CSE 417
Dynamic Programming (pt 4)
Sub-problems on Trees

UNIVERSITY of WASHINGTON

Reminders

> HW4 is due today

> HW5 will be posted shortly

Dynamic Programming Review

> Apply the steps...
1. Describe solution in terms of solution to *any* sub-problems
2. Determine all the sub-problems you'll need to apply this recursively
3. Solve every sub-problem (once only) in an appropriate order

> Key question:
1. Can you solve the problem by combining solutions from sub-problems?

> Count sub-problems to determine running time
– total is number of sub-problems times time per sub-problem

Review From Last Time: More General Sub-problems

even if we have to <u>guess</u> sub-problems (non-obvious cases), can still think about what new solutions are allowed in larger sub-problems vs smaller ones to find opt substructure

> **Previously:**
  - Find opt substructure by considering how the opt solution could use the last input.

> **Knapsack Problem**
  - sub-problems are 1 .. k and weight $V \le W$ — more general than original problem
  - $O(nW)$ algorithm

> **All-Pairs Shortest Paths (with Negative Weights)**
  - application of the basic technique, but simpler code with clever sub-problems
  - sub-problems are paths with intermediate nodes from 1 .. k

> **Single-Source Shortest Paths with Negative Weights**
  - sub-problems are shortest paths of length at most k

Review From Last Time: More General Sub-problems

algorithms are getting slower,
but in different ways...

more sub-problems to solve
but still fast when W is small

(re: shortest path & opt breakout trades...)
have to consider O(n) solutions to problem,
but still get a set that must include opt

- **Previously:**
  - Find opt substructure by considering how the opt solution could use the last input.
- **Knapsack Problem**
  - sub-problems are 1 .. k and weight $V \le W$ — more general than original problem
  - O(nW) algorithm
- **All-Pairs Shortest Paths (with Negative Weights)**
  - application of the basic technique, but simpler code with clever sub-problems
  - sub-problems are paths with intermediate nodes from 1 .. k
- **Single-Source Shortest Paths with Negative Weights**
  - sub-problems are shortest paths of length at most k

Outline for Today

> Optimal Binary Search Trees

> Matrix Chain Multiplication

> Optimal Polygon Triangulation

Optimal Binary Search Tree

- **Problem**: Given a set of elements x_1, ..., x_n and access frequencies f_1, ..., f_n, find the binary search tree storing x_1, ..., x_n whose total time to perform f_1 lookups of x_1, ..., f_n lookups of x_n is smallest.

- The time to access a node at depth d is O(d)
  - to simplify notation, we'll assume the hidden constant is C = 1
- The time to perform f lookups of data at depth d is fd

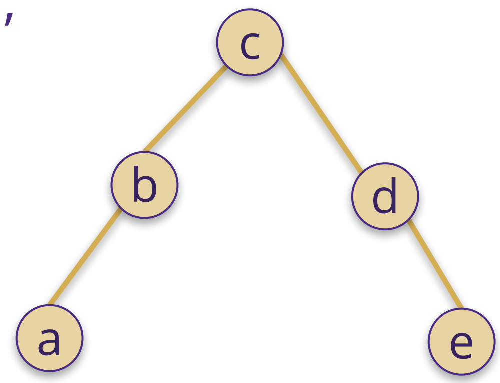
- Let d_i be the depth at which x_i is stored.
  Then the total time is $f_1 d_1 + \ldots + f_n d_n$

Optimal Binary Search Tree Example

- Balanced binary search tree ensures $d_i \leq \lg n$,
  so $f_1 d_1 + \ldots + f_n d_n \leq (f_1 + \ldots + f_n) \lg n$
- BUT that could be far from optimal

- Let the elements be a, b, c, d, e
  with access frequencies 1, 1, 1, 1, 10^{100}

- Balanced tree access time $\approx 3 \cdot 10^{100}$
- Any tree with e at root $\approx 1 \cdot 10^{100}$

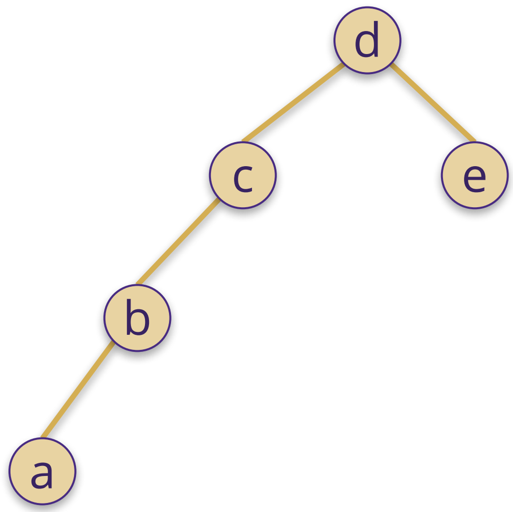


Optimal Binary Search Tree Example 2

- Let the elements be a, b, c, d, e
  with access frequencies 1, 2, 3, 4, 5

- The tree on the right has access time of
  $1 \cdot 4 + 2 \cdot 3 + 3 \cdot 2 + 4 \cdot 1 + 5 \cdot 2 = 30$

- Greedy would put e at the root,
  and get access time of
  $1 \cdot 5 + 2 \cdot 4 + 3 \cdot 3 + 4 \cdot 2 + 5 \cdot 1 = 35$



Optimal Binary Search Tree

- Brute force: the number of possible trees is roughly 4^n...
- Apply dynamic programming...
  - write the solution in terms of solutions to sub-problems
- In this case, considering how the *last element* is used in the optimal solution will not lead anywhere...
  - (no obvious relationship to trees using only 1 .. n-1)

Optimal Binary Search Tree

- Apply dynamic programming...
  - write the solution in terms of solutions to sub-problems

- Still works to think about what the optimal solution looks like...
- Some element x_i must be at the root of the tree
  - then left subtree has $x_1, \ldots, x_{i-1}$
  - and right subtree has $x_{i+1}, \ldots, x_n$

- Claim: both subtrees must be themselves optimal over those subsets of the elements

Optimal Binary Search Tree

- Some element x_i must be at the root of the tree
  - then left subtree has $x_1, \ldots, x_{i-1}$ and right subtree has $x_{i+1}, \ldots, x_n$
- Claim: both subtrees must be themselves optimal over those subsets of the elements
- Let depths be $d_1, \ldots, d_{i-1}$ in left subtree (without root)
- Total access time is $f_1 d_1 + \ldots + f_{i-1} d_{i-1}$

Optimal Binary Search Tree

> Claim: both subtrees must be themselves optimal over those subsets of the elements

