms (Fall 2020)

Dynamic Programming

Instructor: Haifeng Xu

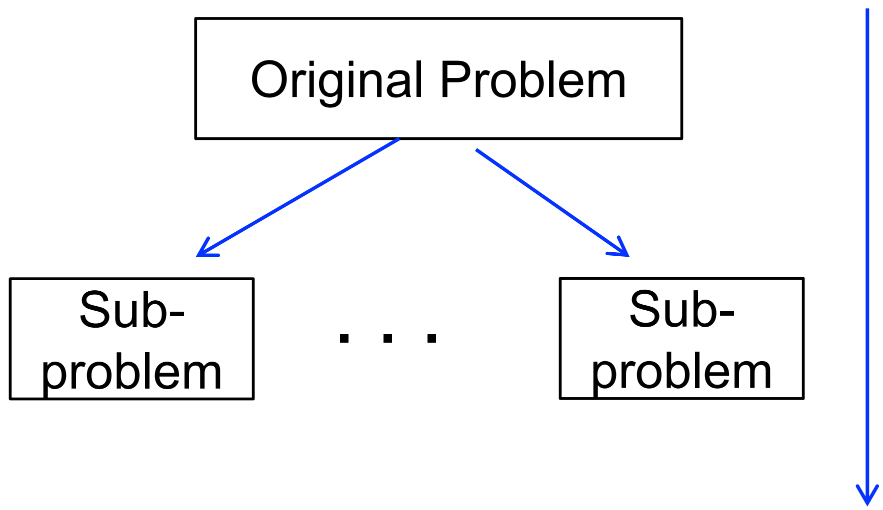
Outline

- Dynamic Programming
- Two Other Examples

Dynamic Programming (DP)

- Like the "reversed" Divide and Conquer

Recall D-and-C:

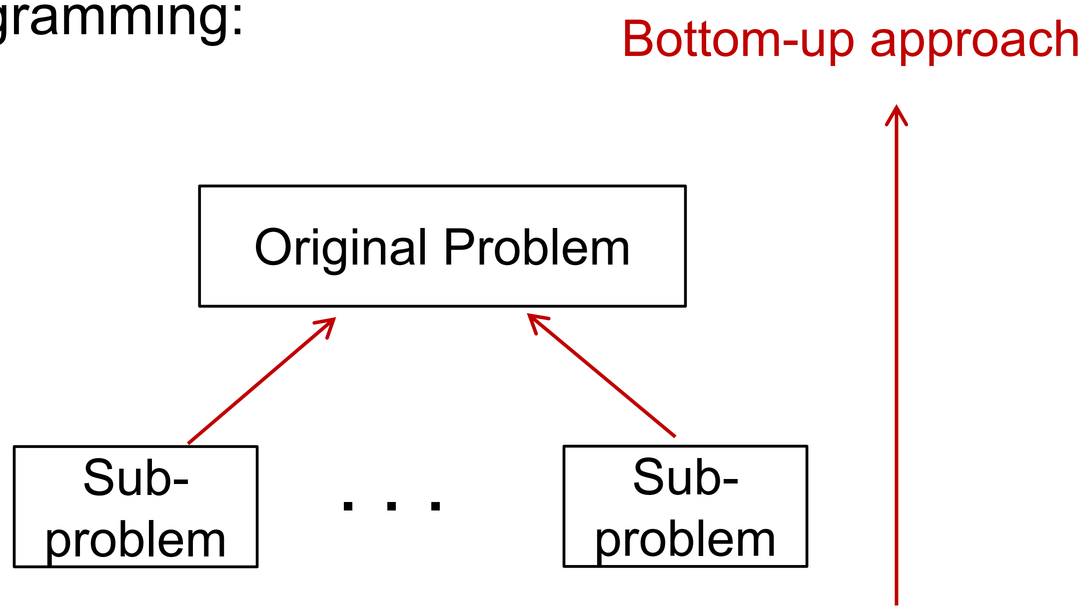


Top-down approach

Dynamic Programming (DP)

- Like the "reversed" Divide and Conquer

Dynamic Programming:



<u>Commonality</u>: both need to find the right subproblems to solve

But…why there is a difference between top-down and bottom-up approaches?

