W10: Dynamic Programming

CS3230 AY21/22 Sem 2

Table of Contents

Click on the link to jump to the relevant sections!

DP: Motivation

Why learn DP?

- A design paradigm to improve solutions that run in **exponential time** down to **polynomial time**
- A good candidate for optimisation problems (find minimum / maximum)

Why learn DP?

- A design paradigm to improve solutions that run in **exponential time** down to **polynomial time**
- A good candidate for optimisation problems (find minimum / maximum)

- Applications:
  - Content-Aware Image Resizing (Graphics)
  - All-Pairs Shortest Paths (Routing)
  - Edit Distance (Auto-correct, DNA similarity)
  - Longest Common Subsequence (`diff` / `git diff`)
  - Parsing (See CYK algorithm)
  - Query Optimization (Databases)

Eg: Seam Carving (Content-Aware Image Resizing)

- Seam Carving Lecture by Grant Sanderson (3blue1brown)
  - Early parts talk about Computer Vision concepts
  - Skip ahead a bit for the DP idea

Eg: Seam Carving (Content-Aware Image Resizing)

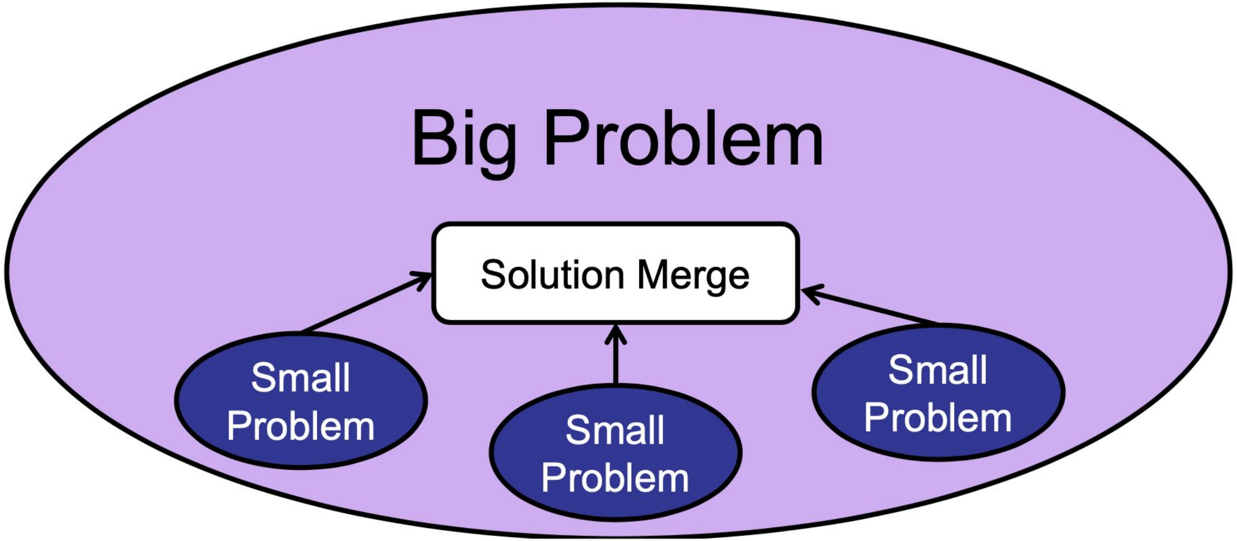
- Potential usage: object removal in images

Figure 11: Simple object removal: the user marks a region for removal (green), and possibly a region to protect (red), on the original image (see inset in left image). On the right image, consecutive vertical seam were removed until no 'green' pixels were left.

Using DP

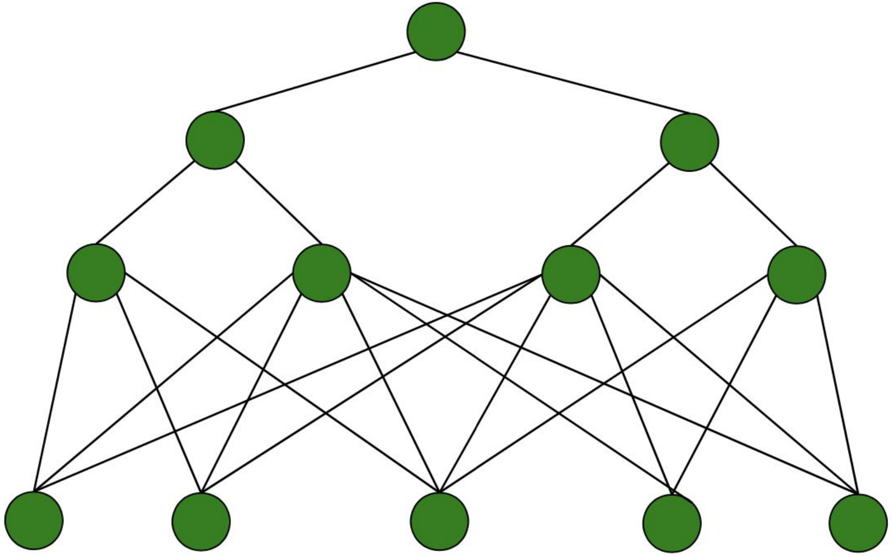
When to use DP?

Optimal substructure: Optimal solutions can be reconstructed from smaller subproblems



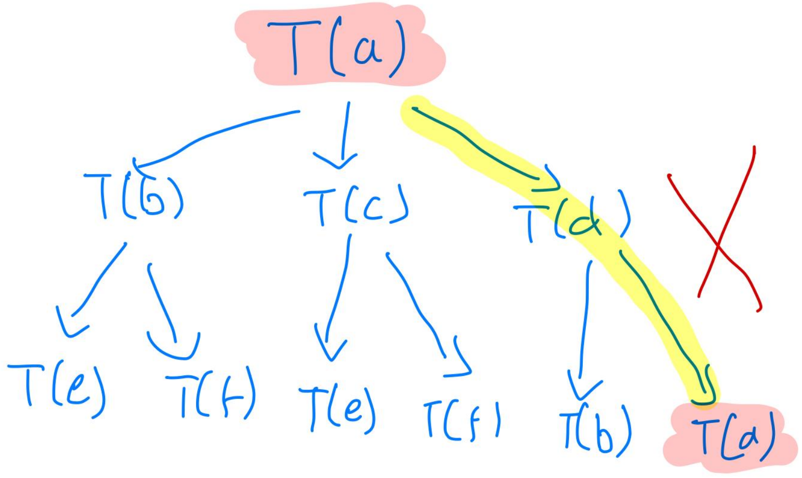
When to use DP?

Overlapping Subproblems: You keep reusing the subproblem to solve the bigger problem - so you store the result somewhere



When to use DP?

Dependency is Directed Acyclic: When computing a particular result, it must not depend on the result of itself



How to DP?

Brute Force, but *carefully*

How to DP?

Brute Force, but *carefully*

1. Identify the subproblems

How to DP?

Brute Force, but *carefully*

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems (Tip: **DON’T unroll the recursion**)

How to DP?

Brute Force, but *carefully*

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems (Tip: **DON'T unroll the recursion**)
3. Relate the smaller subproblem to the current subproblem
    a. Guess the relation!
    b. This might involve trying **all** subproblems!

How to DP?

Brute Force, but *carefully*

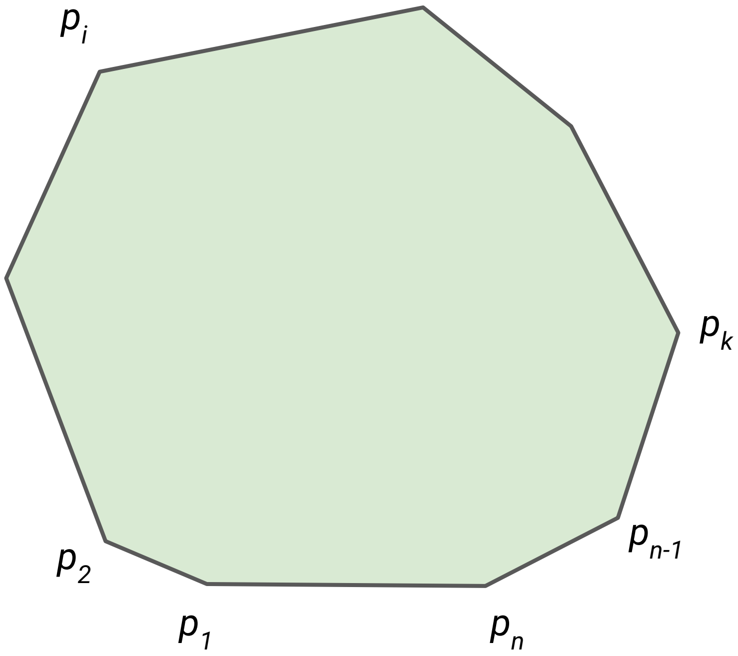
1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems (Tip: **DON'T unroll the recursion**)
3. Relate the smaller subproblem to the current subproblem
    a. Guess the relation!
    b. This might involve trying **all** subproblems!

- Your subproblem result might be re-used -- store it in a table!
- Time complexity: total time to compute **all subproblems**

Question 1: Recursive Formulation of Convex Polygon Triangulation

Note: The question slides use counter clockwise for the points, while this set of slides use clockwise
Techniques however, should apply without loss of generality

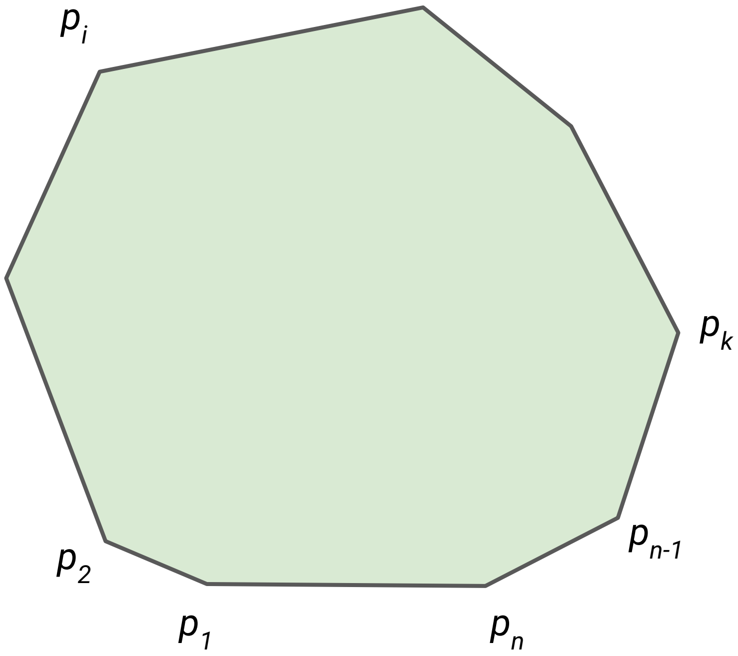
Convex Polygon



A polygon is represented by

$[p_1, p_2, \ldots, p_n]$ stored in array

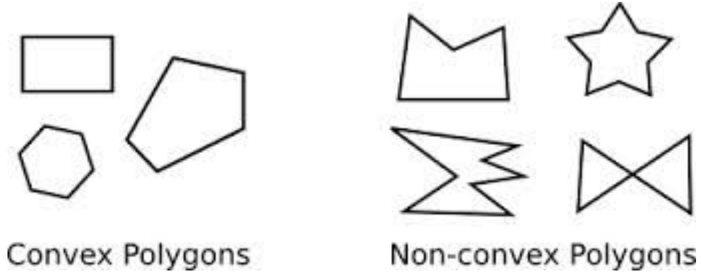
Convex Polygon



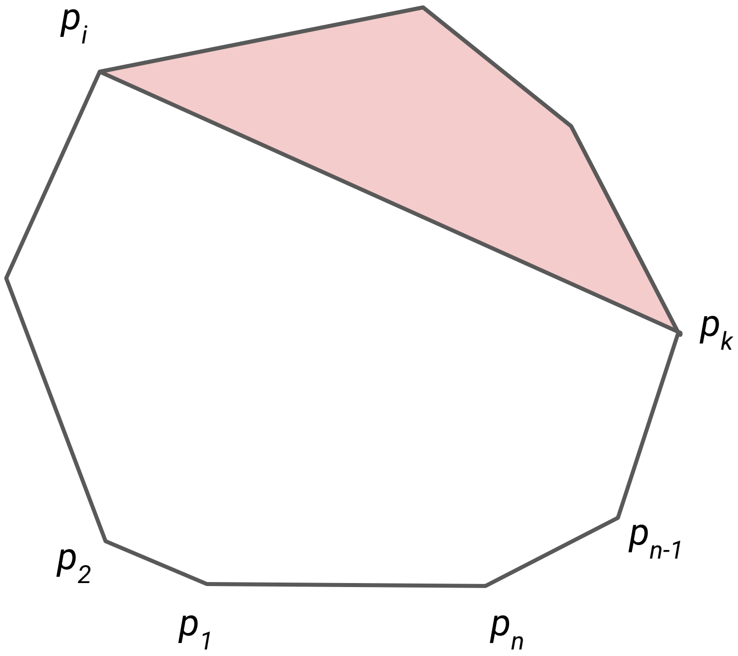
A polygon is represented by

`[`p_1`, `p_2`, …, `p_n`]` stored in array

We will deal only with Convex Polygon today



Convex Polygon



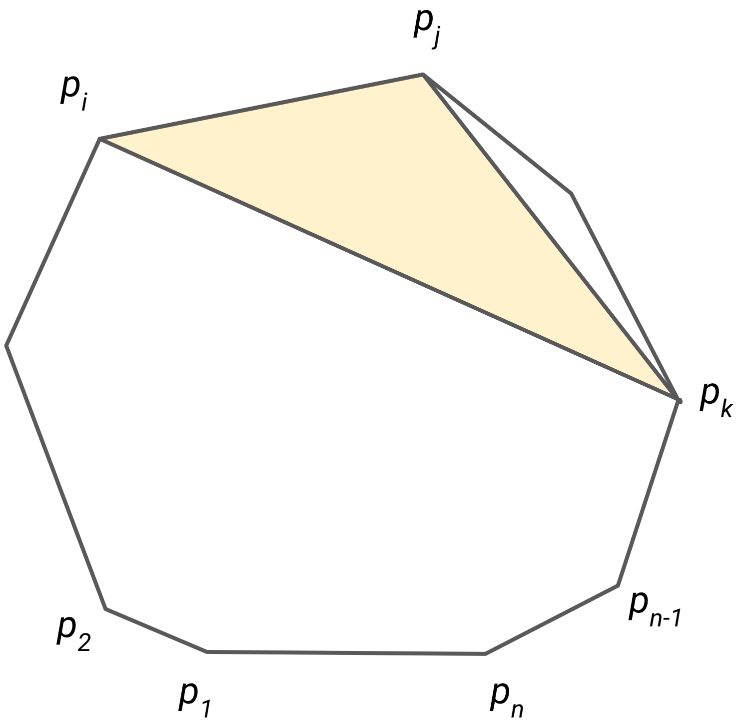
A polygon is represented by

`[`p_1`, `p_2`, …, `p_n`] stored in array`

`[`p_i`, …, `p_k`]`

`Polygon consisting of `p_i`, …, `p_k

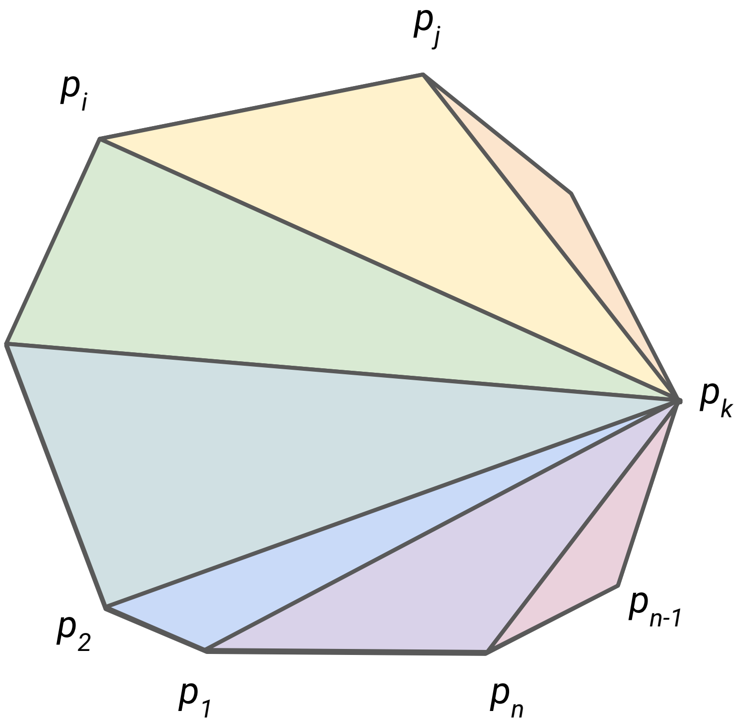
Triangulation of Convex Polygon



$w(i, j, k)$: Weight of triangle formed by (p_i, p_j, p_k)