> Let depths be $d_1, \ldots, d_{i-1}$ in left subtree (without root)

> Total access time is $f_1 d_1 + \ldots + f_{i-1} d_{i-1}$

opt tree must be opt on sub-problem for left subtree

> With root, time is $f_1(d_1+1) + \ldots + f_{i-1}(d_{i-1}+1)$
$= f_1 d_1 + \ldots + f_{i-1} d_{i-1} + f_1 + \ldots + f_{i-1}$

constant independent of depths

Optimal Binary Search Tree

> Apply dynamic programming...
  – write the solution in terms of solutions to sub-problems

> Some element x_i must be at the root of the tree
  – the left subtree must be the opt search tree over $x_1, \ldots, x_{i-1}$
  – the right subtree must be the opt search tree over $x_{i+1}, \ldots, x_n$

> Find the correct root by trying them all
  – opt value = **min** (opt value on $x_1, \ldots, x_{i-1}$) + $f_1 + \ldots + f_{i-1}$ +
    (opt value on $x_{i+1}, \ldots, x_n$) + $f_{i+1} + \ldots + f_n$
    + f_i **over** $i = 1 .. n$

Optimal Binary Search Tree

- Apply dynamic programming…
  - write the solution in terms of solutions to sub-problems
- Some element x_i must be at the root of the tree
  - the left subtree must be the opt search tree over $x_1, \ldots, x_{i-1}$
  - the right subtree must be the opt search tree over $x_{i+1}, \ldots, x_n$
- Find the correct root by trying them all
  - let $F = f_1 + \ldots + f_n$
  - opt value = **min** (opt value on $x_1, \ldots, x_{i-1}$) + F
    (opt value on $x_{i+1}, \ldots, x_n$) **over** i = 1 .. n

Optimal Binary Search Tree

> Apply dynamic programming...
  - write the solution in terms of solutions to sub-problems

> Some element x_i must be at the root of the tree
  - the left subtree must be the opt search tree over $x_1, \ldots, x_{i-1}$
  - the right subtree must be the opt search tree over $x_{i+1}, \ldots, x_n$

> Find the correct root by trying them all
  - let $F = f_1 + \ldots + f_n$
  - opt value = F + **min** (opt value on $x_1, \ldots, x_{i-1}$) + (opt value on $x_{i+1}, \ldots, x_n$) **over** i = 1 .. n

Optimal Binary Search Tree

> Apply dynamic programming…

1. Can find opt on $x_1, \ldots, x_n$ from opt on prefixes $x_1, .., x_{i-1}$ and suffixes $x_{i+1}, \ldots, x_n$

2. To apply this recursively, we need opt on every range $x_i, \ldots, x_j$
   > (suffix for root at x_i > prefix for root at x_j > sub-problem on $x_{i+1}, \ldots, x_{j-1}$)

3. Solve sub-problems starting from ranges of size 1
   > only tree on just x_i is one node x_i: opt value = f_i
   > for $x_i, \ldots, x_j$, try every root…
   opt value = F + **min** (opt value on $x_i, \ldots, x_{k-1}$) +
   (opt value on $x_{k+1}, \ldots, x_j$) **over** k = i .. j

Optimal Binary Search Tree

- Apply dynamic programming...
  1. Can find opt on $x_1, \ldots, x_n$ from opt on prefixes $x_1, .., x_{i-1}$ and suffixes $x_{i+1}, \ldots, x_n$
  2. To apply this recursively, we need opt on every range $x_i, \ldots, x_j$
  3. Solve sub-problems starting from ranges of size 1. Then use formula:
     - opt value = F + **min** (opt value on $x_i, \ldots, x_{k-1}$) + (opt value on $x_{k+1}, \ldots, x_j$) **over** k = i .. j
- $O(n^2)$ sub-problems
- Total running time is $O(n^3)$
  - probably usable for n in the thousands

Optimal Binary Search Tree

> Can be implemented in a spreadsheet as well…
  - though it would get difficult for more than n = 30 or so
  - (in example, "Freq" sheet stores sum of frequencies for each range i .. j)

	A	B	C	D	E	F
1		**a**	**b**	**c**	**d**	**e**
2	**a**	1	4	10	18	30
3	**b**		2	7	15	26
4	**c**			3	10	20
5	**d**				4	13
6	**e**					5

fx =MIN(F3,B2+F4,C2+F5,D2+F6,E2)+Freq!F2

Optimal Binary Search Tree

> Can be implemented in a spreadsheet as well…
  – though it would get difficult for more than n = 30 or so
  – (in example, “Freq” sheet stores sum of frequencies for each range i .. j)

	A	B	C	D	E	F
1		a	b	c	d	e
2	a	1	4	10	18	F2
3	b		2	7	15	26
4	c			3	10	20
5	d				4	13
6	e					5

fx =MIN(F3,B2+F4,C2+F5,D2+F6,E2)+Freq!F2

Foreword

> Problems that are hard on graphs are often easy on trees...
  – tree structure works nicely within dynamic programming framework

> Will see that one way to solve the hard problems on graphs: approximate those graphs with trees

Outline for Today

- Optimal Binary Search Trees
- Matrix Chain Multiplication ⇐
- Optimal Polygon Triangulation

Matrix Chain Multiplication

- **Problem**: Given matrix dimensions d_0 x d_1, d_1 x d_2, ..., d_{n-1} x d_n, find the order in which to multiply them all together in min time.
  - can only multiply m_1 x n_1 by m_2 x n_2 if $n_1 = m_2$
    - that is why the second matrix has dimensions d_1 x d_2 above
  - time to multiply two such matrices is m_1 n_1 m_2
  - result is a matrix with dimensions m_1 x n_2
    - result of multiplying d_{i-1} x d_i, ..., d_{j-1} x d_j is matrix with dimensions d_{i-1} x d_j

- Example: matrices A B C
  - can be multiplied as (A B) C or A (B C)
  - result will be the same, but time could be different

Matrix Chain Multiplication

- Example: matrices A B C
- (A B) C
  - A B in time 10 · 100 · 1 = 1,000
  - (A B) C in time 10 · 1 · 10 = 100
  - total time is 1,100
- A (B C)
  - B C in time 100 · 1 · 10 = 1,000
  - A (B C) in time 10 · 100 · 10 = 10,000
  - total time is 11,000

	rows	cols
A	10	100
B	100	1
C	1	10

Matrix Chain Multiplication

> No brute force: exponentially many orderings

> Apply dynamic programming...
  - write the solution in terms of solutions to sub-problems
  - think about what the optimal solution might look like...