An Example: Rod Cutting

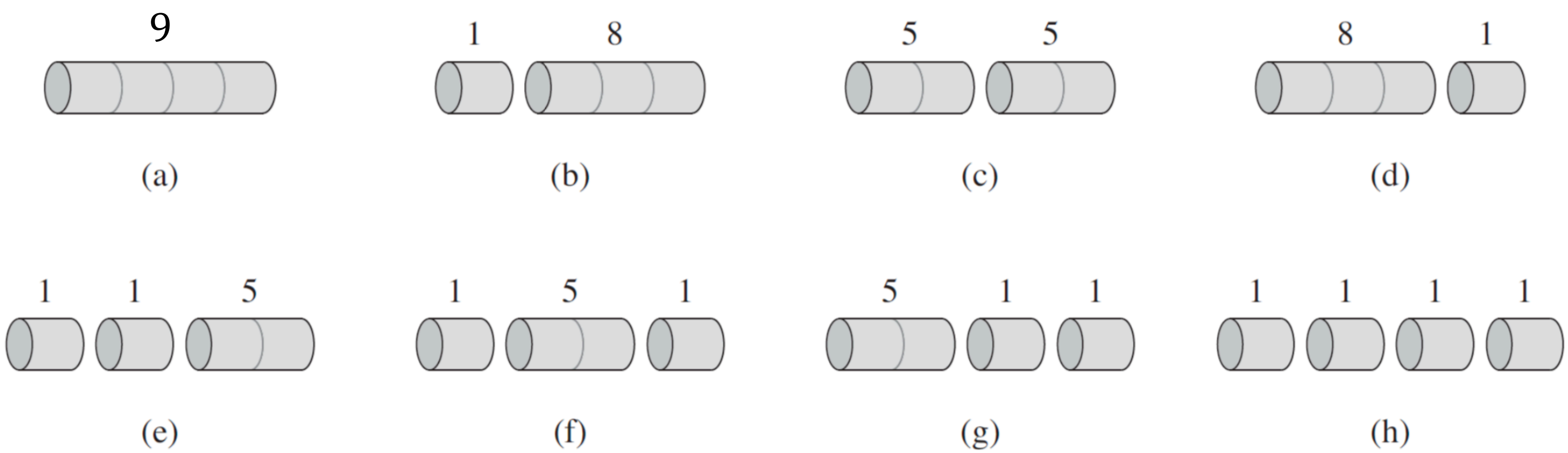
- Want to sell a steel rod with total length n
- The prices for different lengths are different

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Algorithmic Question: how to cut the rod into pieces so that it maximizes your revenue?

An Example: Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



Naïve Algorithm

- Try all possible partitions of number n

Roughly $\frac{e^{\pi\sqrt{\frac{2n}{3}}}}{4\sqrt{3}\,n}$ many ways to partition n . . . Still too many

What About Divide and Conquer?

```
Rod-Cut(p, n)
    If n == 0
        return 0
```

What About Divide and Conquer?

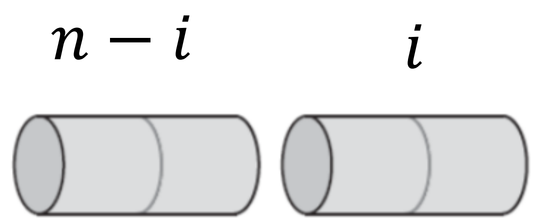
Rod-Cut(p, n)

If $n == 0$

  return 0

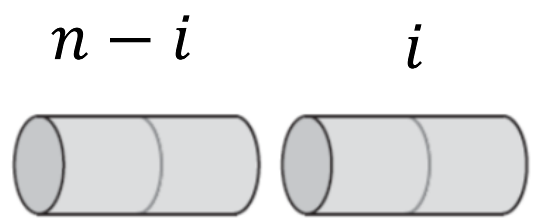
$q = -\infty$

for $i = 1, \ldots, n$



What About Divide and Conquer?

```
Rod-Cut(p, n)
    If n == 0
        return 0
    q = −∞
    for i = 1, … , n
        q = max{q,  p[i] + Rod-Cut(p, n − i)}
    return q
```



<u>Key Property</u>: If your cut is optimal overall, then after a cut of length i, the remaining $(n - i)$ must be optimally cut as well.