Assume it takes $O(1)$ time to compute $w(i, j, k)$

Triangulation of Convex Polygon



$w(i, j, k)$: Weight of triangle formed by (p_i, p_j, p_k)

Assume it takes $O(1)$ time to compute $w(i, j, k)$

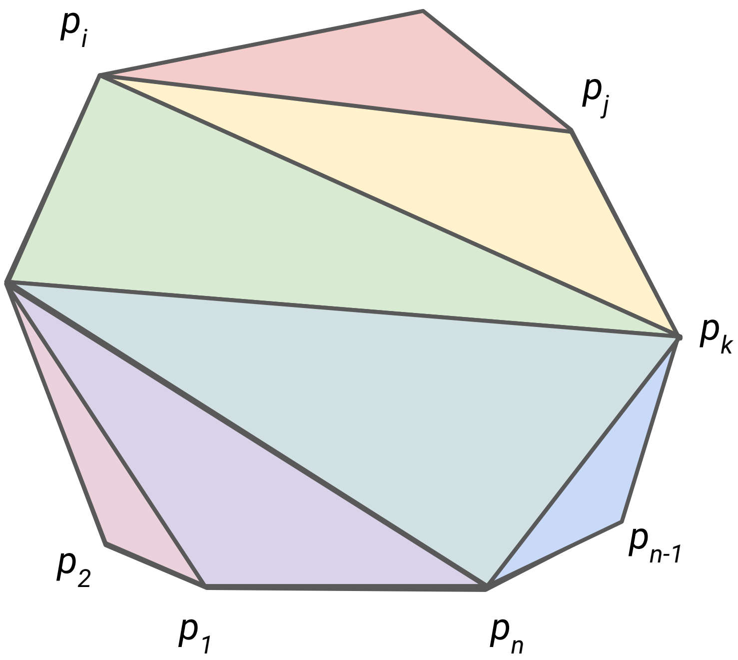
Cost of triangulation:

Sum of weights of the n - 2 triangles formed

(Why n-2? Try some simple examples to convince yourself. Otherwise, you can try to prove by induction on the number of points)

Triangulation of Convex Polygon

Another triangulation example!



$w(i, j, k)$: Weight of triangle formed by (p_i, p_j, p_k)

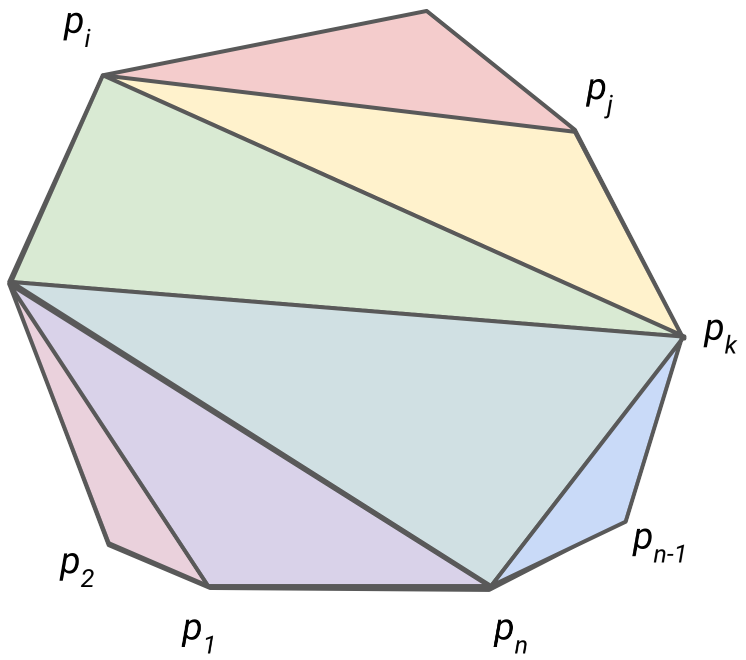
Assume it takes $O(1)$ time to compute $w(i, j, k)$

Cost of triangulation:

Sum of weights of the n - 2 triangles formed

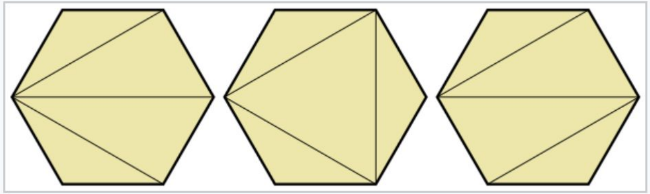
(Why n-2? Try some simple examples to convince yourself. Otherwise, you can try to prove by induction on the number of points)

Motivation of Triangulation



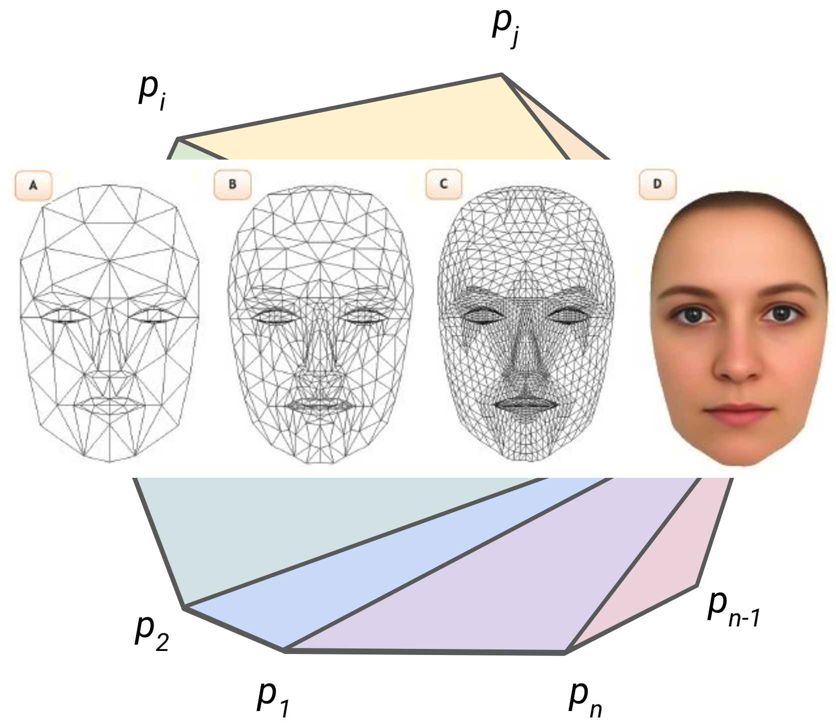
$w(i, j, k)$: Weight of triangle formed by (p_i, p_j, p_k)

Classic definition of weights: *the length of the triangle edges*



Three possible triangulations of the same polygon. Central triangulation has less weight (sum of perimeters).

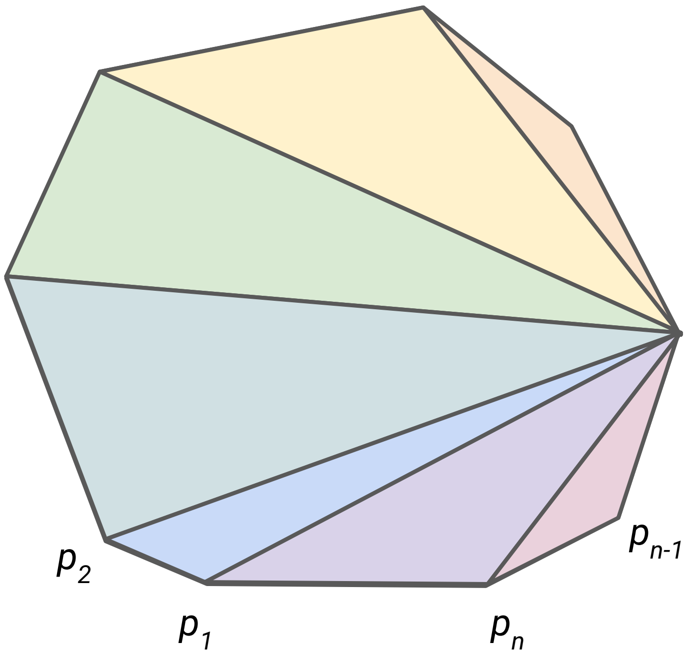
Motivation of Triangulation



$w(i, j, k)$: Weight of triangle formed by (p_i, p_j, p_k)

Triangles are used in computer graphics to **approximate smoothness of surfaces**

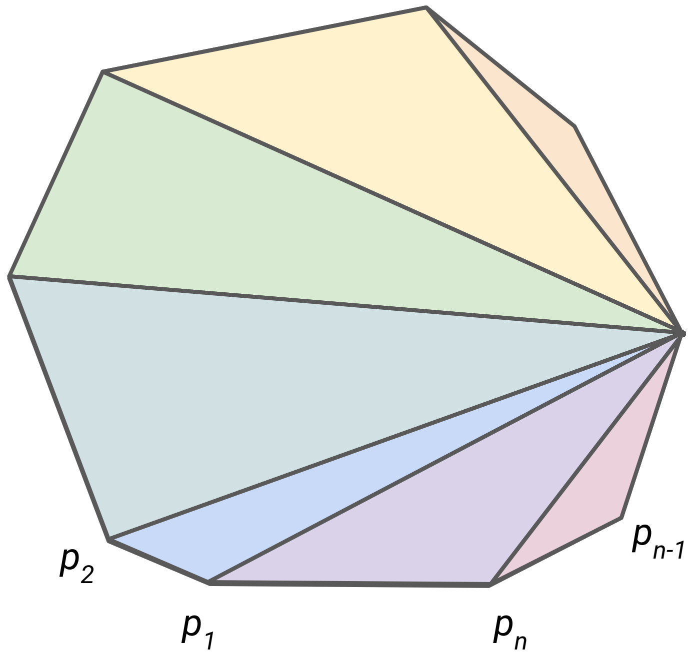
Question 1: Minimise cost



Given a convex polygon represented by, `[`p_1`, `p_2`, …, `p_n`]` find the triangulation with **minimum** cost

Let *c(i, j)*: cost of **optimal triangulation** of `[`p_i`, …, `p_j`]`

Question 1: Minimise cost



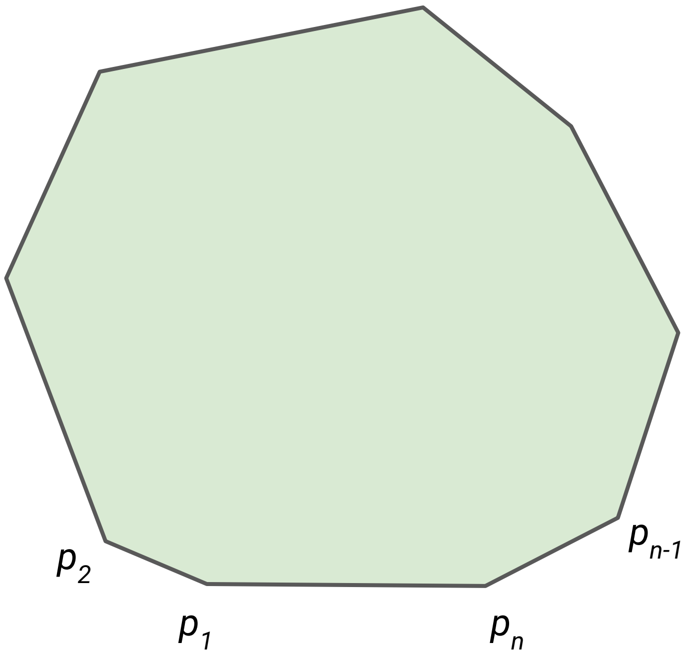
Given a convex polygon represented by, $[p_1, p_2, \ldots, p_n]$ find the triangulation with **minimum** cost

Let *c(i, j)*: cost of **optimal triangulation** of $[p_i, \ldots, p_j]$

Write down recursive formula for the above problem

Express *c(i, j)* in terms of *c(i', j')*, where *c(i', j')* represents a smaller polygon

Question 1: Minimise cost



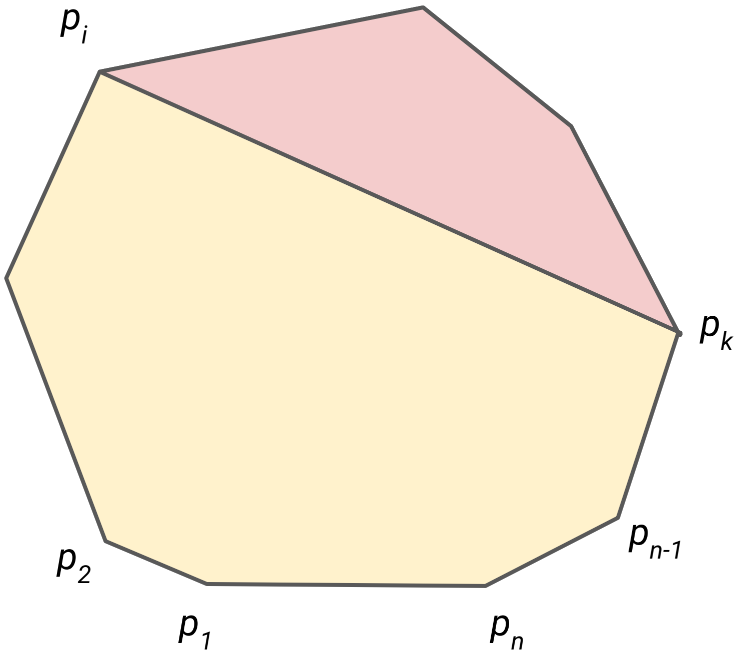
Given a convex polygon represented by, `[p`$_1$`, p`$_2$`, …, p`$_n$`]` find the triangulation with **minimum** cost

Let *c(i, j)*: cost of **optimal triangulation** of [`p`$_i$`, …, p`$_j$]

Write down recursive formula for the above problem

Express *c(i, j)* in terms of *c(i', j')*, where *c(i', j')* represents a smaller polygon
E.g. *c(1, n)*

Question 1: Minimise cost



Given a convex polygon represented by, $[p_1, p_2, \ldots, p_n]$ find the triangulation with **minimum** cost

Let $c(i, j)$: cost of **optimal triangulation** of $[p_i, \ldots, p_j]$

Write down recursive formula for the above problem

Express $c(i, j)$ in terms of $c(i', j')$, where $c(i', j')$ represents a smaller polygon
E.g. $c(1, n) =?= c(i, k) + c(k, i)$

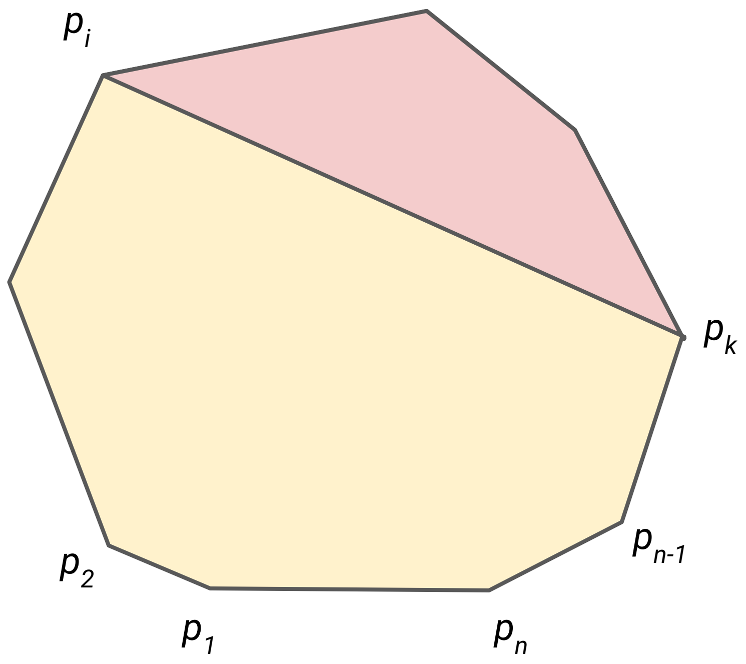
How to DP?