> **Q**: What is the last multiplication performed in the opt solution

> **A**: Must be $(A_1 \ldots A_{i-1})\ (A_i \ldots A_n)$ for some i
  - matrices can only be multiplied by those next to them
  - each multiplication merges two groups together into one
  - last merges the final two groups of adjacent matrices

Matrix Chain Multiplication

> Apply dynamic programming...
  - write the solution in terms of solutions to sub-problems

> **Q**: What is the last multiplication performed in the opt solution

> **A**: Must be $(A_1 \ldots A_i)(A_{i+1} \ldots A_n)$ for some i ← min over each choice of i

> Opt solution must multiply each of $A_1 \ldots A_i$ and $A_{i+1} \ldots A_n$ optimally
  - total time is time to multiply each group plus $d_0\, d_i\, d_n$
  - any way of multiplying $A_1 \ldots A_i$ is allowed, so the minimum total time is achieved by taking the best one
    - for each choice of i, the term $d_0\, d_i\, d_n$ is a fixed constant

Matrix Chain Multiplication

> Apply dynamic programming...

1. Can find opt on $A_1, \ldots, A_n$ from opt on prefixes $A_1, .., A_i$ and suffixes $A_{i+1}, \ldots, A_n$

2. To apply this recursively, we need opt on every range $A_i, \ldots, A_j$
   > (same as before: prefix of a suffix is an arbitrary range)

3. Solve sub-problems starting from ranges of size 1
   > multiply A_1 by itself in 0 time (already have it)

   > for $A_i, \ldots, A_j$, try every splitting point...
   opt value = **min** (opt value on $A_i, \ldots, A_k$) + $d_{i-1}\, d_k\, d_j$ +
   (opt value on $A_{k+1}, \ldots, A_j$) **over** k = i .. j-1

Matrix Chain Multiplication

> Apply dynamic programming...

1. Can find opt on $A_1, \ldots, A_n$ from opt on prefixes $A_1, .., A_i$ and suffixes $A_{i+1}, \ldots, A_n$
2. To apply this recursively, we need opt on every range $A_i, \ldots, A_j$
3. Solve sub-problems starting from ranges of size 1. Then use formula:
   > opt value = **min** (opt value on $A_i, \ldots, A_k$) + (opt value on $A_{k+1}, \ldots, A_j$) + $d_{i-1}\, d_k\, d_j$ **over** k = i .. j-1

> $O(n^2)$ sub-problems

> Total running time is $O(n^3)$
  – probably usable for n in the thousands

Matrix Chain Multiplication

> This looks very similar to previous problem...

1. Compute opt on every range $A_i, \ldots, A_j$
2. Solve sub-problems starting from ranges of size 1. Then use formula:
   > opt value = **min** (opt value on $A_i, \ldots, A_k$) + (opt value on $A_{k+1}, \ldots, A_j$) + $d_{i-1}\, d_k\, d_j$ **over** $k = i \,..\, j-1$

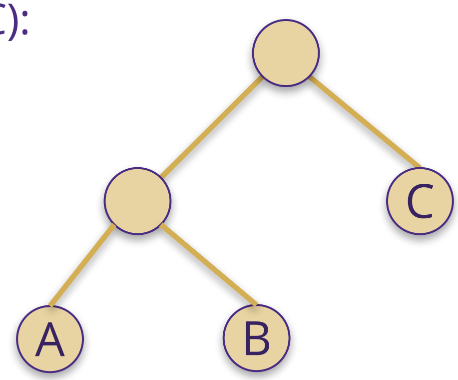
vs

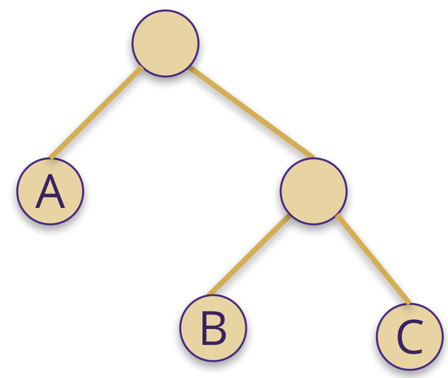
1. Compute opt on every range $x_i, \ldots, x_j$
2. Solve sub-problems starting from ranges of size 1, then use formula:
   > opt value = **min** (opt value on $x_i, \ldots, x_{k-1}$) + (opt value on $x_{k+1}, \ldots, x_j$) + F **over** $k = i \,..\, j$

> This is not an accident...

Matrix Chain Multiplication

- > Orderings of multiplications are trees...
  - – they are "parse trees" of the expression
  - – e.g., for (A B) C versus A (B C):





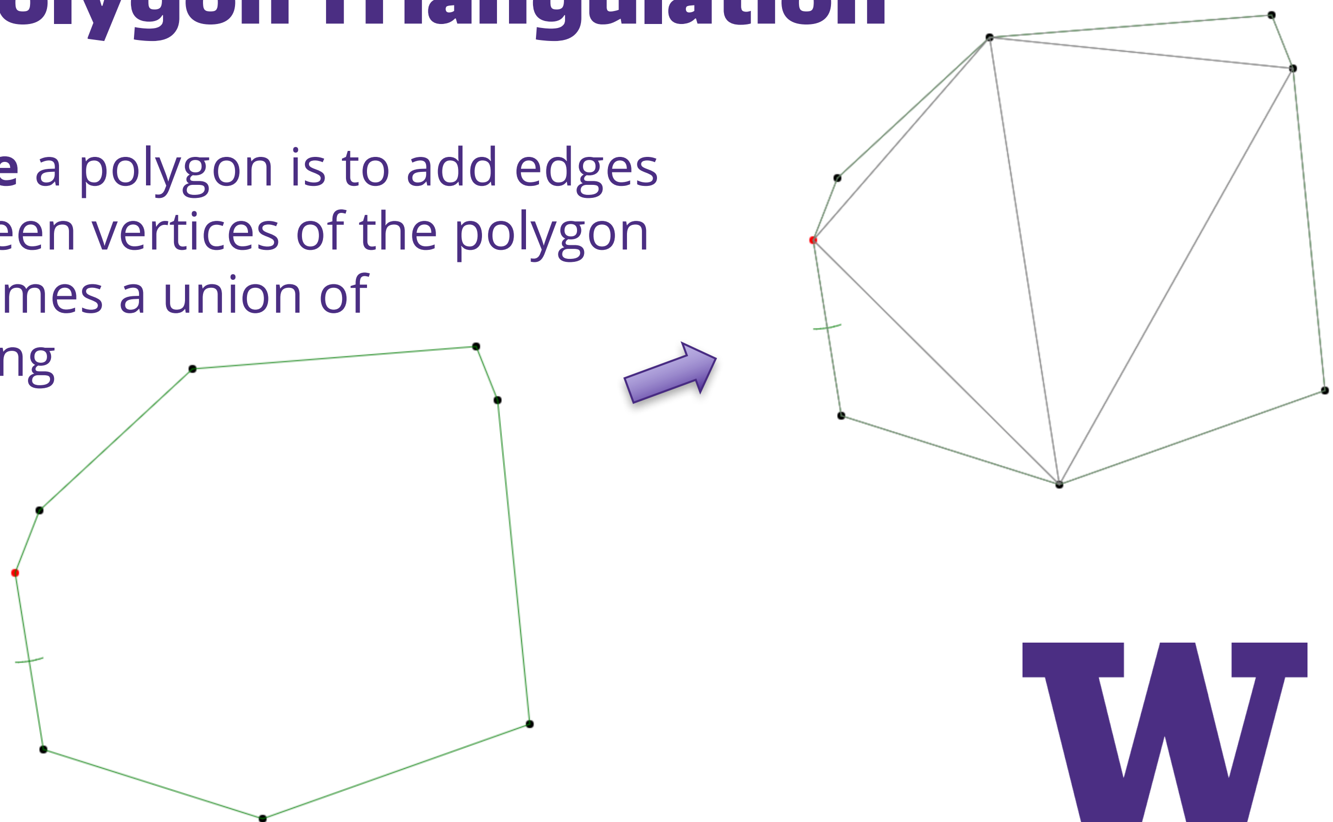
- > These are *essentially* the same problem.
  - – only notable difference is matrices only appearing in leaves