Running Time Analysis

Rod-Cut(p, n)

If $n == 0$

    return 0

$q = -\infty$

for $i = 1, \ldots, n$

    $q = \max\{q,\ p[i] + \text{Rod-Cut}(p, n - i)\}$

return q

Recursion: $T(n) = 1 + \sum_{i=1}^{n-1} T(i) \quad \Rightarrow T(n) = 2^{n-1}$

What is the Issue with This Algorithm?

- Solving the same sub-problems for too many times
  - Solved $\text{Rod}-\text{Cut}(p, n-2)$ once when considering n, and once again when considering $n-1$

Rod-Cut(p, n)

1 **If** $n == 0$

2 **return** 0

3 $q = -\infty$

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

5 $q = \max\{q,\ p[i] + \text{Rod-Cut}(p, n-i)\}$

6 **return** q

How to fix?

- Once $\text{Rod}-\text{Cut}(p, k)$ is solved, remember its answer!
- In principle, you can also do a *top-down* approach
  - Use an array to remember your solved ks
  - In step 5, instead of $\text{Rod}-\text{Cut}(p, n-i)$, use your recorded sol whenever possible

The Dynamic Programming Approach

- DP uses a bottom-up process

Rod-Cut-DP(p, n)

The Dynamic Programming Approach

- DP uses a bottom-up process

```
Rod-Cut-DP(p, n)
    Let r[0:n] be a new array
    r[0] = 0
```

The Dynamic Programming Approach

- DP uses a bottom-up process

Rod-Cut-DP(p, n)

```
Let r[0:n] be a new array
r[0] = 0
for j = 1, ..., n
        r[j] = q
```

- Bottom-up: from small instances up to large instances
  - Dynamically built up

The Dynamic Programming Approach

- DP uses a bottom-up process

Rod-Cut-DP(p, n)

Let $r[0{:}n]$ be a new array
$r[0] = 0$
for $j = 1, \dots, n$
    $q = -\infty$
    for $i = 1, \dots, j$
        $q = \max\{q,\ p[i] + r(j - i)\}$
    $r[j] = q$

- Bottom-up: from small instances up to large instances
  - Dynamically built up
- When solving case j, all its subproblems have already been solved

The Dynamic Programming Approach

- DP uses a bottom-up process

Rod-Cut-DP(p, n)

Let $r[0:n]$ be a new array
$r[0] = 0$
for $j = 1, \dots, n$
  $q = -\infty$
  for $i = 1, \dots, j$
    $q = \max\{q,\ p[i] + r(j-i)\}$
  $r[j] = q$
return $r[n]$

- Bottom-up: from small instances up to large instances
  - Dynamically built up
- When solving case j, all its subproblems have already been solved

How to Figure Out the Cuts?

- We only computed the optimal revenue $r[n]$
- Simple modifications to record optimal first cut

Rod-Cut-DP-Expanded(p, n)

Let $r[0{:}n]$ and $c[0{:}n]$ be a new array
$r[0] = 0$
for $j = 1, \ldots, n$
  $q = -\infty$
  for $i = 1, \ldots, j$
    if $p[i] + r(j - i) > q$
      $q = p[i] + r(j - i)$
      $c[j] = i$
  $r[j] = q$
return $r[n], c[1{:}n]$

- To output all cuts, print
  - $i_1 = c[n]$
  - $i_2 = c[n - i_1]$
  - $i_3 = c[n - i_1 - i_2]$
  - …

Algorithm Analysis

- Correctness follows easily
- Running time: $O(n^2)$
  - Due to 2 for-loops

Rod-Cut-DP-Expanded(p, n)