Brute Force, but *carefully*

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
   a. Guess the relation!
   b. This might involve trying **all** subproblems!

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
    a. Guess the relation!
    b. This might involve trying **all** subproblems!

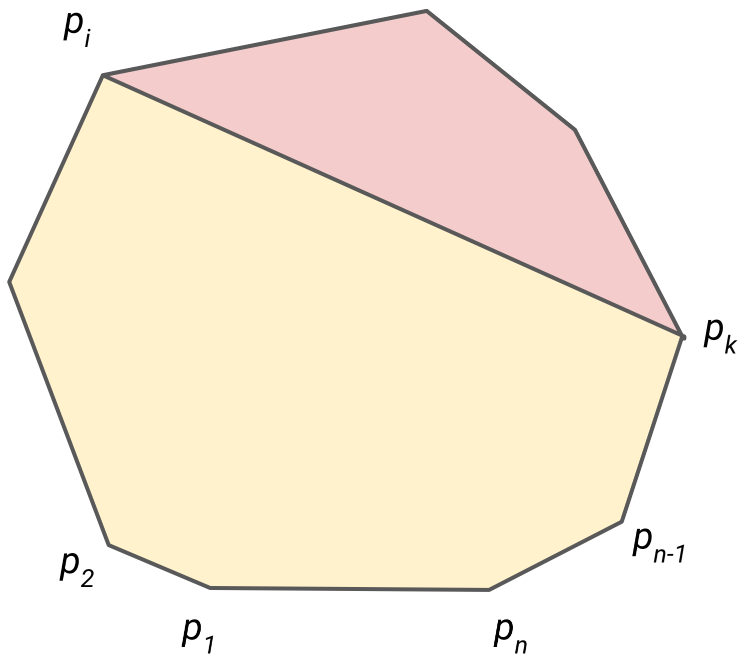


Let *c(i, j)*: cost of **optimal triangulation** of $[p_i, \ldots, p_j]$

The question was nice enough to give you how the subproblem should be formulated!

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
    a. Guess the relation!
    b. This might involve trying **all** subproblems!



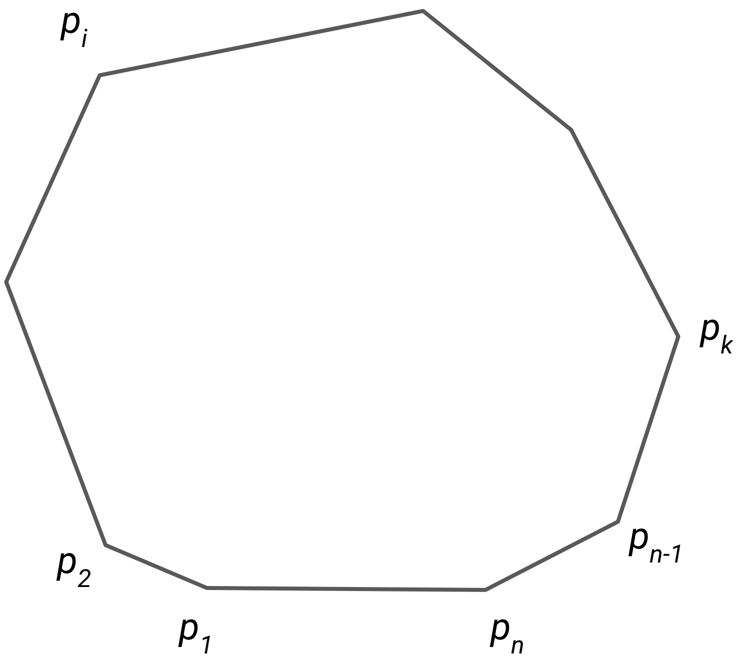
Let *c(i, j)*: cost of **optimal triangulation** of $[p_i, \ldots, p_j]$

The question was nice enough to give you how the subproblem should be formulated!

e.g. Assume we know that *c(i, k) = 10*, and *c(k, i) = 30*. Can we use it to compute *c(1, n)*?

Question 1 Idea

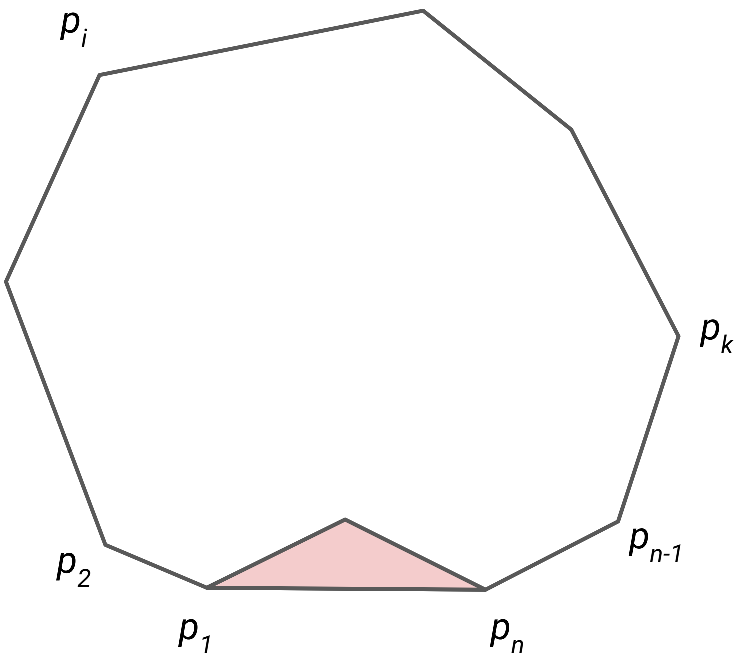
1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
   a. Guess the relation!
   b. This might involve trying **all** subproblems!



Which subproblems are appropriate for this problem? (we must cover all the triangulation)

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
   a. Guess the relation!
   b. This might involve trying **all** subproblems!

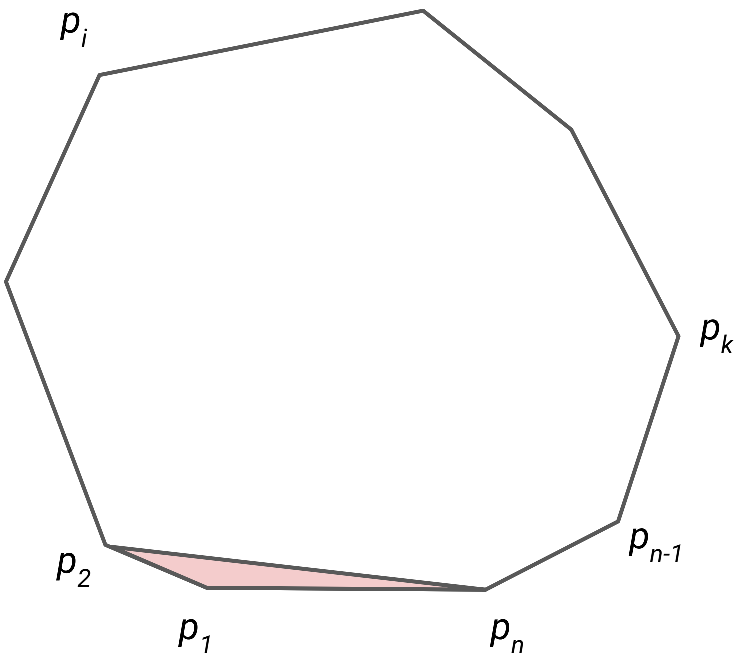


Which subproblems are appropriate for this problem? (we must cover all the triangulation)

Idea: Guess where the triangle with side (p_1, p_n) should be!

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
    a. Guess the relation!
    b. This might involve trying **all** subproblems!



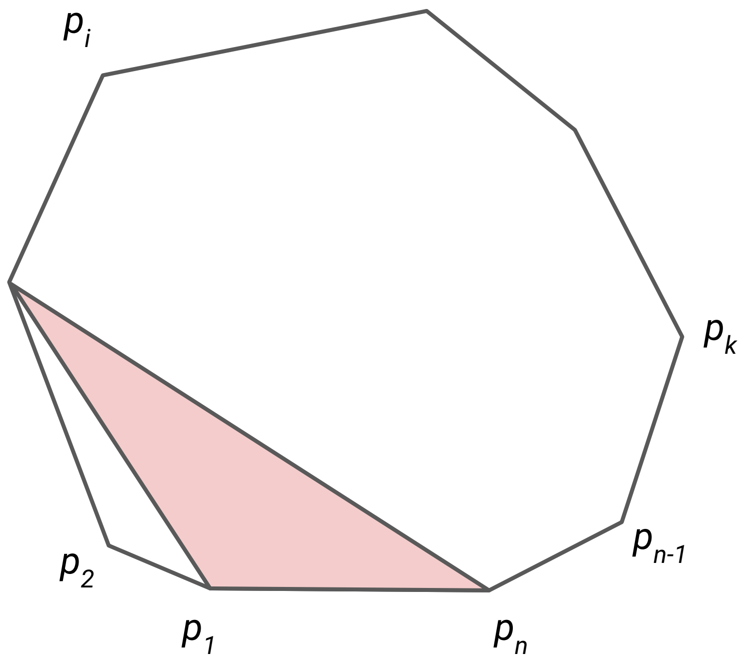
Which subproblems are appropriate for this problem? (we must cover all the triangulation)

Idea: Guess where the triangle with side (p_1, p_n) should be!

It could be here #1

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
    a. Guess the relation!
    b. This might involve trying **all** subproblems!



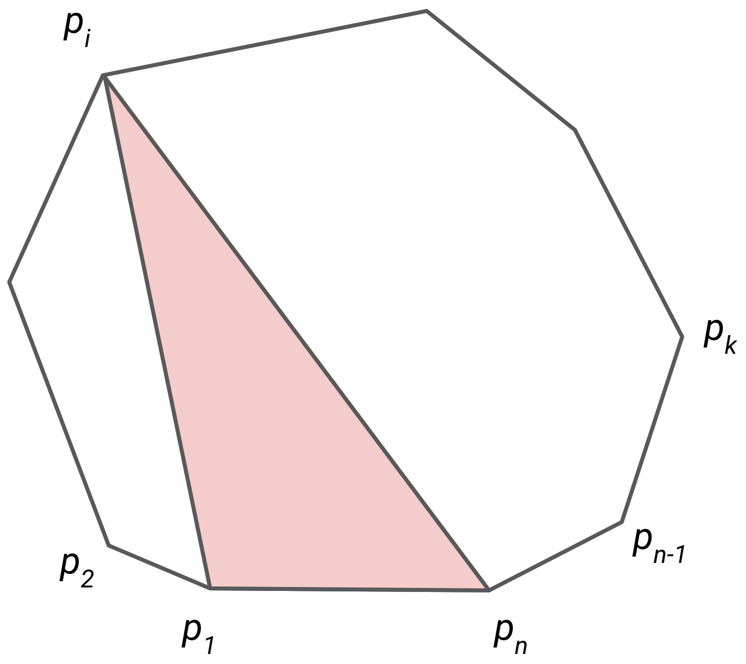
Which subproblems are appropriate for this problem? (we must cover all the triangulation)

Idea: Guess where the triangle with side (p_1, p_n) should be!

Or it could be here #2

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
   a. Guess the relation!
   b. This might involve trying **all** subproblems!



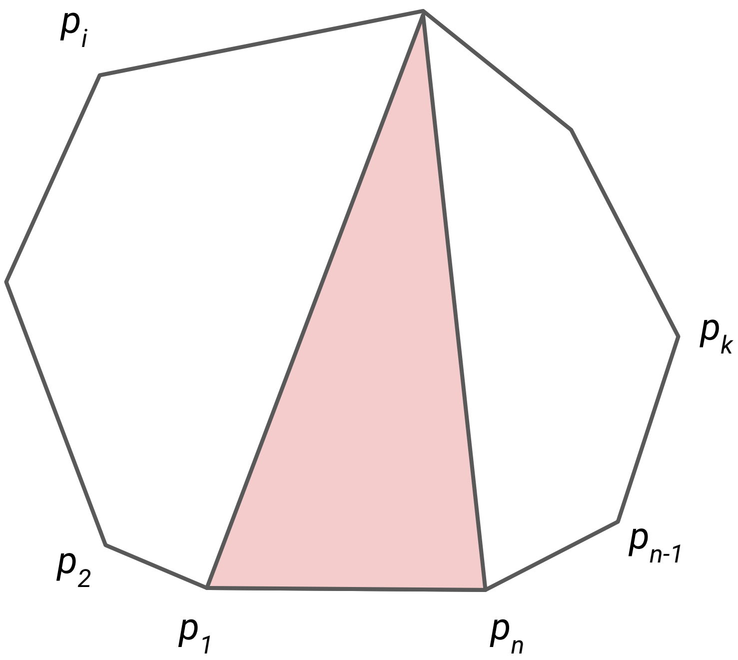
Which subproblems are appropriate for this problem? (we must cover all the triangulation)

Idea: Guess where the triangle with side (p_1 , p_n) should be!

Or it could be here #3

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
    a. Guess the relation!
    b. This might involve trying **all** subproblems!



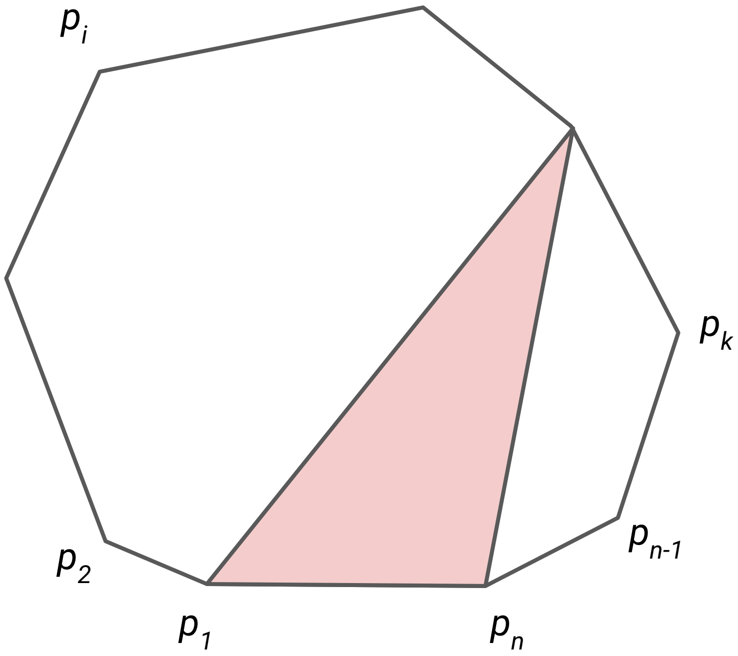
Which subproblems are appropriate for this problem? (we must cover all the triangulation)

Idea: Guess where the triangle with side (p_1 , p_n) should be!