Outline for Today

> Optimal Binary Search Trees
> Matrix Chain Multiplication
> Optimal Polygon Triangulation

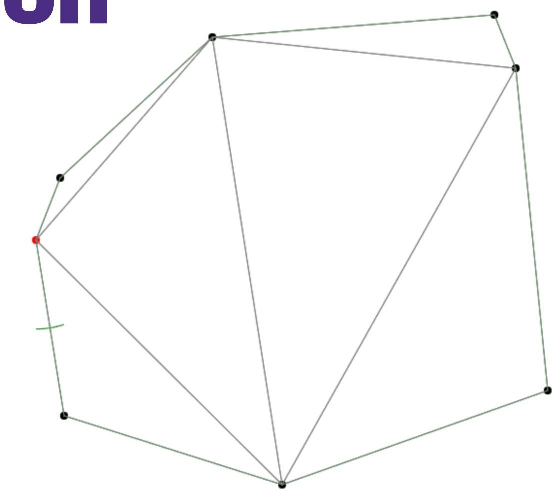
Optimal Polygon Triangulation

> To **triangulate** a polygon is to add edges (chords) between vertices of the polygon so that it becomes a union of non-overlapping triangles.
  – allowed to touch only on edges



Optimal Polygon Triangulation

> **Problem**: Find the triangulation of a given (convex) polygon that optimizes some quality metric over the choice of triangles.



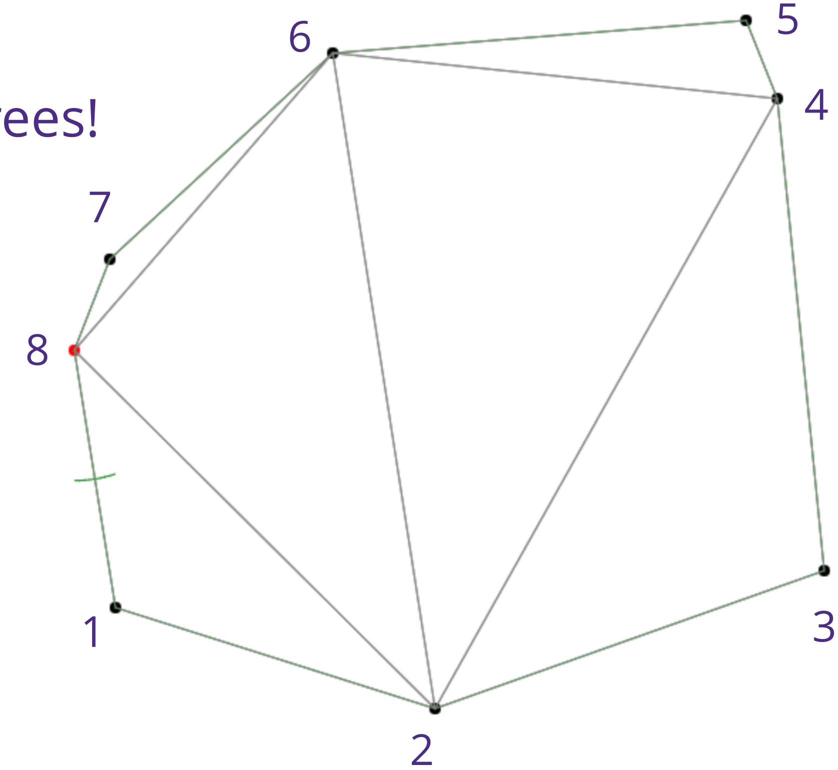
> Example metrics:
  - sum of the side lengths (minimize)
  - area divided by the sum of squared side lengths (minimize)
    > prefers triangles that are "more equilateral"

Optimal Polygon Triangulation

- Applications:
  - graphics
    - 3D hardware wants triangles
    - poorly shaped triangles can result in visual artifacts
  - finite element analysis (engineering & physics)
    - reduce complicated shapes to simple ones: triangles
    - often want triangles that are close to equilateral