Let $r[0{:}n]$ and $c[0{:}n]$ be a new array
$r[0] = 0$
for $j = 1, \ldots, n$
    $q = -\infty$
    for $i = 1, \ldots, j$
        if $p[i] + r(j - i) > q$
            $q = p[i] + r(j - i)$
            $c[j] = i$
    $r[j] = q$
return $r[n], c[1{:}n]$

Some Notes

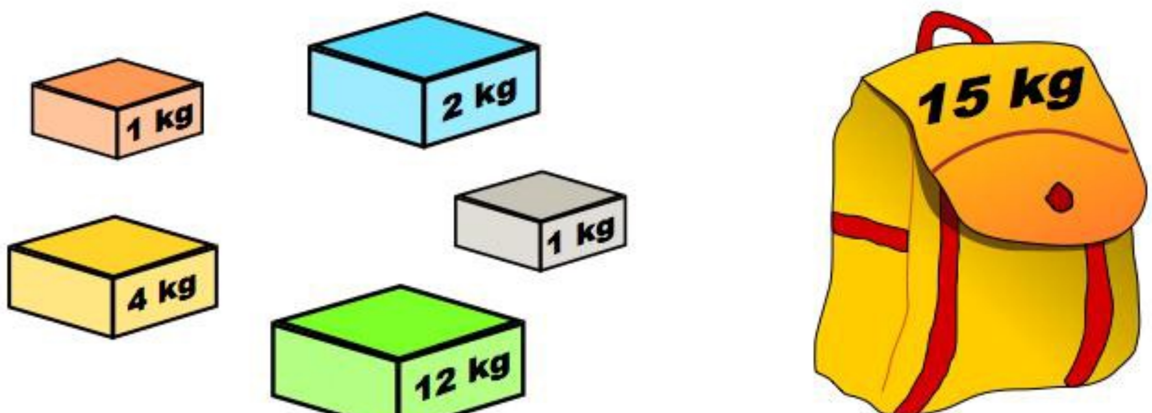
- Useful when you need to solve sub-problems for many times
- Not all problems are suitable
  - Sorting: each sub-problems is solved exactly once, so no repetitive work
- Usually the problem has an "order" (e.g., length $n, n-1, \cdots$) and has "optimality of subproblems" structure

Outline

- Dynamic Programming
- Two Other Examples

Example 1: Unbounded Knapsack Problem

- You have a knapsack with weight capacity W
- There are n different items – item i has value p_i and weight w_i
- Each item has infinitely many copies



Example 1: Unbounded Knapsack Problem

- You have a knapsack with weight capacity W
- There are n different items – item i has value p_i and weight w_i
- Each item has infinitely many copies

The Algorithmic Problem

- Input: $W, \{\mathrm{p_i}, w_i\}_{i=1,\cdots,n}$
- Output: integer array $x[1:n]$ that maximizes $\sum_{i=1}^{n} x_i p_i$, *subject to* $\sum_{i=1}^{n} x_i w_i \leq W$

Note: this is a generalization of the rod cutting problem!

- Rod cutting: $W = n$, $w_i = i$

Example 1: Unbounded Knapsack Problem

- You have a knapsack with weight capacity W
- There are n different items – item i has value p_i and weight w_i
- Each item has infinitely many copies

The Algorithmic Problem

- Input: $W, \{\mathrm{p_i}, w_i\}_{i=1,\cdots,n}$
- Output: integer array $x[1:n]$ that maximizes $\sum_{i=1}^{n} x_i p_i$, *subject to* $\sum_{i=1}^{n} x_i w_i \leq W$

Remarks:

- Many applications in combinatorial optimization
- A very important NP-hard problem in complexity theory
- Many other variants: bounded knapsack, 0-1 knapsack…

DP for Unbounded Knapsack Problems

- Assume W and w_is are all integers

DP-Knapsack$(W, \{\mathrm{p_i}, w_i\}_{i=1,\cdots,n})$

Let $r[0{:}\,W]$ be a new array

$r[0] = 0$

DP for Unbounded Knapsack Problems

- Assume W and w_is are all integers

DP-Knapsack$(W, \{\mathrm{p_i}, w_i\}_{i=1,\cdots,n})$

Let $r[0{:}W]$ be a new array
$r[0] = 0$
for $j = 1, \dots, W$
    $q = -\infty$
    for $i = 1, \dots, n$
        $q = \max\{q,\ p_i + r(j - w_i)\}$
    $r[j] = q$
return $r[n]$

Running Time Analysis

$O(nW)$!