Or it could be here #4

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
    a. Guess the relation!
    b. This might involve trying **all** subproblems!



Which subproblems are appropriate for this problem? (we must cover all the triangulation)

Idea: Guess where the triangle with side (p_1 , p_n) should be!

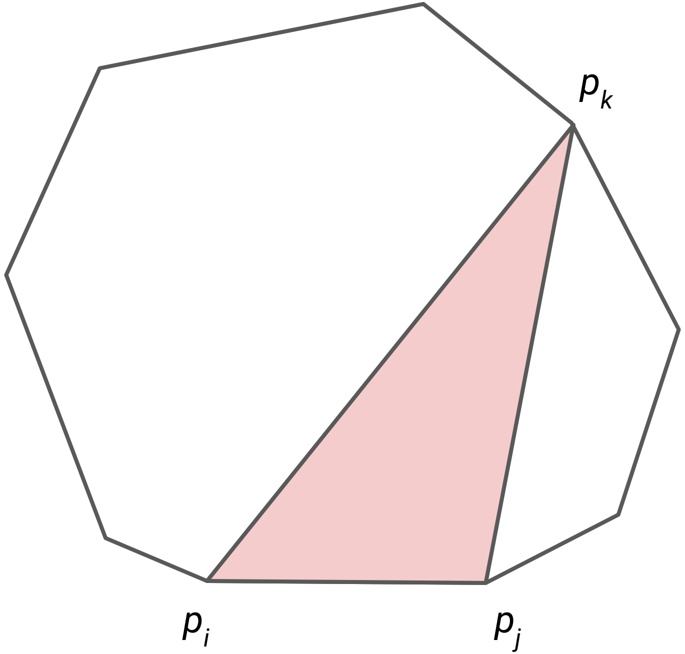
Or it could be here #5

and so on...

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
    a. Guess the relation!
    b. This might involve trying **all** subproblems!

- Let $c(i, j)$: cost of **optimal triangulation** of $[p_i, \ldots, p_j]$
- $w(i, j, k)$: Weight of triangle formed by (p_i, p_j, p_k)



Which subproblems are appropriate for this problem? (we must cover all the triangulation)

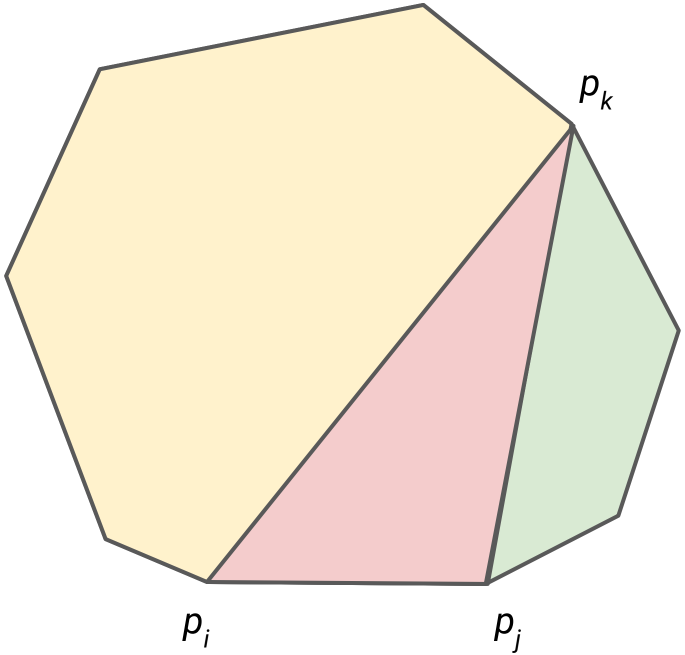
Idea: Guess where the triangle with side (p_1, p_n) should be!

Recurrence for one triangle:

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
   a. Guess the relation!
   b. This might involve trying **all** subproblems!

- Let $c(i, j)$: cost of **optimal triangulation** of $[p_i, \ldots, p_j]$
- $w(i, j, k)$: Weight of triangle formed by (p_i, p_j, p_k)



Which subproblems are appropriate for this problem? (we must cover all the triangulation)

Idea: Guess where the triangle with side (p_1, p_n) should be!

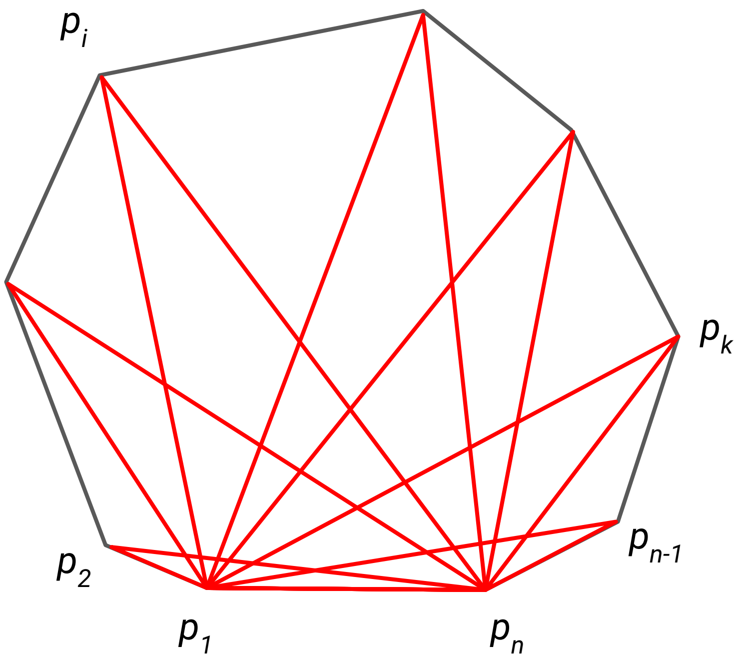
Recurrence for one triangle:

$c(i, k) + w(i, k, j) + c(k, j)$

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
   a. Guess the relation!
   b. This might involve trying **all** subproblems!

- Let $c(i, j)$: cost of **optimal triangulation** of $[p_i, \ldots, p_j]$
- $w(i, j, k)$: Weight of triangle formed by (p_i, p_j, p_k)



Which subproblems are appropriate for this problem? (we must cover all the triangulation)

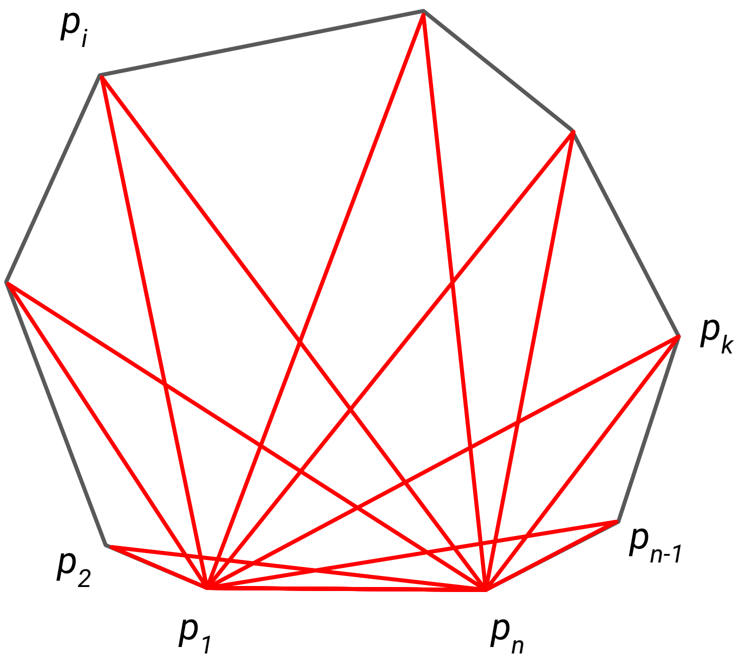
Idea: Guess where the triangle with side (p_1, p_n) should be!

Which triangle to use?

Question 1 Idea

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
   a. Guess the relation!
   b. This might involve trying **all** subproblems!

- Let $c(i, j)$: cost of **optimal triangulation** of $[p_i, \ldots, p_j]$
- $w(i, j, k)$: Weight of triangle formed by (p_i, p_j, p_k)



Which subproblems are appropriate for this problem? (we must cover all the triangulation)

Idea: Guess where the triangle with side (p_1, p_n) should be!

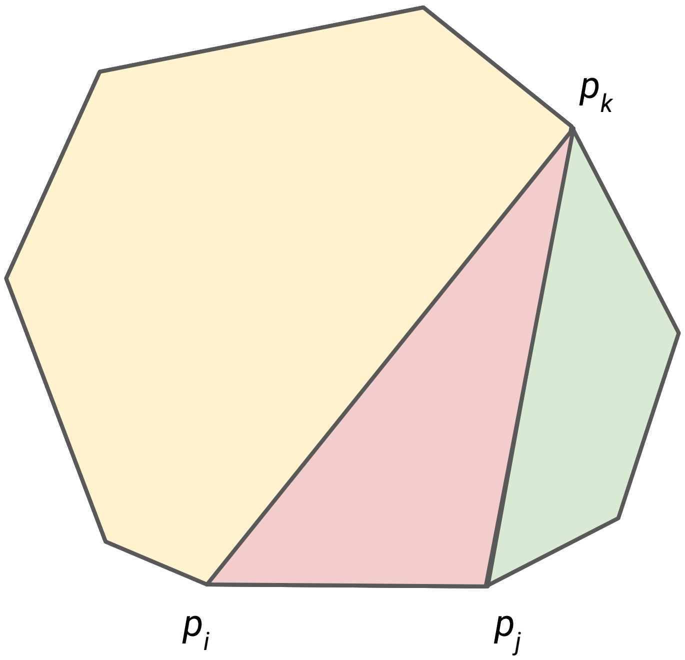
Which triangle to use?

Try them all!

Question 1 Soln

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
   a. Guess the relation!
   b. This might involve trying **all** subproblems!

- Let $c(i, j)$: cost of **optimal triangulation** of $[p_i, \ldots, p_j]$
- $w(i, j, k)$: Weight of triangle formed by (p_i, p_j, p_k)



if $j > i + 1$:

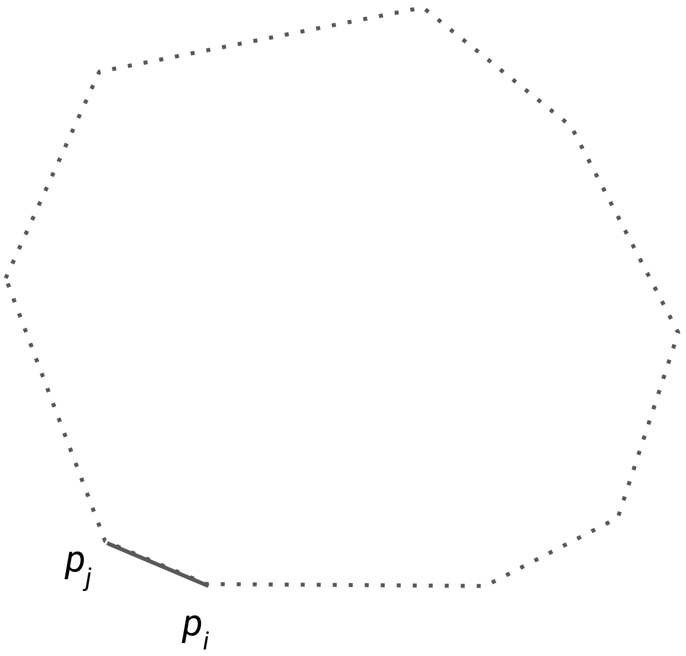
$min(c(i, k) + w(i, k, j) + c(k, j))$

for all $i < k < j$

Question 1 Soln

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems
3. Relate the smaller subproblem to the current subproblem
   a. Guess the relation!
   b. This might involve trying **all** subproblems!

- Let $c(i, j)$: cost of **optimal triangulation** of $[p_i, \ldots, p_j]$
- $w(i, j, k)$: Weight of triangle formed by (p_i, p_j, p_k)

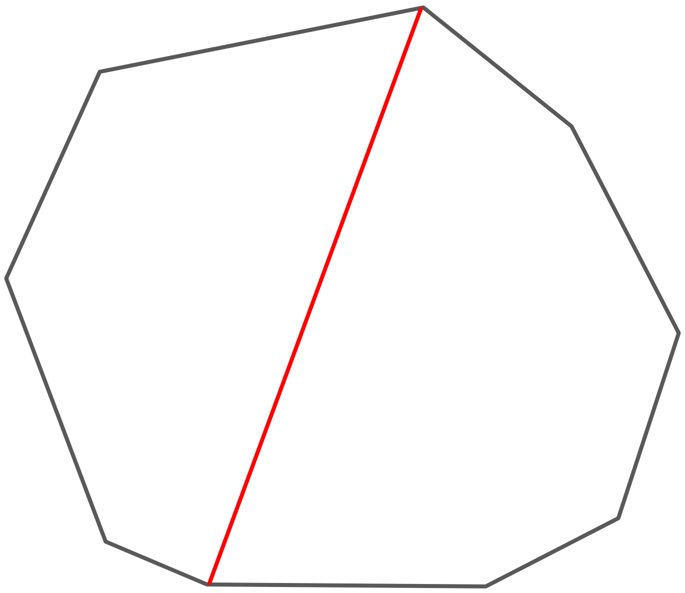


if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$\min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

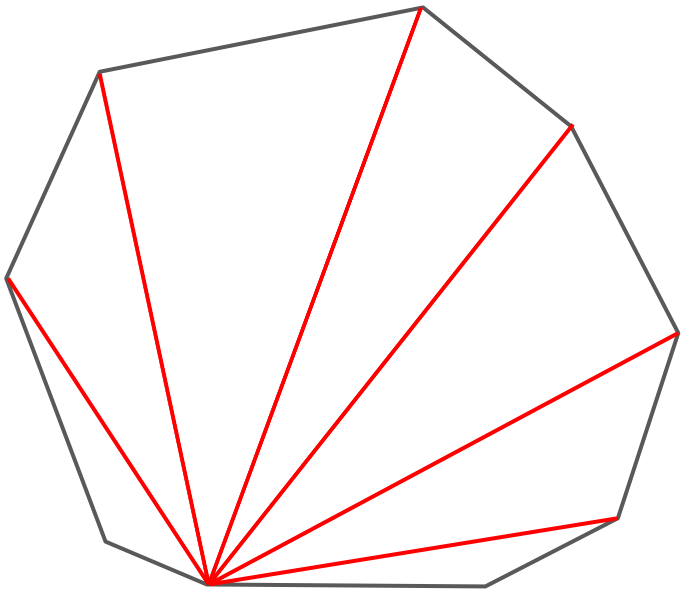
Alternative #1?