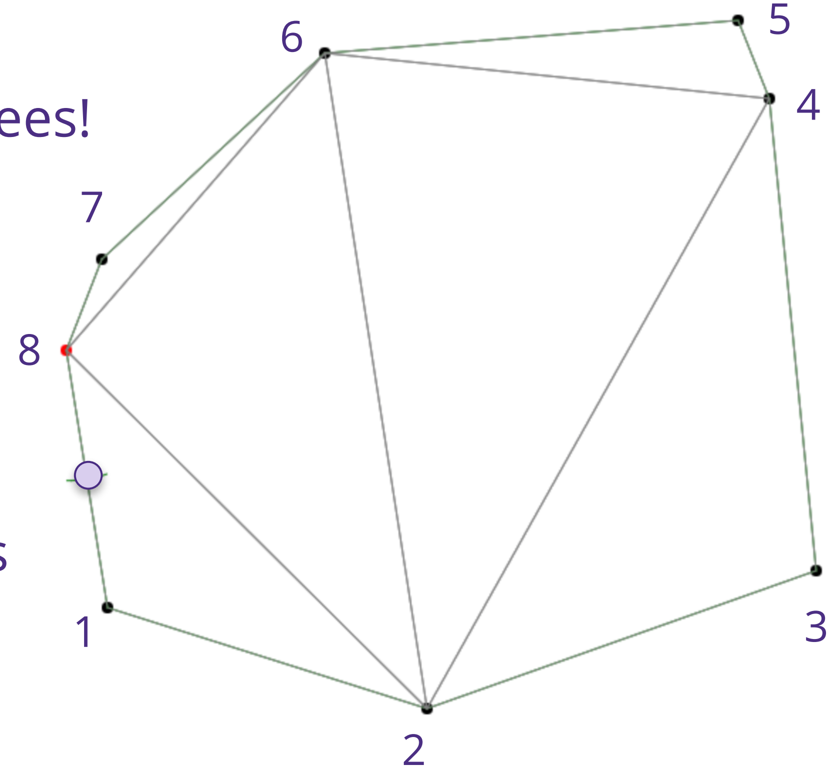
Optimal Polygon Triangulation

- Triangulations are trees!
  - label vertices 1 .. n
  - picture:
    - red is n
    - marked edge to 1



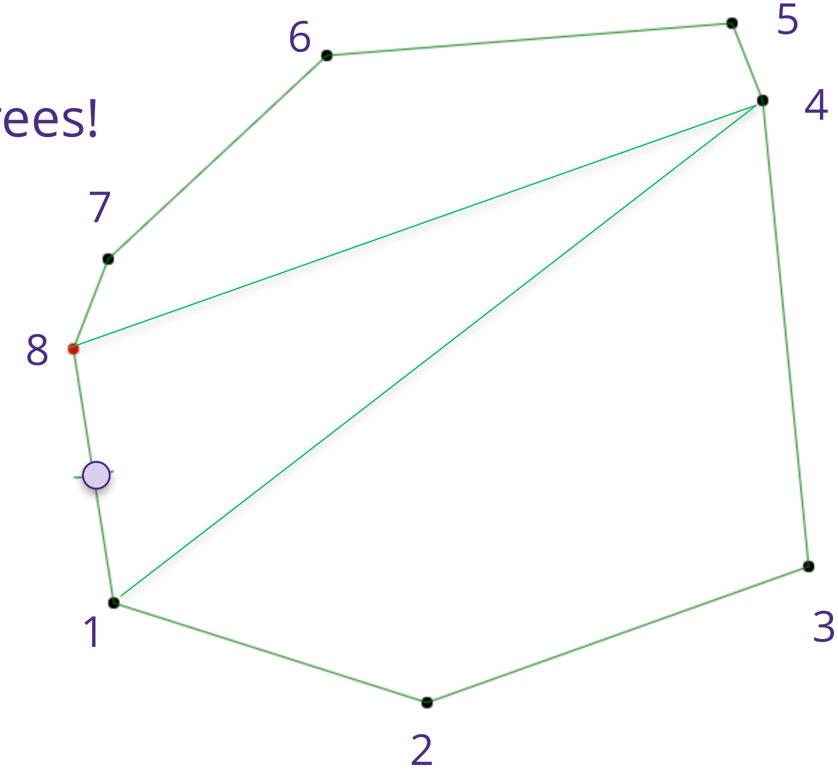
Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- Edge (1,n) must be part of a triangle, so 1 and n are both connected by chords to some node i
  - here, i = 2



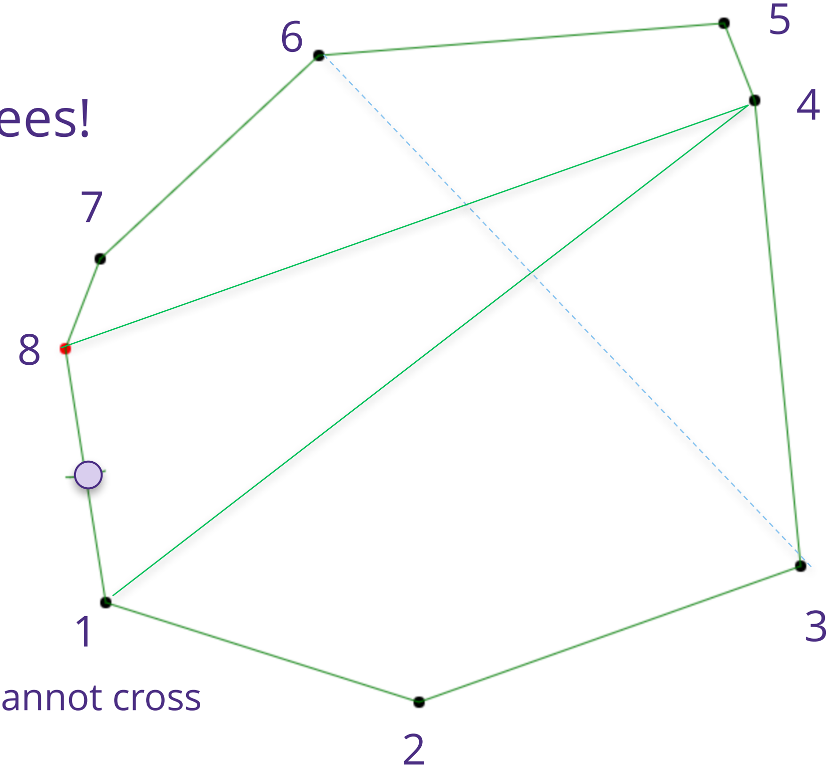
Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- 1 and n are both connected to some i
- (1,i,n) triangle cuts polygon in 1–2 pieces



Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- 1 and n are both connected to some i
- (1,i,n) triangle cuts polygon in 1–2 pieces
  - must triangulate *separately* since chords cannot cross