DP-Knapsack$(W, \{\mathrm{p_i}, w_i\}_{i=1,\cdots,n})$

Let $r[0{:}W]$ be a new array
$r[0] = 0$
for $j = 1, \dots, W$
    $q = -\infty$
    for $i = 1, \dots, n$
        $q = \max\{q,\ p_i + r(j - w_i)\}$
    $r[j] = q$
return $r[n]$

Question: Polynomial time? But this is an NP-hard problem – what goes wrong? Did we prove P=NP?

Certainly not….

Pseudo Polynomial Time Algorithm

- The subtlety is all about the input size of your problem

Q: How many bits it takes to describe input $W, \{\mathrm{p_i}, w_i\}_{i=1,\cdots,n}$?

- Describe a number W takes about $\log W$ binary bits
- So the input size is of Knapsack is $\mathrm{logW} + \sum_i(\log w_i + \log p_i)$

An $O(nW)$ time algorithm is exponential in its input size!

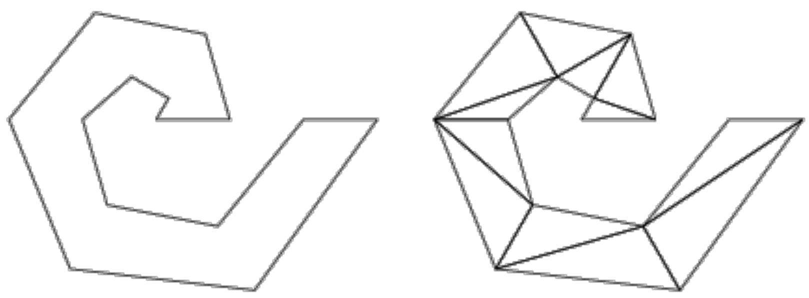
Any polynomial time algorithm must be in $\mathrm{poly}(\log W, n)$ time.

Pseudo Polynomial Time Algorithm

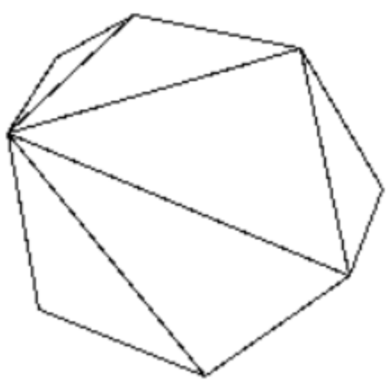
- NP-hard problems that admit pseudo-polynomial time algorithm are called weakly NP-hard
  - They cannot be solved exactly in poly time but usually admits fast approximate algorithms

Example 2: Polygon Triangulation

- Connect vertices to partition the polygon into triangles



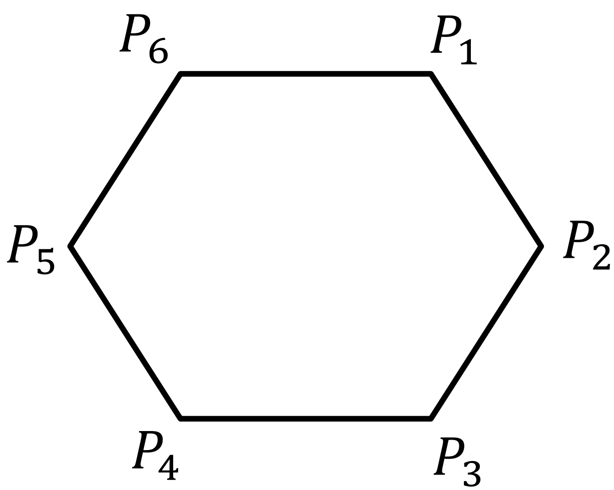
Non-convex case



Convex case

Minimum-Weight Triangulation (MWT)