Strategy:

1. Choose an anchor point
2. Draw a dividing line
3. Find cost of both polygons and sum

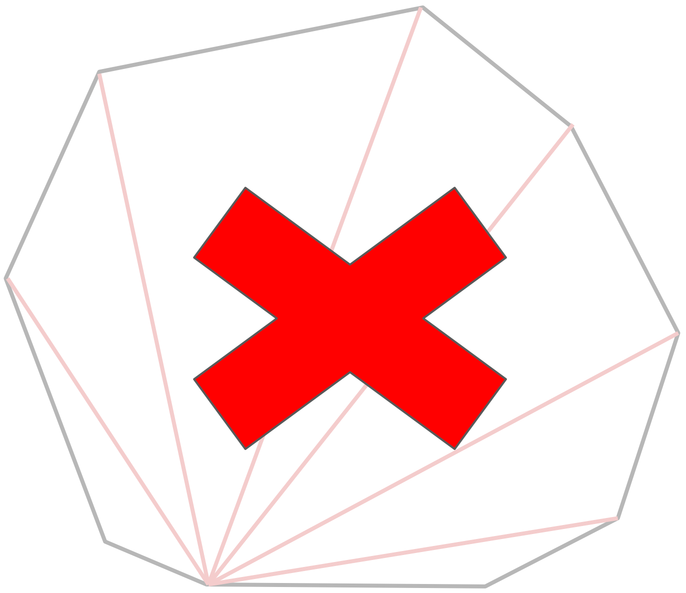
Alternative #1?



Strategy:

1. Choose an anchor point
2. Draw a dividing line
    a. Try them all and find smallest
3. Find cost of both polygons and sum

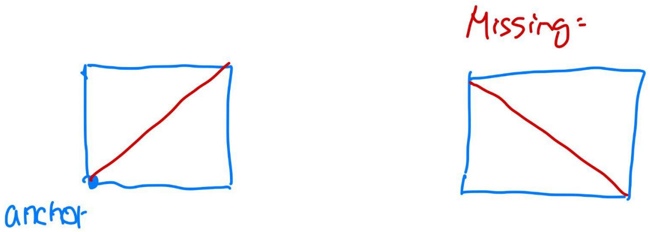
Alternative #1?



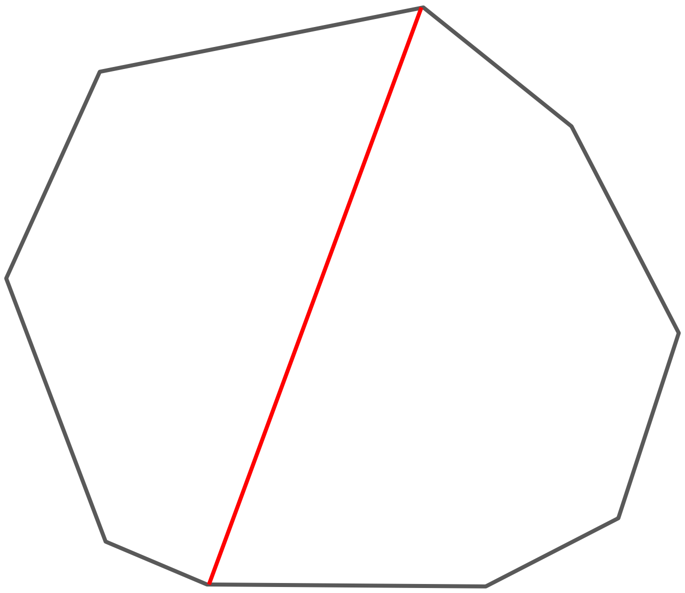
~~Strategy:~~

1. ~~Choose an anchor point~~
2. ~~Draw a dividing line~~
   a. ~~Try them all and find smallest~~
3. ~~Find cost of both polygons and sum~~

Does not cover all! Will miss some triangulations:



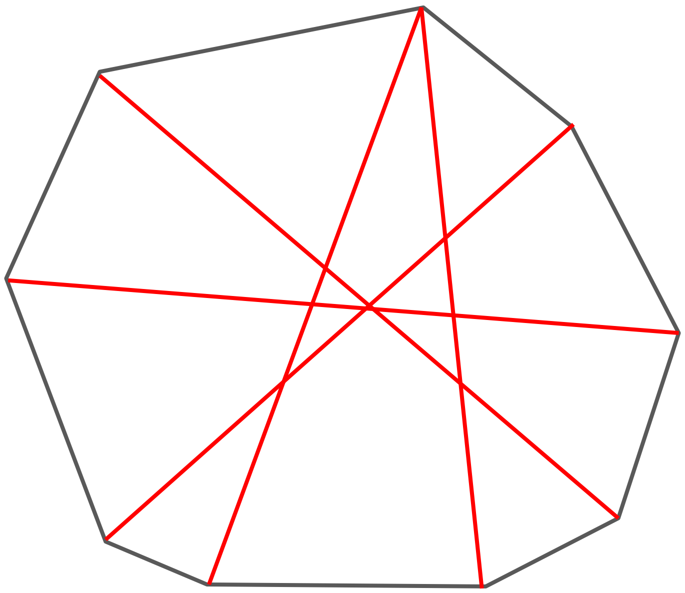
Alternative #2?



Strategy:

1. Choose an anchor point
2. Draw a dividing line
3. Find cost of both polygons and sum

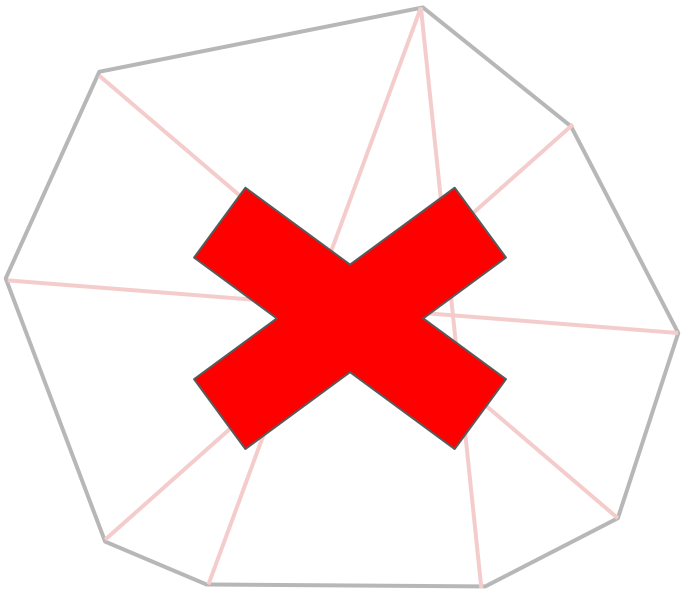
Alternative #2?



Strategy:

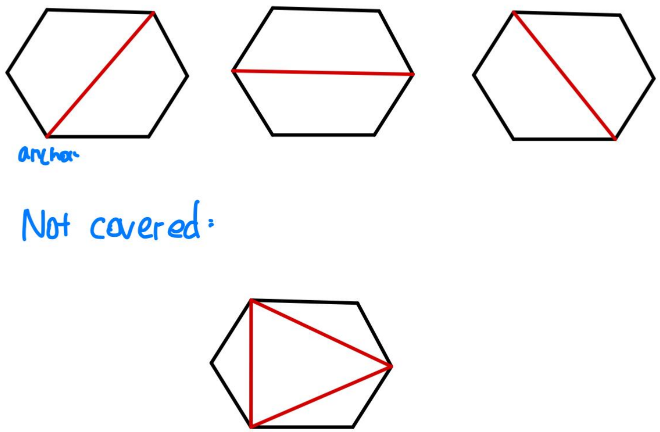
1. Choose an anchor point
2. Draw a dividing line
   a. Rotate clockwise, try them all and find smallest
3. Find cost of both polygons and sum

Alternative #2?

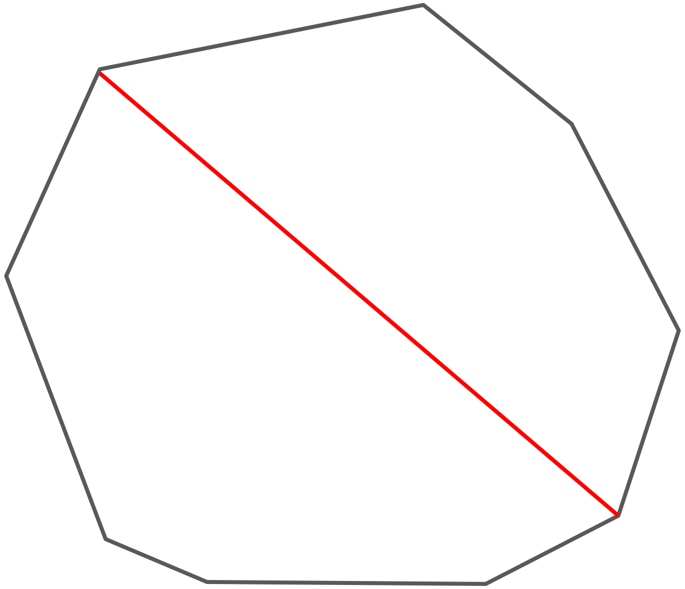


Again: does **not** cover all!

(This method seems to be fine up to a Pentagon)



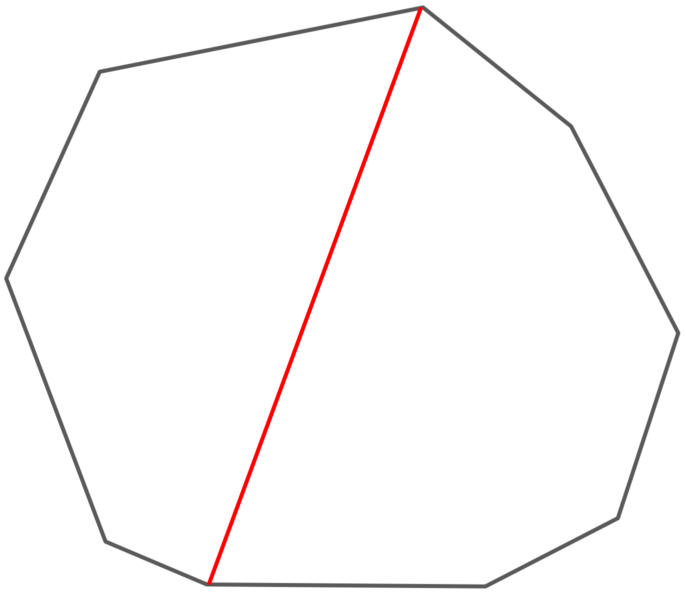
Alternative #3?



Strategy:

1. Try all the dividing lines!
2. Find cost of both polygons that got divided and sum
   a. Take the minimum over everything

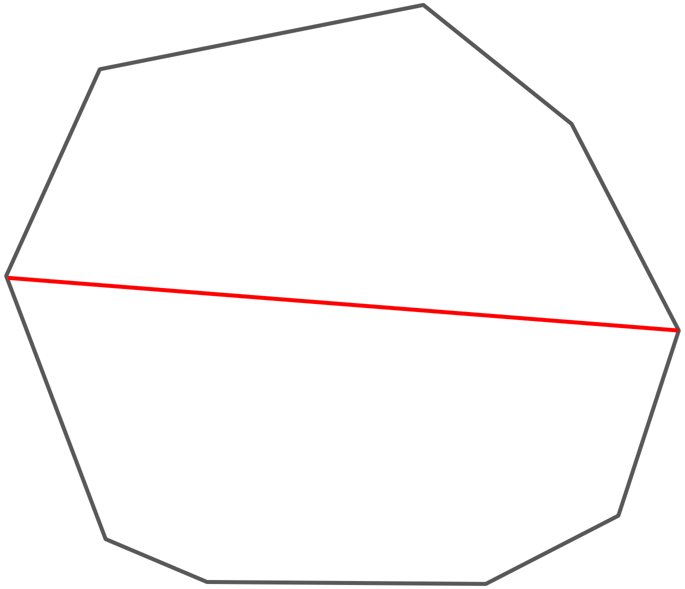
Alternative #3?



Strategy:

1. Try all the dividing lines!
2. Find cost of both polygons that got divided and sum
   a. Take the minimum over everything

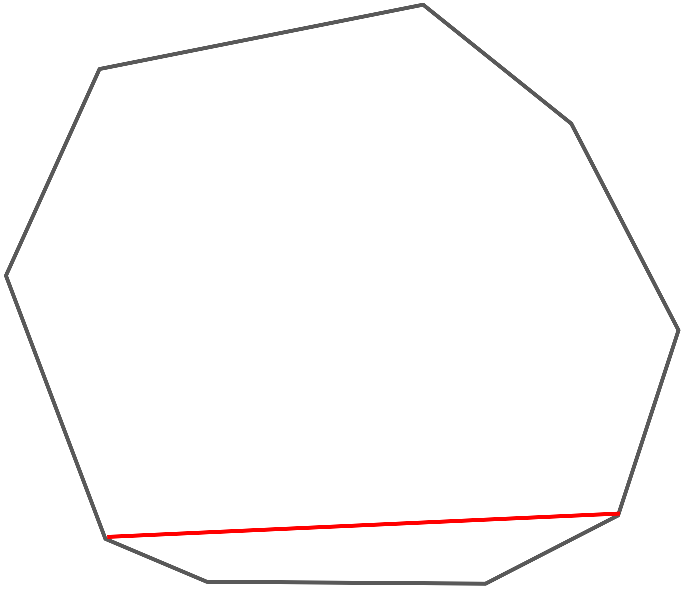
Alternative #3?



Strategy:

1. Try all the dividing lines!
2. Find cost of both polygons that got divided and sum
   a. Take the minimum over everything

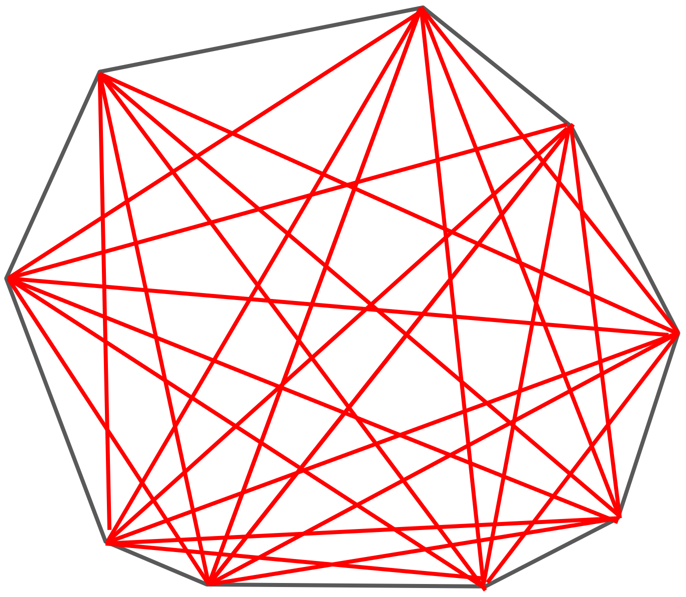
Alternative #3?



Strategy:

1. Try all the dividing lines!
2. Find cost of both polygons that got divided and sum
   a. Take the minimum over everything

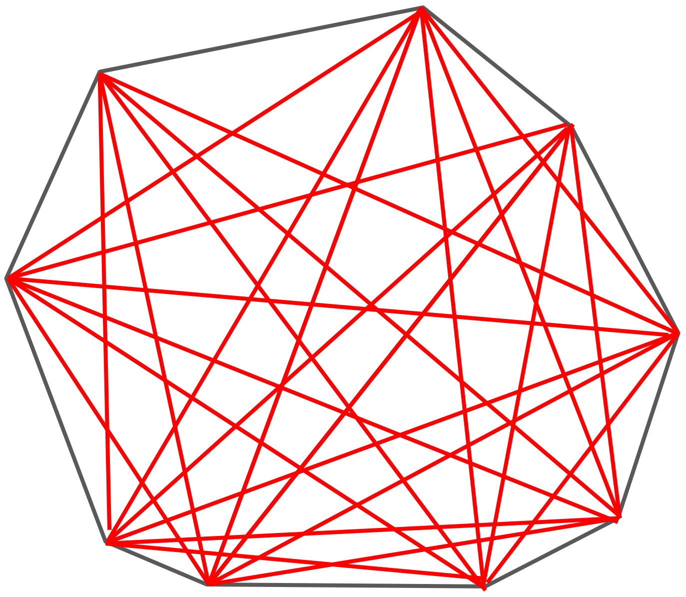
Alternative #3?