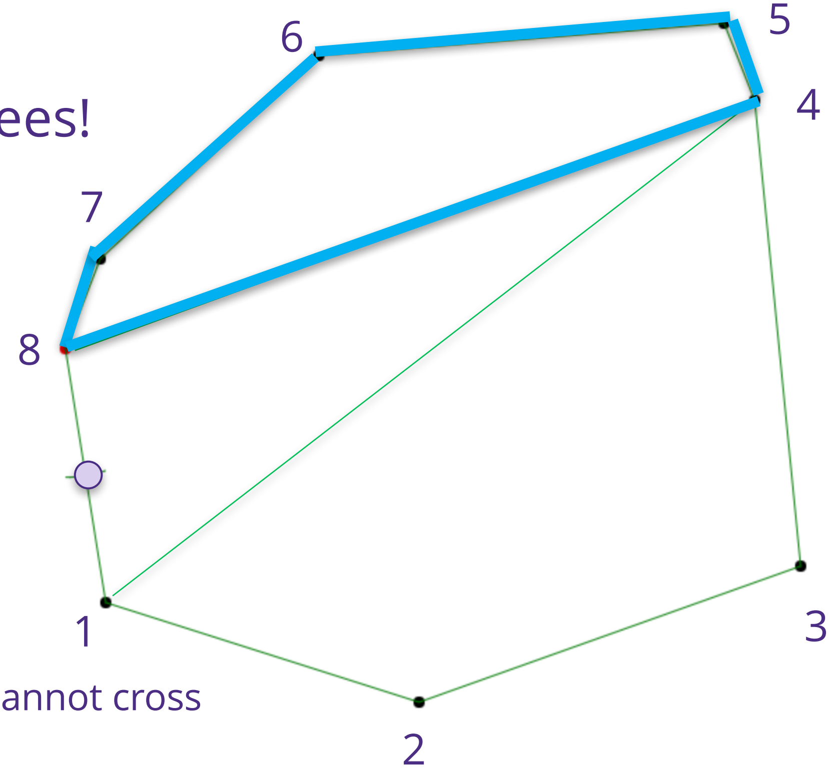


Optimal Polygon Triangulation

> Triangulations are trees!
  – leaves are edges
  – root is edge (1,n)

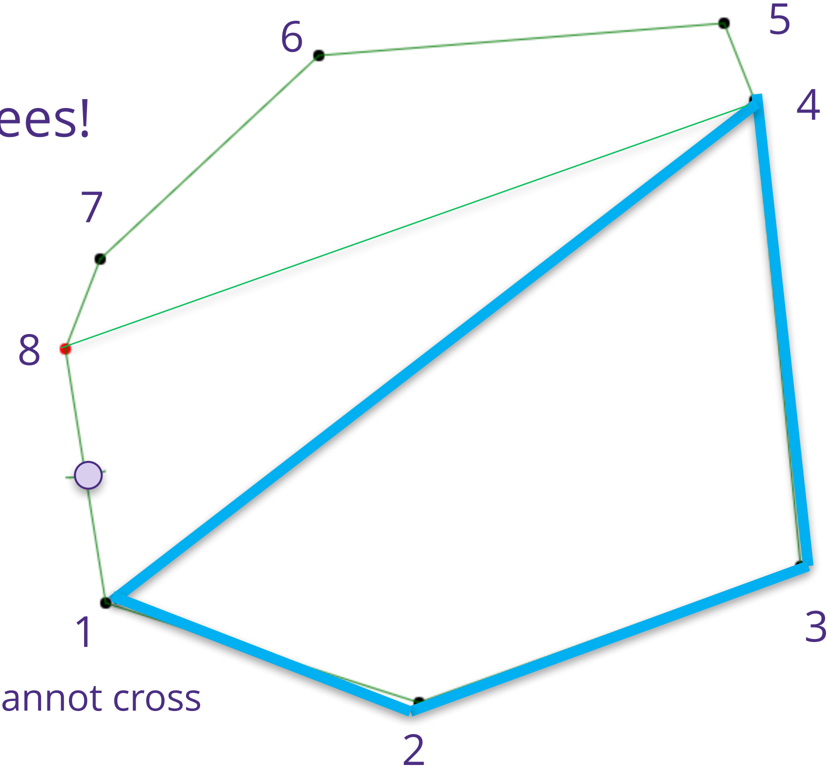
> 1 and n are both connected to some i

> (1,i,n) triangle cuts polygon in 1–2 pieces
  – must triangulate *separately* since chords cannot cross



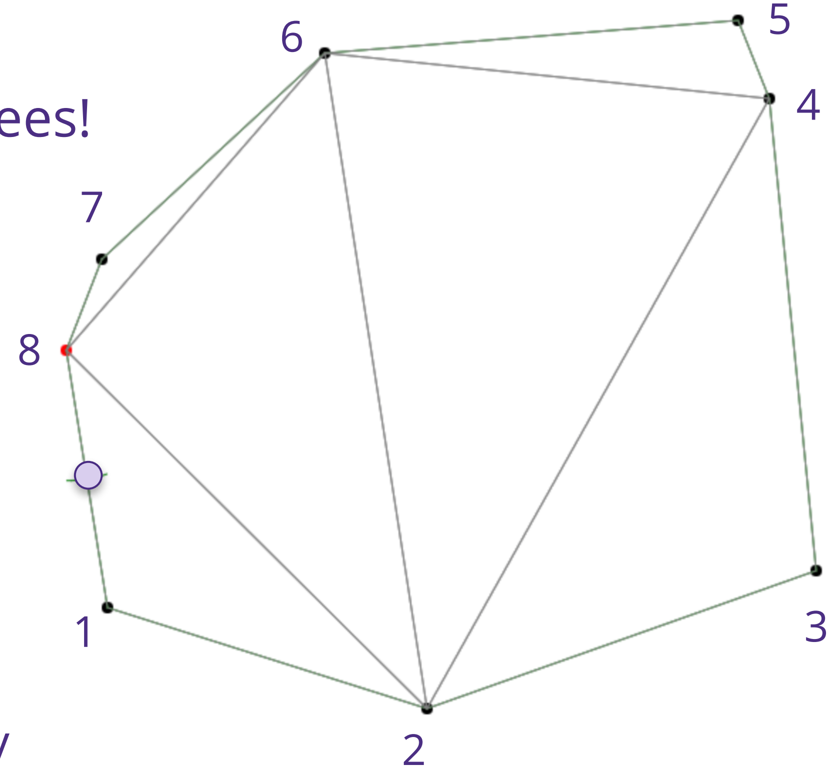
Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- 1 and n are both connected to some i
- (1,i,n) triangle cuts polygon in 1–2 pieces
  - must triangulate *separately* since chords cannot cross