- Input: A polygon, described as $P_1, \cdots, P_n$
- Output: a triangulation, described by edges, which minimum total edge length (including boundary edges)

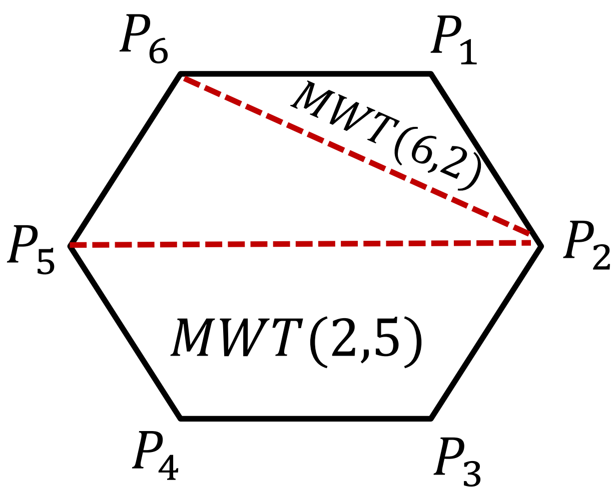


NP-hard for non-convex polygons,

But has efficient algorithms for convex polygons

DP for Convex MWT

Q1: how to break the problem into sub-problems?

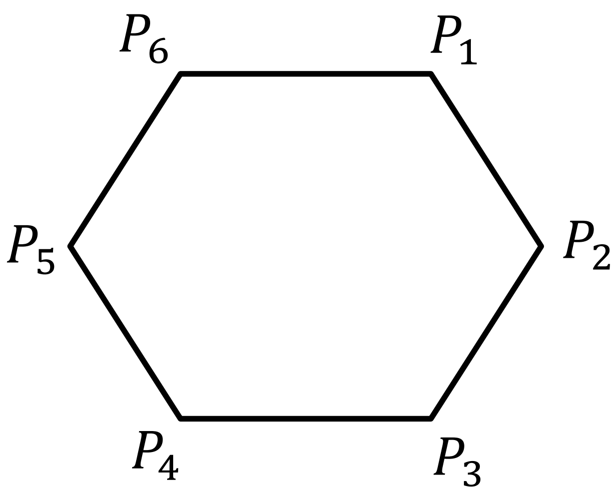


$$MWT(6,5) \\ = d(P_6, P_5) + MWT(6,2) + MWT(2,5)$$

- In any triangulation, any two adjacent vertices must be in the same triangle with some other vertex k
- Thus, subproblems $MWT(i,j)$ where j may be smaller than i

DP for Convex MWT

Q2: Where to start? Base cases?



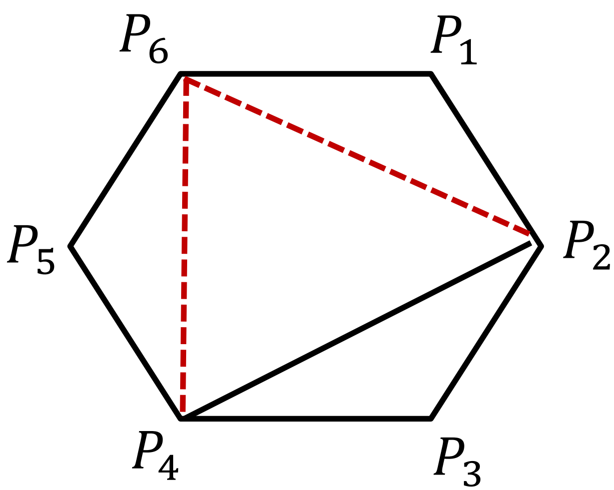
Subproblems $MWT(i, j)$

- Easy when j $= i + 1$
- $j = i + k$ can be built upon cases with $j < i + k$

- In Knapsack, weight capacity follows a natural order: $0,1,2, \cdots$
- What is a natural order here?