Strategy:

1. Try all the dividing lines!
2. Find cost of both polygons that got divided and sum
   a. Take the minimum over everything

Alternative #3?



Strategy:

1. Try all the dividing lines!
2. Find cost of both polygons that got divided and sum
    a. Take the minimum over everything

This one is ok! We are literally brute-forcing all the possible cuts. However, this is inefficient (we will analyse later)

Lesson Learnt

Brute Force, but *carefully*

Lesson Learnt

Brute Force, but *carefully*

- Different possible ways to divide the subproblems and form the recursive formulation!
    - Some are more efficient than the other

Lesson Learnt

Brute Force, but *carefully*

- Different possible ways to divide the subproblems and form the recursive formulation!
    - Some are more efficient than the other

- Need to be careful and ensure that all possibilities are accounted for
    - There are problems where you don’t need to account all possibilities (Greedy Algorithms), but you need to prove that you can ignore those possibilities

Question 2: Triangulation Running Time

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, i))$
for all $i < k < j$

Q2: 3 mins to fill into Archipelago

```
def compute_cost(i, j):
  if (j == i + 1):
    return 0
  else:
    cost = INF # dummy
    for (i < k < j):
      curr = compute_cost(i, k)
             + w(i, k, j)
             + compute_cost(k, j)
      # take the better result
      cost = min(cost, curr)
    return cost
```

Essentially the recurrence we had in Q1!

What is the running time?

1. $2^{O(j - i)}$
2. $O((j - i)^2)$
3. $O((j - i)^3)$

n points

Let $T(n)$ = compute_cost$(1, n)$

runtime of ^

Q2 Recurrence

Some definitions to simplify

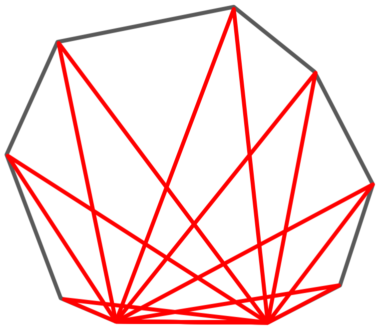
```
def compute_cost(i, j):
if (j == i + 1):
  return 0
else:
  cost = INF # dummy
  for (i < k < j):
    curr = compute_cost(i, k)
           + w(i, k, j)
           + compute_cost(k, j)
    # take the better result
    cost = min(cost, curr)
  return cost
```

n points

Let $T(n) = \text{compute\_cost}(1, n)$

runtime of ^

$$
\begin{aligned}
T(n) = \; & T(2) + T(n-1) + c \\
& + T(3) + T(n-2) + c \\
& \vdots \\
& + T(n-2) + T(3) + c \\
& + T(n-1) + T(2) + c
\end{aligned}
$$



Q2 Recurrence

Expand the recurrence based on all the triangulation

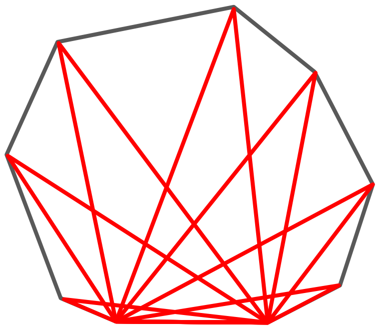
```
def compute_cost(i, j):
if (j == i + 1):
  return 0
else:
  cost = INF # dummy
  for (i < k < j):
    curr = compute_cost(i, k)
           + w(i, k, j)
           + compute_cost(k, j)
    # take the better result
    cost = min(cost, curr)
  return cost
```

n points

Let $T(n) = \text{compute\_cost}(1, n)$

runtime of ^

$$
\begin{aligned}
T(n) = \; & T(2) + T(n-1) + c \\
+ \; & T(3) + T(n-2) + c \\
& \vdots \\
+ \; & T(n-2) + T(3) + c \\
+ \; & T(n-1) + T(2) + c
\end{aligned}
$$



Q2 Recurrence

Notice that every smaller recurrence occurs twice!

```
def compute_cost(i, j):
  if (j == i + 1):
    return 0
  else:
    cost = INF # dummy
    for (i < k < j):
      curr = compute_cost(i, k)
           + w(i, k, j)
           + compute_cost(k, j)
      # take the better result
      cost = min(cost, curr)
    return cost
```

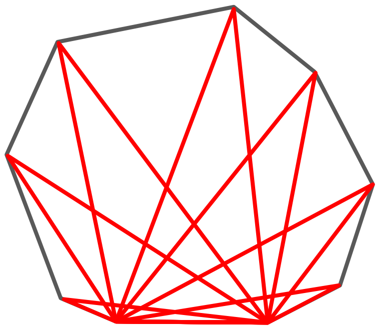
n points

Let $T(n)$ = compute_cost(1, n)

runtime of ^

$$
\begin{aligned}
T(n) = {} & T(2) + T(n-1) + c \\
& + T(3) + T(n-2) + c \\
& \quad \vdots \\
& + T(n-2) + T(3) + c \\
& + T(n-1) + T(2) + c
\end{aligned}
$$

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



Q2 Recurrence

Can be compacted in this way

```
def compute_cost(i, j):
if (j == i + 1):
  return 0
else:
  cost = INF # dummy
  for (i < k < j):
    curr = compute_cost(i, k)
           + w(i, k, j)
           + compute_cost(k, j)
    # take the better result
    cost = min(cost, curr)
  return cost
```

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

Q2 Recurrence

From previous slide

```
def compute_cost(i, j):
if (j == i + 1):
  return 0
else:
  cost = INF # dummy
  for (i < k < j):
    curr = compute_cost(i, k)
           + w(i, k, j)
           + compute_cost(k, j)
    # take the better result
    cost = min(cost, curr)
  return cost
```

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

$$T(n-1) = 2\sum_{i=2}^{n-2} T(i) \quad + c(n-3)$$

Q2 Recurrence

Substitute n-1 into the recurrence (or think about one value less)

```
def compute_cost(i, j):
  if (j == i + 1):
    return 0
  else:
    cost = INF # dummy
    for (i < k < j):
      curr = compute_cost(i, k)
             + w(i, k, j)
             + compute_cost(k, j)
      # take the better result
      cost = min(cost, curr)
    return cost
```

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

$$T(n-1) = 2\sum_{i=2}^{n-2} T(i) \quad + c(n-3) \qquad -$$

$$T(n) - T(n-1) = 2T(n-1) \quad + c$$

Q2 Recurrence

Notice how all the sums cancel off!
Everything from i = 2 to n-2

```
def compute_cost(i, j):
if (j == i + 1):
  return 0
else:
  cost = INF # dummy
  for (i < k < j):
    curr = compute_cost(i, k)
           + w(i, k, j)
           + compute_cost(k, j)
    # take the better result
    cost = min(cost, curr)
  return cost
```

$$T(n) = 2\sum_{i=2}^{n-1} T(i) + c(n-2)$$

$$T(n-1) = 2\sum_{i=2}^{n-2} T(i) + c(n-3)$$

$-$

$$T(n) - T(n-1) = 2T(n-1) + c$$

$$T(n) = 3T(n-1) + c$$

Q2 Recurrence

Rearranging equation

```
def compute_cost(i, j):
01. if (j == i + 1):
02.   return 0
03. else:
04.   cost = INF # dummy
05.   for (i < k < j):
06.     curr = compute_cost(i, k)
07.            + w(i, k, j)
08.            + compute_cost(k, j)
09.     # take the better result
10.     cost = min(cost, curr)
11.   return cost
```

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

$$T(n-1) = 2\sum_{i=2}^{n-2} T(i) \quad + c(n-3)$$

$-$

$$T(n) - T(n-1) = 2T(n-1) \quad + c$$

$$T(n) = 3T(n-1) \quad + c$$

$$T(n) = O(3^n) = 2^{O(n)}$$

Q2 Recurrence

Point is: you get exponential time -- bad!

```
def compute_cost(i, j):
if (j == i + 1):
  return 0
else:
  cost = INF # dummy
  for (i < k < j):
    curr = compute_cost(i, k)
           + w(i, k, j)
           + compute_cost(k, j)
    # take the better result
    cost = min(cost, curr)
  return cost
```

Question 3: Triangulation Sub-Problems

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, i))$
for all $i < k < j$

Q3: 3 mins to fill into Archipelago

```
def compute_cost(i, j):
if (j == i + 1):
  return 0
else:
  cost = INF # dummy
  for (i < k < j):
    curr = compute_cost(i, k)
           + w(i, k, j)
           + compute_cost(k, j)
    # take the better result
    cost = min(cost, curr)
  return cost
```

Consider the previous `compute_cost(1, n)` algorithm. Which one of the following is/are true?

1. `compute_cost(1, n)` computes 2^n different sub-problems
2. `compute_cost(1, n)` computes only at most n^2 different sub-problems, but to compute each sub-problem (non-recursively) it takes $\Omega(\frac{2^n}{n^2})$ time
3. `compute_cost(1, n)` computes only at most n^2 different sub−problems, but each sub-problem multiple times.

Q3 Soln

1. `compute_cost(1, n)` computes 2^n different sub-problems
2. `compute_cost(1, n)` computes only at most n^2 different sub-problems, but to compute each sub-problem (non-recursively) it takes $\Omega(\frac{2^n}{n^2})$ time
3. `compute_cost(1, n)` computes only at most n^2 different sub−problems, but each sub-problem multiple times.

Ans: 3. `compute_cost(1, n)` computes only at most n^2 different sub−problems, but each sub-problem multiple times.

$T(1, n)$: compute_cost(1,n)

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

Q3 Soln

1. `compute_cost(1, n)` computes 2^n different sub-problems
2. `compute_cost(1, n)` computes only at most n^2 different sub-problems, but to compute each sub-problem (non-recursively) it takes $\Omega(\frac{2^n}{n^2})$ time
3. `compute_cost(1, n)` computes only at most n^2 different sub−problems, but each sub-problem multiple times.

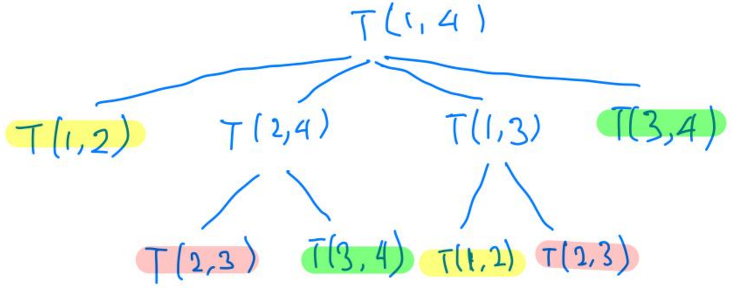
Ans: 3. `compute_cost(1, n)` computes only at most n^2 different sub−problems, but each sub-problem multiple times.

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

$T(1, n)$: compute_cost$(1, n)$



There are 6 distinct subproblems $= \binom{4}{2}$

1. `compute_cost(1, n)` computes 2^n different sub-problems

2. `compute_cost(1, n)` computes only at most n^2 different sub-problems, but to compute each sub-problem (non-recursively) it takes $\Omega(\frac{2^n}{n^2})$ time

3. `compute_cost(1, n)` computes only at most n^2 different sub−problems, but each sub-problem multiple times.

Q3 Soln

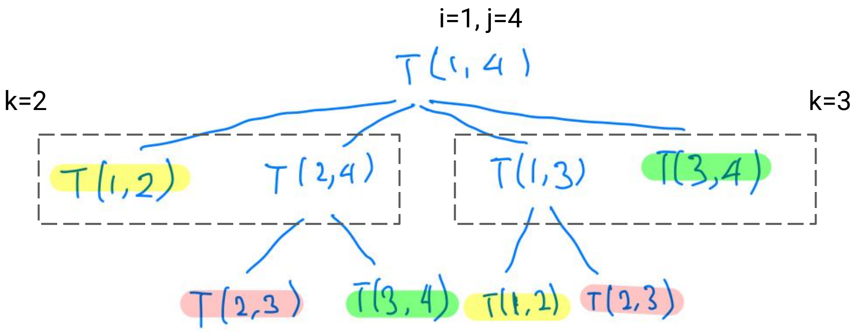
Ans: 3. `compute_cost(1, n)` computes only at most n^2 different sub−problems, but each sub-problem multiple times.

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

T(1, n) : compute_cost(1,n)



There are 6 distinct subproblems $= \binom{4}{2}$

Q3 Soln

1. `compute_cost(1, n)` computes 2^n different sub-problems
2. `compute_cost(1, n)` computes only at most n^2 different sub-problems, but to compute each sub-problem (non-recursively) it takes $\Omega(\frac{2^n}{n^2})$ time
3. `compute_cost(1, n)` computes only at most n^2 different sub−problems, but each sub-problem multiple times.

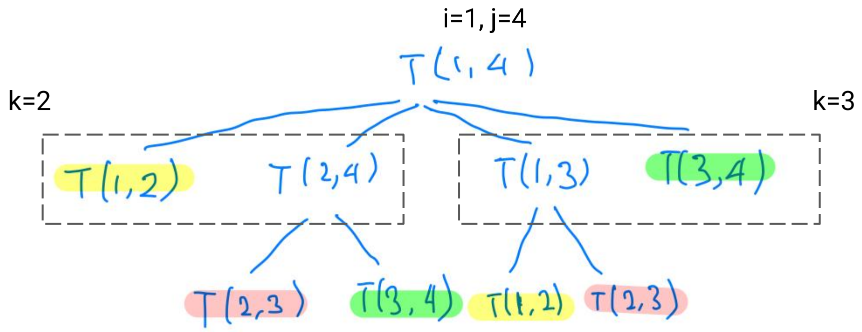
Ans: 3. `compute_cost(1, n)` computes only at most n^2 different sub−problems, but each sub-problem multiple times.