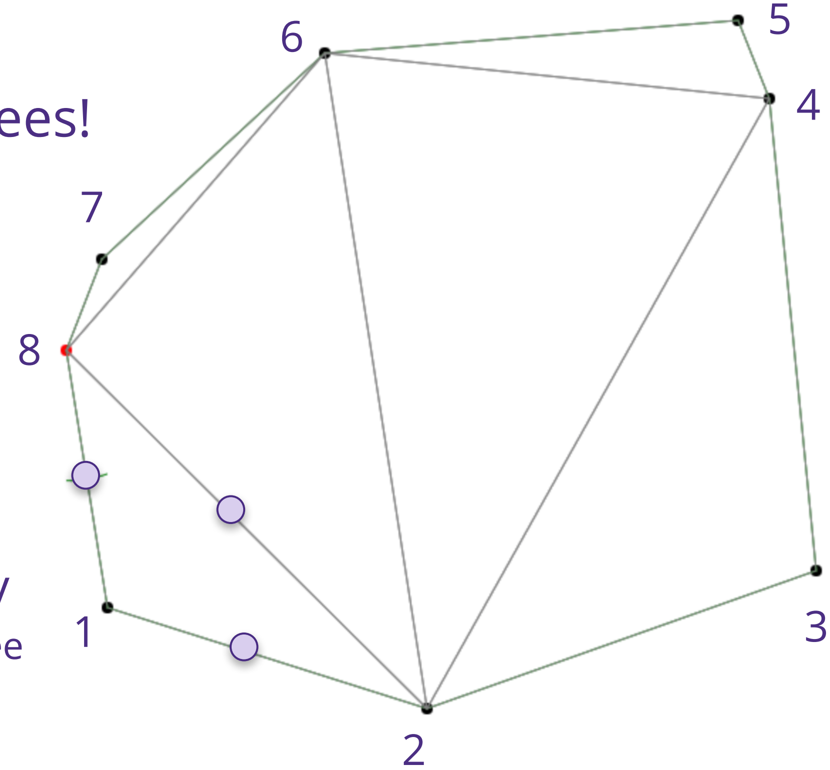
Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- 1 and n are both connected to some i
- (1,i,n) triangle cuts polygon in 1–2 pieces
- 1 .. i and i .. n are triangulated separately



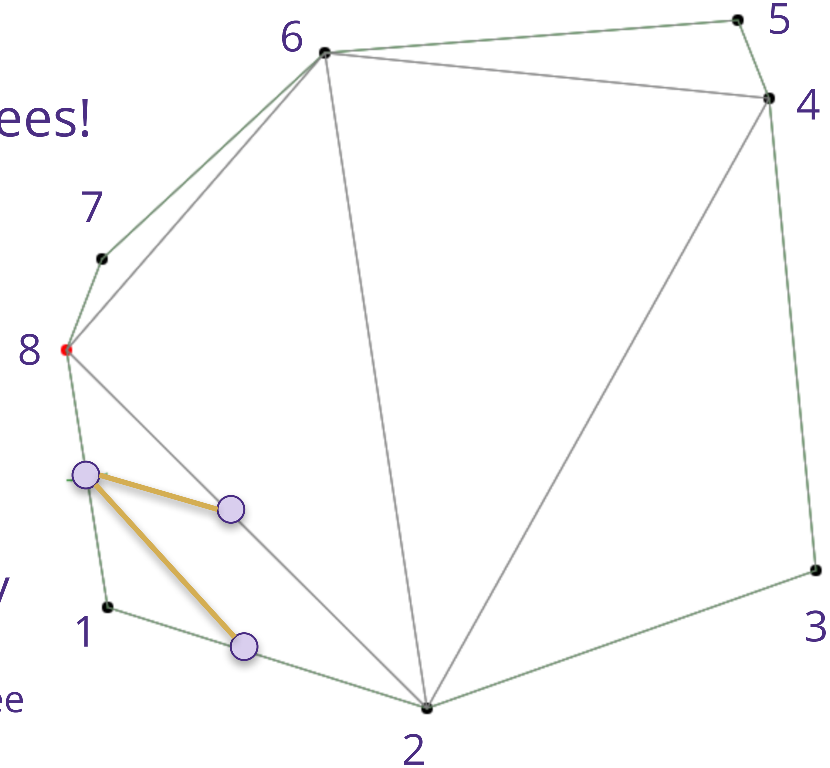
Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- 1 and n are both connected to some i
- 1 .. i and i .. n are triangulated separately
  - (1,i) is root of one subtree
  - (i,n) is root of the other



Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- 1 and n are both connected to some i
- 1 .. i and i .. n are triangulated separately
  - (1,2) is an edge => leaf
  - (2,8) is a chord => subtree

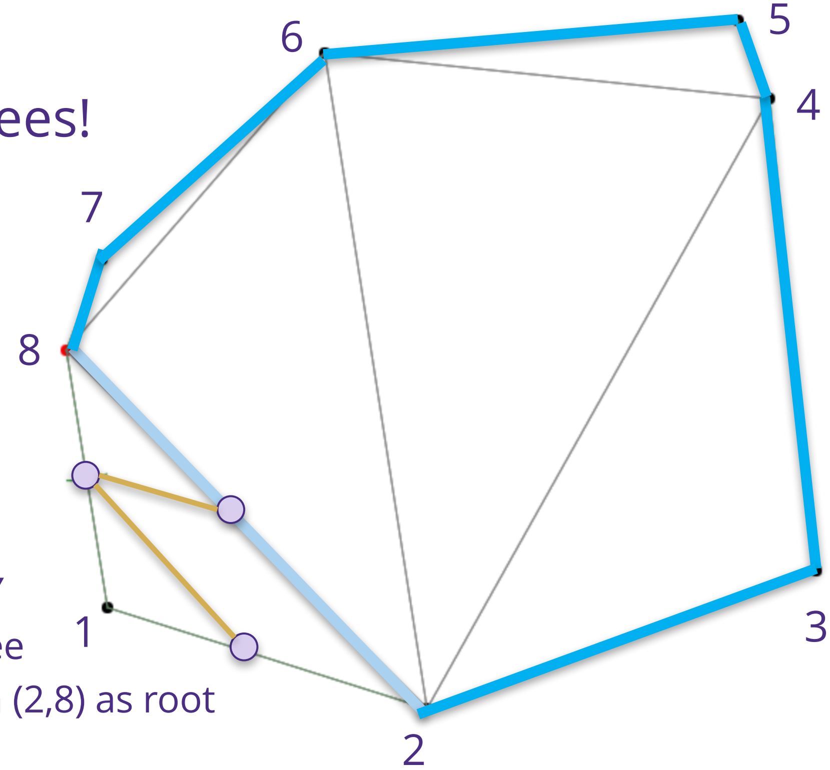


Optimal Polygon Triangulation

> Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)