DP for Convex MWT

Q2: Where to start? Base cases?



Subproblems $MWT(i,j)$

- Easy when j $= i + 1$
- $j = i + k$ can be built upon cases with $j < i + k$
- So $(j - i)$ is the order we want

- $MWT(4{,}2)$ is broken into $MWT(4{,}6)$ and $MWT(6{,}2)$, with additional edge dost $d(P_2, P_4)$

DP for Convex MWT

DP-MWT$(P_1, \cdots, P_n)$

```
MWT[i, j] = new array for any i ≠ j
for i = 1, ..., n
    MWT[i, i + 1] = d(P_i, P_{i+1})
```

DP for Convex MWT

```
DP-MWT(P_1, ⋯, P_n)
    MWT[i, j] = new array for any i ≠ j
    for i = 1, …, n
        MWT[i, i + 1] = d(P_i, P_{i+1})
    for k = 2, …, n − 1
```

DP for Convex MWT

DP-MWT$(P_1, \cdots, P_n)$

$MWT[i, j]$ = new array for any $i \neq j$

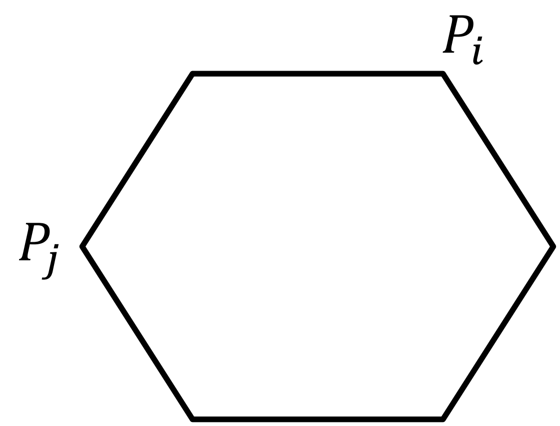
for $i = 1, \ldots, n$

    $MWT[i, i+1] = d(P_i, P_{i+1})$

for $k = 2, \ldots, n-1$

    for $i = 1, \ldots, n$

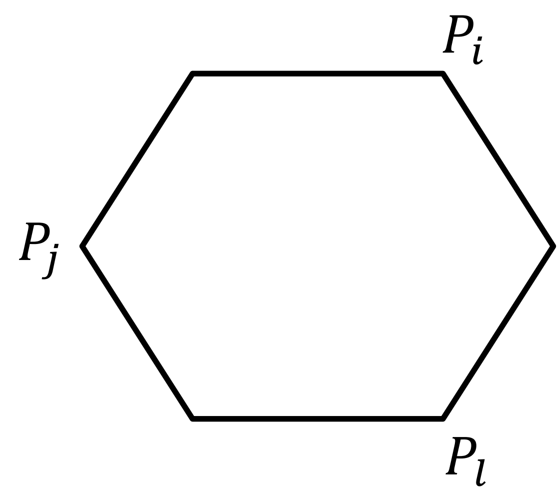
        $j = i + k$



DP for Convex MWT

DP-MWT$(P_1, \cdots, P_n)$

$MWT[i,j]$ = new array for any $i \neq j$
for $i = 1, \ldots, n$
    $MWT[i, i+1] = d(P_i, P_{i+1})$
for $k = 2, \ldots, n-1$
    for $i = 1, \ldots, n$
        $j = i + k$
        $q = +\infty$
        for $l = i+1, \ldots, j-1$ (mod n)