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

$T(1, n)$: compute_cost$(1,n)$



There are 6 distinct subproblems $= \binom{4}{2}$

In general : $\binom{n}{2}$ subproblems , all the pairs $T(i,j)$ where $i<j$

Recall:
$nC2 = n(n-1)/2 < n^2$

Question 4: Bottom-up Triangulation

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

Q4

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases

      for (i < k < j):

  return T[1, n]
```

Fill the blocks so the following are true

1. Algorithm finds value of $c(i, j)$
2. This algorithm runs in $O(n^3)$ time
3. Computes only at most n^2 different sub-problems, **each exactly once**

Bottom-up Dynamic Programming Strategy

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

Question 4 Intuition

1. Start from a small polygon (bottom-up)
2. Increase size of polygon over time
   a. Note that you will need all polygons smaller than it to be computed
   b. Try all the polygons of that size
      i. Guess all the triangles for that polygon

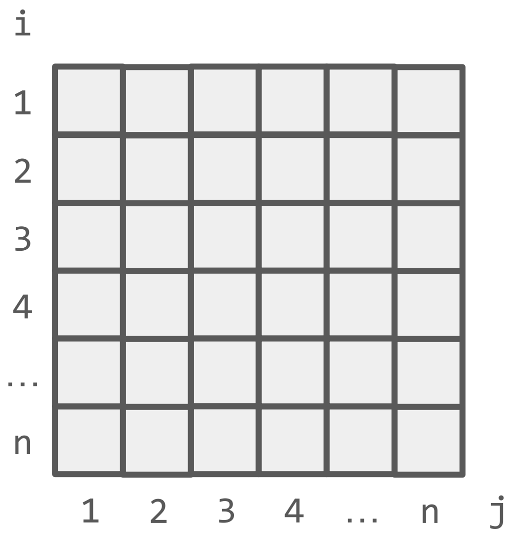
Q4

Visualising the table

```
def iter_opt_triangulation(1, n):
    for (i =1 to n-1):
      T[i, i+1] = 0 # base cases

        for (i < k < j):

    return T[1, n]
```



Q4

Note the base case and our goal

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
     T[i, i+1] = 0 # base cases

        for (i < k < j):

  return T[1, n]
```

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						

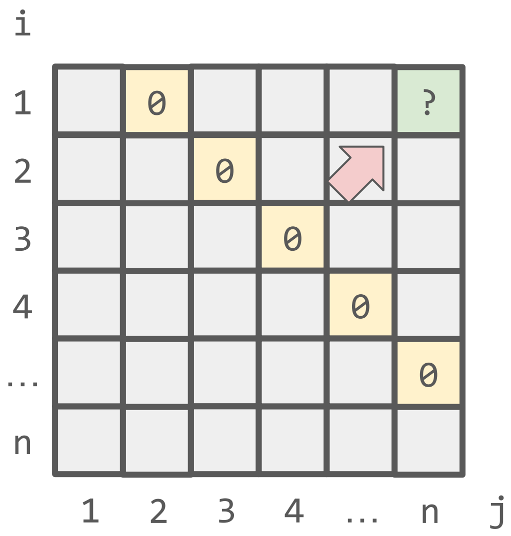
Q4

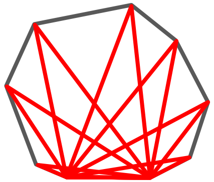
We need a way to compute it diagonally!

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
     T[i, i+1] = 0 # base cases

        for (i < k < j):

  return T[1, n]
```



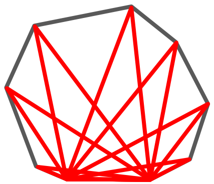


Q4

How to compute a cell?

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases
        for (i < k < j):
  return T[1, n]
```

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						



Q4

How to compute a cell?

Think about the recurrence relation!

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

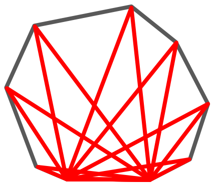
```
def iter_opt_triangulation(1, n):
    for (i =1 to n-1):
        T[i, i+1] = 0 # base cases

            for (i < k < j):

    return T[1, n]
```

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						

$T[1, 3] = T[1,2] + T[2, 3] + w(1,2,3)$



Q4

Let's say we are done with the next diagonal. What about T[1, 4]?

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$\min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

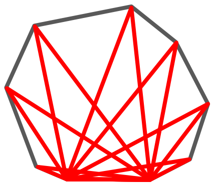
```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
     T[i, i+1] = 0 # base cases

        for (i < k < j):

  return T[1, n]
```

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						

T[1,4] = min(
T[1, 2] + T[2, 4] + w(1, 2, 4),
T[1, 3] + T[3, 4] + w(1, 3, 4))



Q4

This is what changing all the k in-between is for!

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

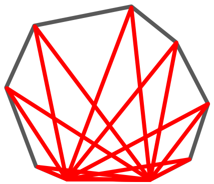
$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases
```

```
        for (i < k < j):
          curr = T[i, k] + T[k, j] + w(i, k, j)
          # take the better result
          T[i, j] = min(T[i, j], curr)
  return T[1, n]
```

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						

$T[1,4] = min($
$T[1, 2] + T[2, 4] + w(1, 2, 4),$
$T[1, 3] + T[3, 4] + w(1, 3, 4))$



Q4

Another example of how the recursion looks like (focus on the cells. Sorry my table is quite small!). T[1, n]

```
def iter_opt_triangulation(1, n):
    for (i =1 to n-1):
        T[i, i+1] = 0 # base cases

            for (i < k < j):
                curr = T[i, k] + T[k, j] + w(i, k, j)
                # take the better result
                T[i, j] = min(T[i, j], curr)
    return T[1, n]
```

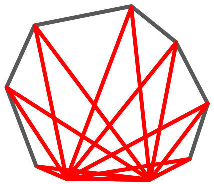
1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

i \ j	1	2	3	4	...	n
1		0				?
2			0			
3				0		
4					0	
...						0
n						



Q4

T[1,2] + T[2,n] + w(1,2,n)

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

if $j == i + 1$: $c(i, j) = 0$

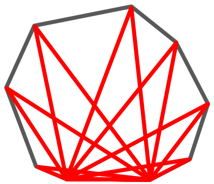
if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

```
def iter_opt_triangulation(1, n):
    for (i =1 to n-1):
        T[i, i+1] = 0 # base cases

            for (i < k < j):
                curr = T[i, k] + T[k, j] + w(i, k, j)
                # take the better result
                T[i, j] = min(T[i, j], curr)
    return T[1, n]
```

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						



Q4

$$T[1,3] + T[3,n] + w(1,3,n)$$

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
     T[i, i+1] = 0 # base cases

        for (i < k < j):
          curr = T[i, k] + T[k, j] + w(i, k, j)
          # take the better result
          T[i, j] = min(T[i, j], curr)
  return T[1, n]
```

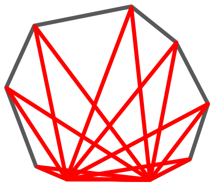
1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						



Q4

T[1,4] + T[4,n] + w(1,4,n)

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

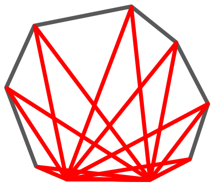
if $j == i + 1$: $c(i, j) = 0$

if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
     T[i, i+1] = 0 # base cases
       for (i < k < j):
         curr = T[i, k] + T[k, j] + w(i, k, j)
         # take the better result
         T[i, j] = min(T[i, j], curr)
  return T[1, n]
```

i \ j	1	2	3	4	...	n
1		0				?
2			0			
3				0		
4					0	
...						0
n						



Q4

`T[1,...] + T[...,n] + w(1,4,n)`

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

if $j == i + 1$: $c(i, j) = 0$

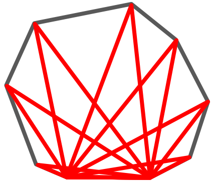
if $j > i + 1$:

$min(c(i, k) + w(i, k, j) + c(k, j))$
for all $i < k < j$

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
     T[i, i+1] = 0 # base cases
```

```
        for (i < k < j):
            curr = T[i, k] + T[k, j] + w(i, k, j)
            # take the better result
            T[i, j] = min(T[i, j], curr)
  return T[1, n]
```

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						



Q4

T[1,...] + T[...,n] + w(1,4,n)

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

```
def iter_opt_triangulation(1, n):
    for (i =1 to n-1):
        T[i, i+1] = 0 # base cases

        for (i < k < j):
            curr = T[i, k] + T[k, j] + w(i, k, j)
            # take the better result
            T[i, j] = min(T[i, j], curr)
    return T[1, n]
```

At this point you know how to compute a cell! The hard part is over

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						

Q4

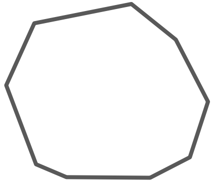
In bottom-up DP, **figure out in what order should I fill up the table?**

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
     T[i, i+1] = 0 # base cases

        for (i < k < j):
          curr = T[i, k] + T[k, j] + w(i, k, j)
          # take the better result
          T[i, j] = min(T[i, j], curr)
  return T[1, n]
```

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						



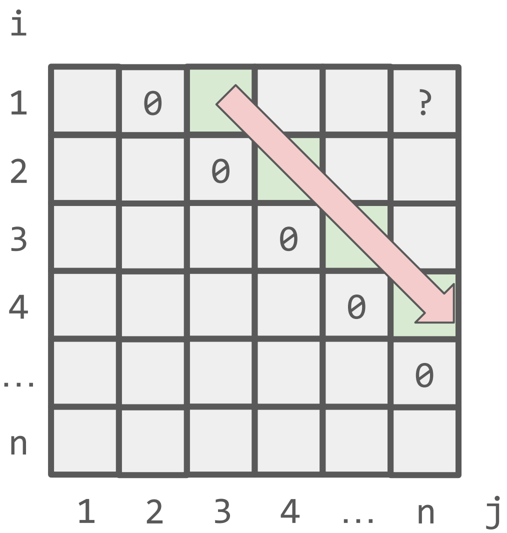
Q4

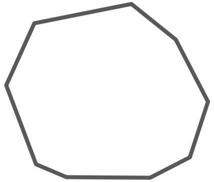
Next qn: How to traverse the diagonal?
Recall: traverse layer by layer. Want to traverse one layer first

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases

      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```





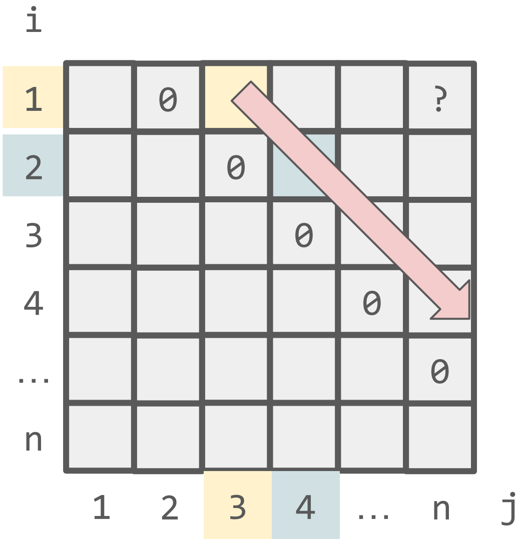
Q4

Notice that j - i is always different by a constant. Here it is 2

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

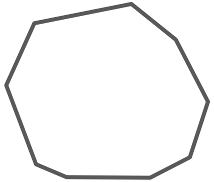
```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases

      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```



3 - 1 = 2

4 - 2 = 2



Q4

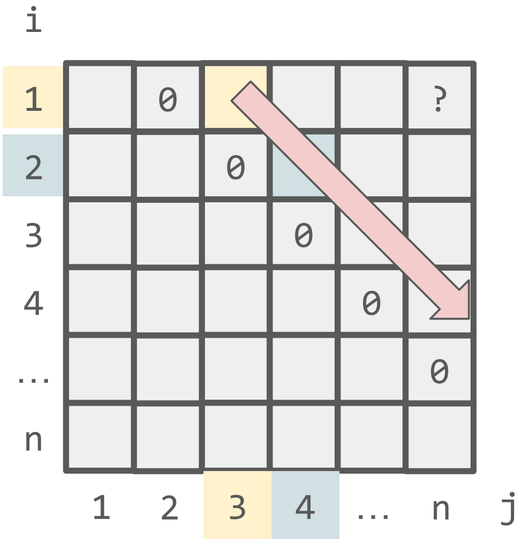
To traverse diagonal we try to increase i, and get the j index by adding that constant 2 to i

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases
```

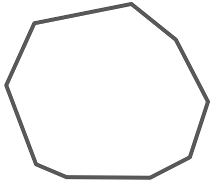
In our case, size = 2

```
    for (i = 1 to n-size):
      j = i + size
      T[i, j] = INF
      # try triangles
      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```



3 - 1 = 2

4 - 2 = 2



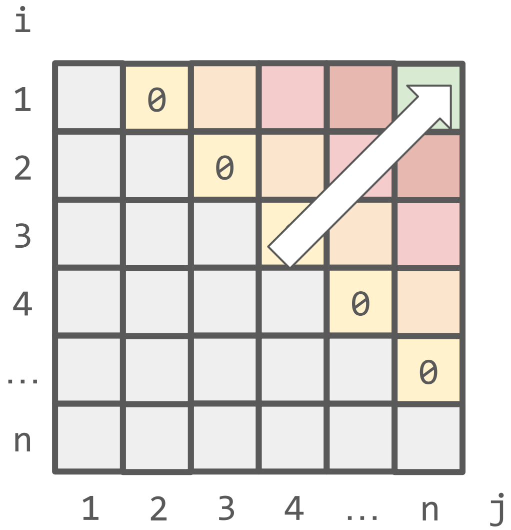
Q4

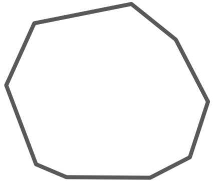
How to traverse the diagonal towards the upper right?

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases

    for (i = 1 to n-size):
      j = i + size
      T[i, j] = INF
      # try triangles
      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```





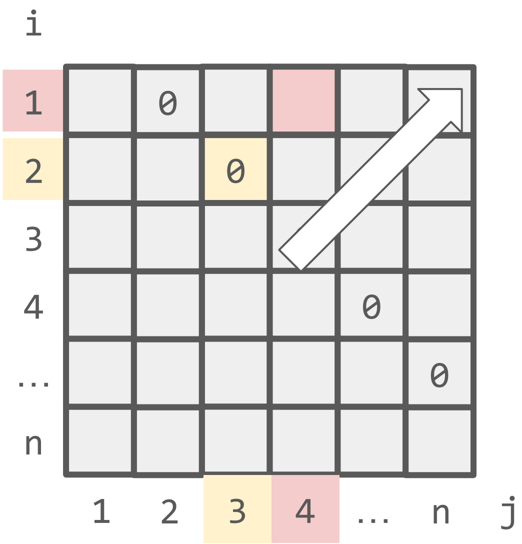
Q4

Difference between i and j increases!