DP for Convex MWT

DP-MWT$(P_1, \cdots, P_n)$

$MWT[i,j]$ = new array for any $i \neq j$

for $i = 1, \dots, n$

  $MWT[i, i+1] = d(P_i, P_{i+1})$

for $k = 2, \dots, n-1$

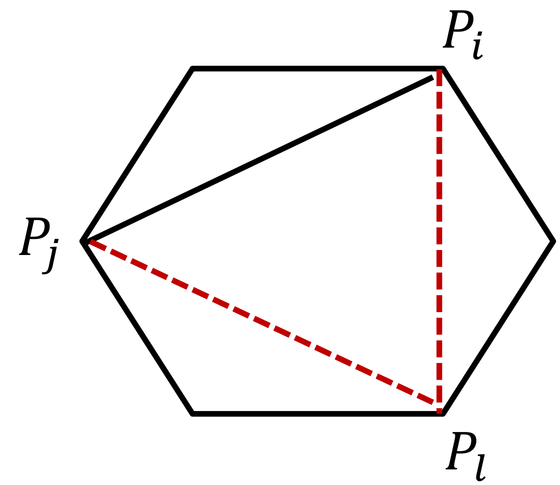
  for $i = 1, \dots, n$

    $j = i + k$

    $q = +\infty$

    for $l = i+1, \dots, j-1$ (mod n)

      $q = \min\{q,\ d(P_i, P_j) + MWT(i,l) + MWT(l,j)\}$



DP for Convex MWT

DP-MWT$(P_1, \cdots, P_n)$

$MWT[i,j]$ = new array for any $i \neq j$

for $i = 1, \ldots, n$

  $MWT[i, i+1] = d(P_i, P_{i+1})$

for $k = 2, \ldots, n-1$

  for $i = 1, \ldots, n$

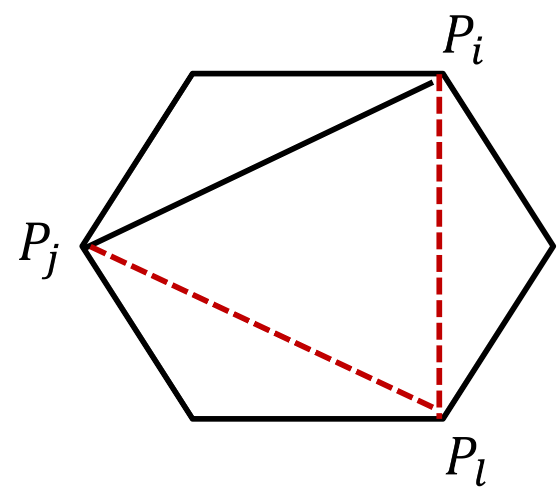
    $j = i + k$

    $q = +\infty$

    for $l = i+1, \ldots, j-1$ (mod n)

      $q = \min\{q,\ d(P_i, P_j) + MWT(i,l) + MWT(l,j)\}$

    $MWT[i,j] = q$



DP for Convex MWT

DP-MWT($P_1, \cdots, P_n$)

$MWT[i,j]$ = new array for any $i \neq j$

for $i = 1, \dots, n$

  $MWT[i, i+1] = d(P_i, P_{i+1})$

for $k = 2, \dots, n-1$

  for $i = 1, \dots, n$

    $j = i + k$

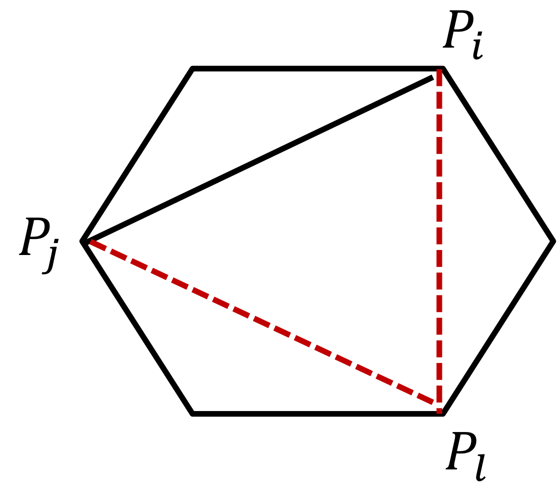
    $q = +\infty$

    for $l = i+1, \dots, j-1$ (mod n)

      $q = \min\{q,\ d(P_i, P_j) + MWT(i,l) + MWT(l,j)\}$

    $MWT[i,j] = q$

return $MWT[1, n]$



Thank You

Haifeng Xu
University of Virginia
hx4ad@virginia.edu