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

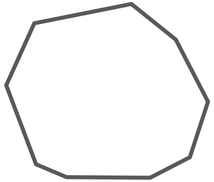
```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases

    for (i = 1 to n-size):
      j = i + size
      T[i, j] = INF
      # try triangles
      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```



3 - 2 = 1

4 - 1 = 3

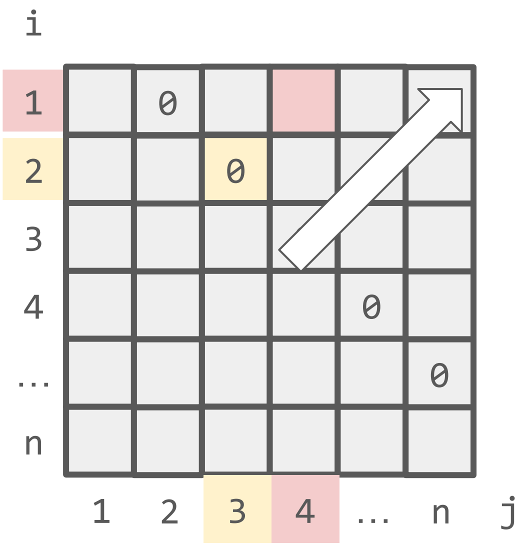


Q4

So we increase the difference along the way

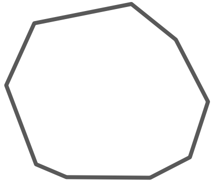
1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases
  # size=k => polygon with k+1 pts
  for (size = 2 to n-1):
    # n-1-size prevents overflow of i+size
    for (i = 1 to n-size):
      j = i + size
      T[i, j] = INF
      # try triangles
      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```



3 - 2 = 1

4 - 1 = 3

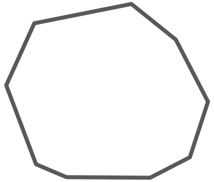


Q4

Intuition (if we don’t try to view it based on the table)

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases
  # size=k => polygon with k+1 pts
  for (size = 2 to n-1):
    # n-1-size prevents overflow of i+size
    for (i = 1 to n-size):
      j = i + size
      T[i, j] = INF
      # try triangles
      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```

1. Start from a small polygon

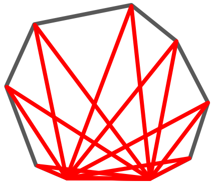


Q4

Intuition (if we don’t try to view it based on the table)

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases
  # size=k => polygon with k+1 pts
  for (size = 2 to n-1):
    # n-1-size prevents overflow of i+size
    for (i = 1 to n-size):
      j = i + size
      T[i, j] = INF
      # try triangles
      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```

1. Start from a small polygon
2. Increase size of polygon over time
   a. Note that you will need all polygons smaller than it to be computed
   b. Try all the polygons of that size

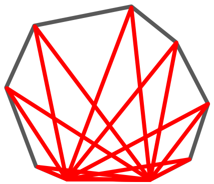


Q4

Intuition (if we don't try to view it based on the table)

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases
  # size=k => polygon with k+1 pts
  for (size = 2 to n-1):
    # n-1-size prevents overflow of i+size
    for (i = 1 to n-size):
      j = i + size
      T[i, j] = INF
      # try triangles
      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```

1. Start from a small polygon
2. Increase size of polygon over time
   a. Note that you will need all polygons smaller than it to be computed
   b. Try all the polygons of that size
      i. Guess all the triangles for that polygon

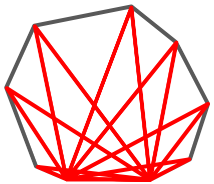


Q4

Time Complexity = Total time required to compute all cells

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases
  # size=k => polygon with k+1 pts
  for (size = 2 to n-1):
    # n-1-size prevents overflow of i+size
    for (i = 1 to n-size):
      j = i + size
      T[i, j] = INF
      # try triangles
      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						



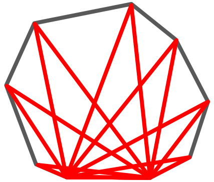
Q4

Time Complexity = Total time required to compute all cells

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases
  # size=k => polygon with k+1 pts
  for (size = 2 to n-1):
    # n-1-size prevents overflow of i+size
    for (i = 1 to n-size):
      j = i + size
      T[i, j] = INF
      # try triangles
      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```

i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						

1 cell = $O(n)$ [the range of i to j can be up to n elements]



Q4

Time Complexity = Total time required to compute all cells

```
def iter_opt_triangulation(1, n):
  for (i =1 to n-1):
    T[i, i+1] = 0 # base cases
  # size=k => polygon with k+1 pts
  for (size = 2 to n-1):
    # n-1-size prevents overflow of i+size
    for (i = 1 to n-size):
      j = i + size
      T[i, j] = INF
      # try triangles
      for (i < k < j):
        curr = T[i, k] + T[k, j] + w(i, k, j)
        # take the better result
        T[i, j] = min(T[i, j], curr)
  return T[1, n]
```

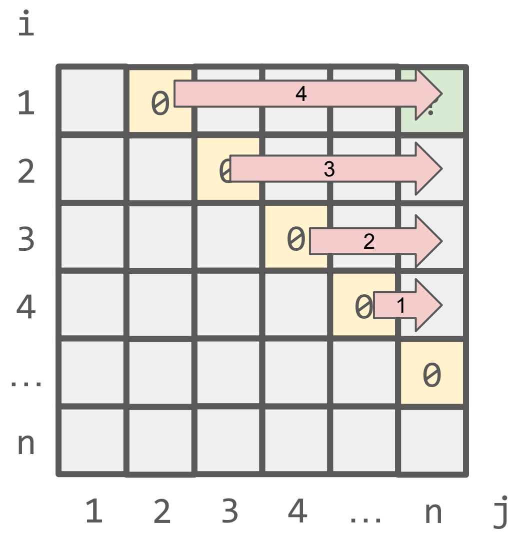
i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						

1 cell = $O(n)$ [the range of i to j can be up to n elements]

$O(n^2)$ cells = $O(n^2) \times O(n) = O(n^3)$

Q4

Alternative way to fill up the table



Q4

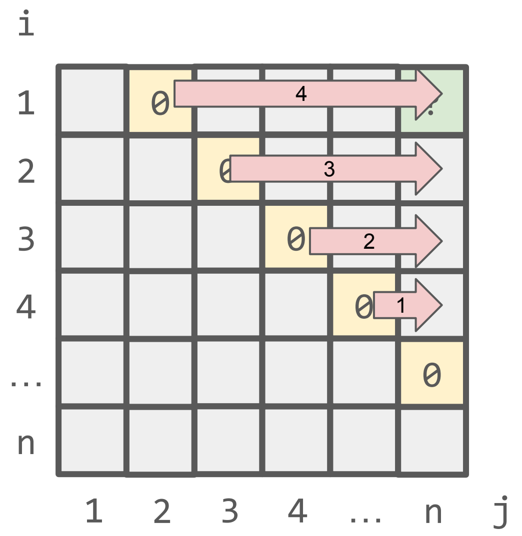
Alternative way to fill up the table

Remember, what's important is:

1. Given a cell, how can you compute them?
2. In what order should I fill them up to reach the target?
    a. There can be different orderings
    b. As long as when I reach a cell, I already have all the different subproblems needed, then it's fine.

Verify that this ordering is still okay!

This might be easier to implement too if you are not used to diagonals



Q4

You can also write a recursive version and store the results in a table!

```
def compute_cost(i, j):
    if (j == i + 1):
        return 0
    else:
        if (T[i, j] is not null):
            return T[i, j] # avoid recomputation
        cost = INF # dummy
        for (i < k < j):
            curr = compute_cost(i, k)
                   + w(i, k, j)
                   + compute_cost(k, j)
            # take the better result
            cost = min(cost, curr)
        T[i, j] = cost # save value
        return T[i, j]
```

Note: Pseudocode lacks initialisation of the table but it isn't hard to write

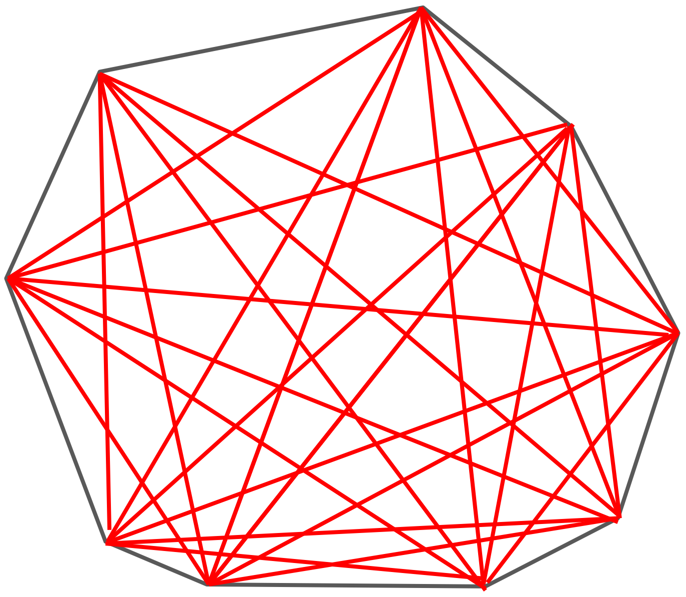
i \ j	1	2	3	4	…	n
1		0				?
2			0			
3				0		
4					0	
…						0
n						

Bottom-up vs Top-down

- There aren’t a lot of differences in terms of time complexity, usually
- But there could be some differences:

- Bottom-up
  - Implementation might have better performance due to low-level computer details
    - Caching performance
    - Top-down having to maintain stack frames for recursive calls
- Top-down
  - On rare cases, it might not need to compute as many subproblems as its bottom-up counterpart

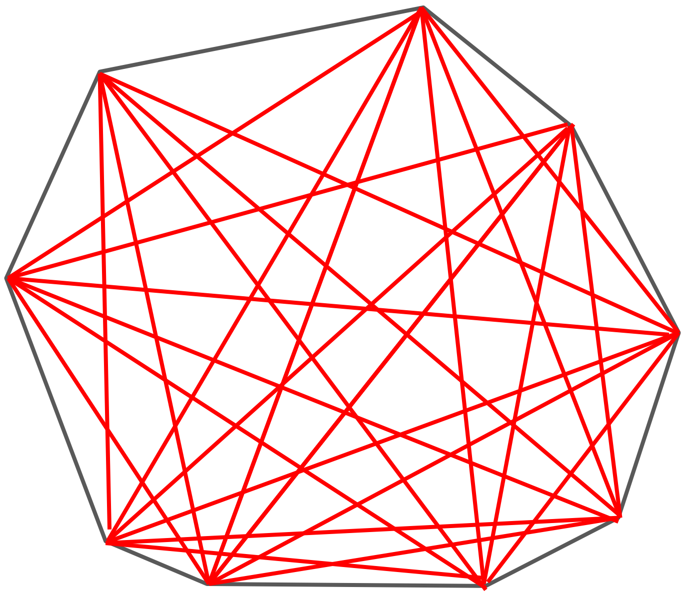
Time Complexity of this other strategy?



Strategy:

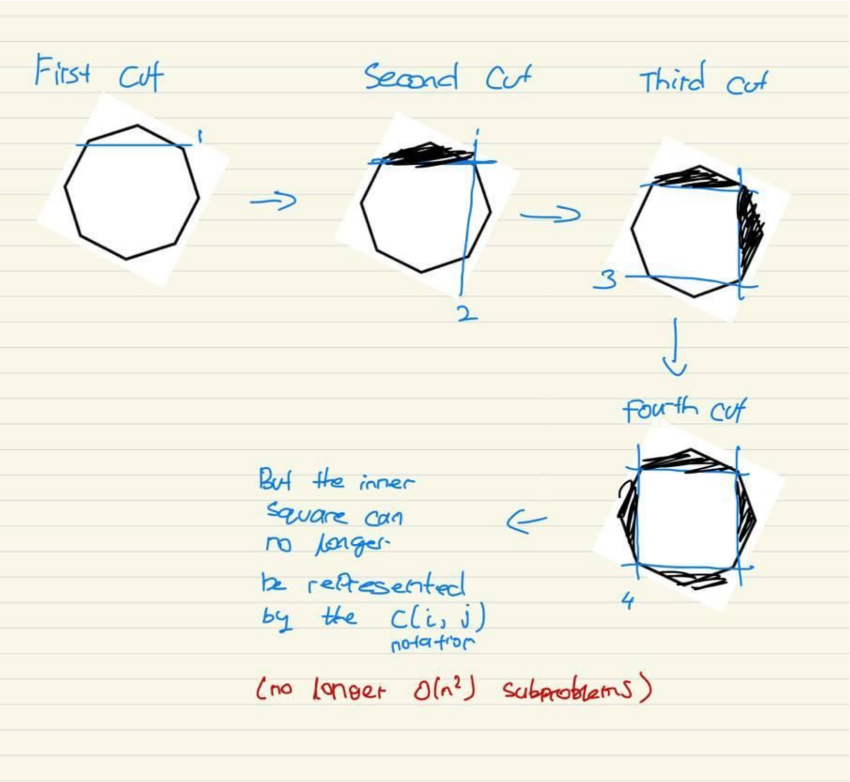
1. Try all the dividing lines!
2. Find cost of both polygons that got divided and sum
   a. Take the minimum over everything

Time Complexity of this other strategy?



Unfortunately, this other strategy takes exponential time for the reason that there are exponential subproblems

Time Complexity of this other strategy?

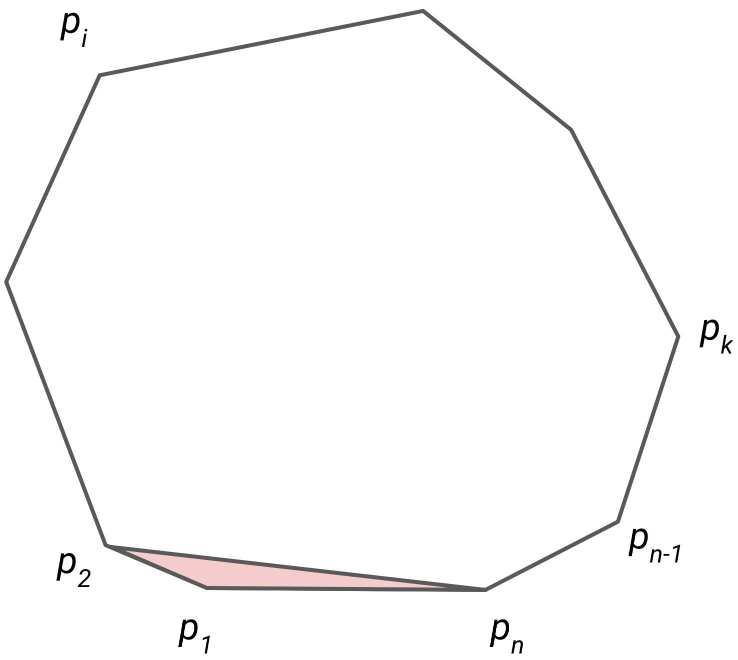


The reason is because cutting arbitrarily can generate subpolygons that can no longer be described using the c(i, j) notation (going clockwise from p_i to p_j)

Thus to describe an arbitrary subpolygon, it can only be described by taking the subset of points involved in the polygon. We know that given n points there are 2^n possible subsets of points.

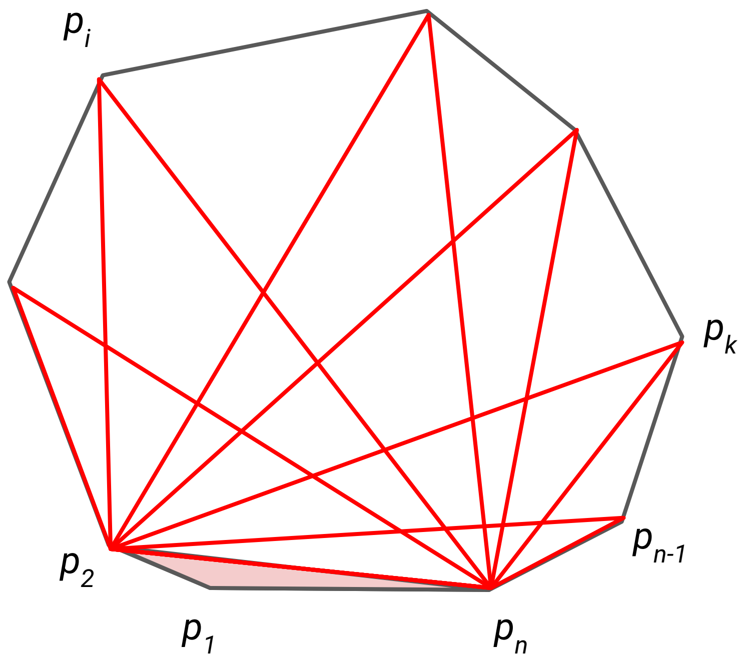
Therefore this approach even with DP will take $\Omega(2^n)$ time

Does the original strategy exhibit this problem?



The original strategy is careful in the placement of the triangle. Take for example the first choice of the triangle on the left

Does the original strategy exhibit this problem?



The original strategy is careful in the placement of the triangle. Take for example the first choice of the triangle on the left

In the c(2, n) subproblem, the next choice of triangle is “stacked” on top of the first triangle. It will avoid the scenario where it will create “holes” (like before) so that all the subproblems can be described just by using the start and end points

Summary - How to DP?

Brute Force, but *carefully*

1. Identify the subproblems
2. To solve the current subproblem, **assume** you have solved the other (smaller) subproblems (Tip: **DON'T unroll the recursion**)
3. Relate the smaller subproblem to the current subproblem
   a. Guess the relation!
   b. This might involve trying **all** subproblems!

- Your subproblem result might be re-used -- store it in a table!
- Time complexity: total time to compute **all subproblems**