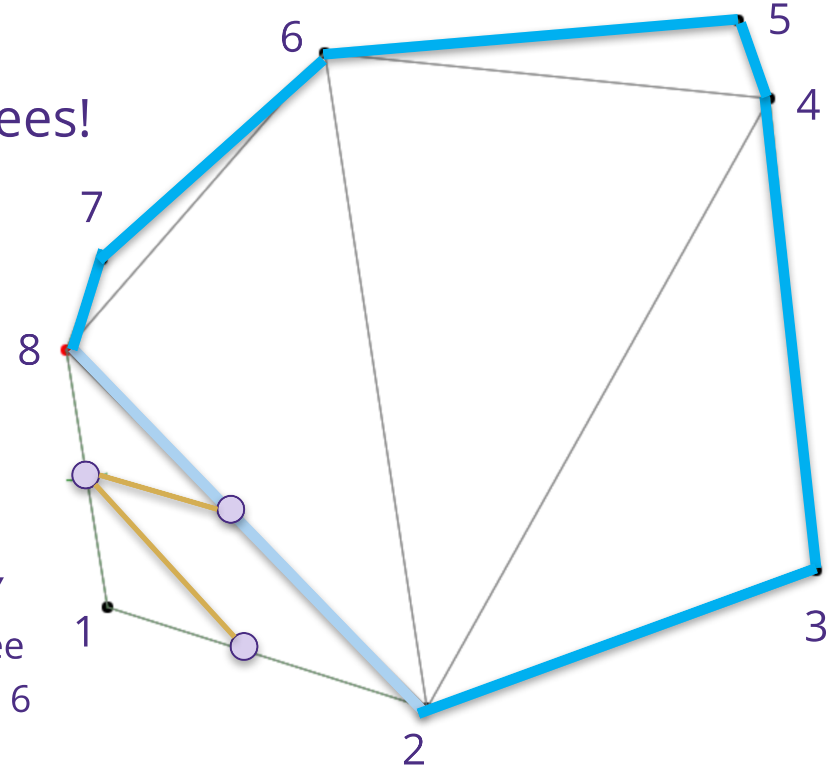
> 1 and n are both connected to some i

> 1 .. i and i .. n are triangulated *recursively*
  - (2,8) is a chord => subtree
  - triangulation of 2..8 with (2,8) as root



Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- 1 and n are both connected to some i
- 1 .. i and i .. n are triangulated *recursively*
  - (2,8) is a chord => subtree
  - (2,8) makes triangle with 6

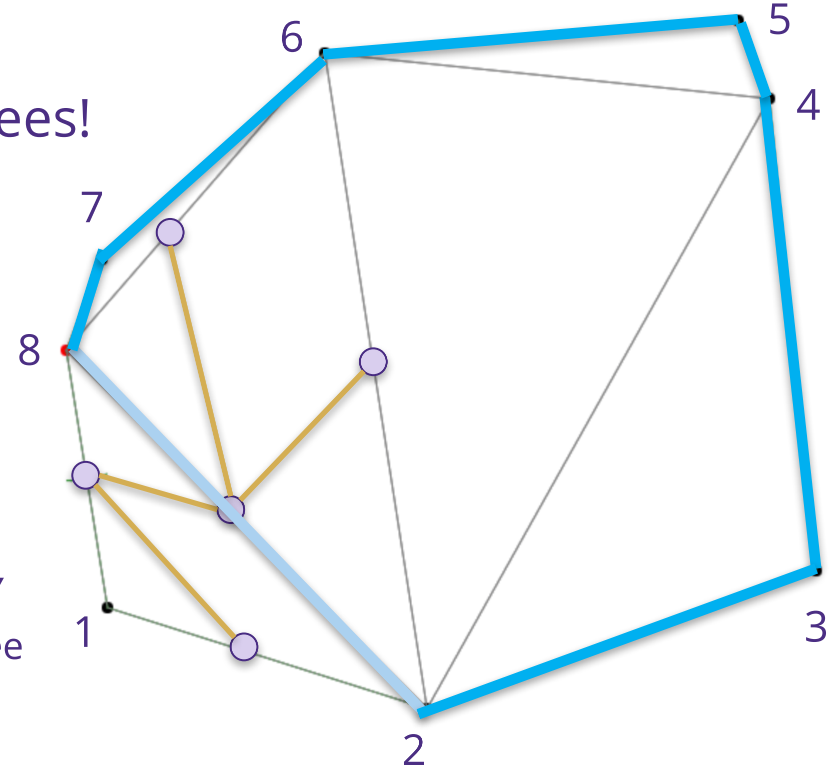


Optimal Polygon Triangulation

> Triangulations are trees!
  – leaves are edges
  – root is edge (1,n)

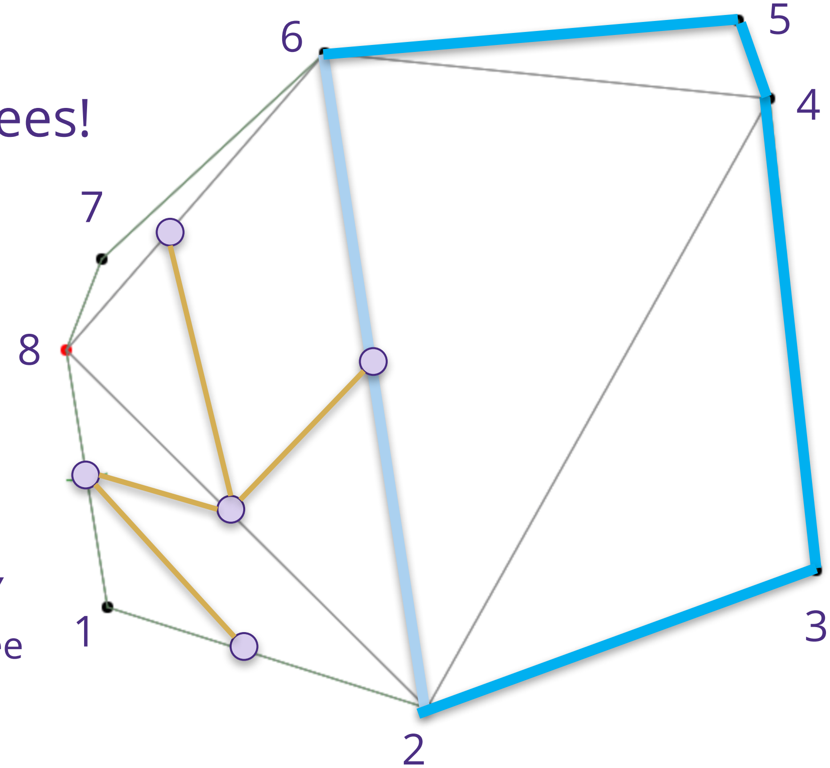
> 1 and n are both connected to some i

> 1 .. i and i .. n are triangulated *recursively*
  – (1,i) is root of one subtree
  – (i,n) is root of the other



Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- 1 and n are both connected to some i
- 1 .. i and i .. n are triangulated *recursively*
  - (1,i) is root of one subtree
  - (i,n) is root of the other

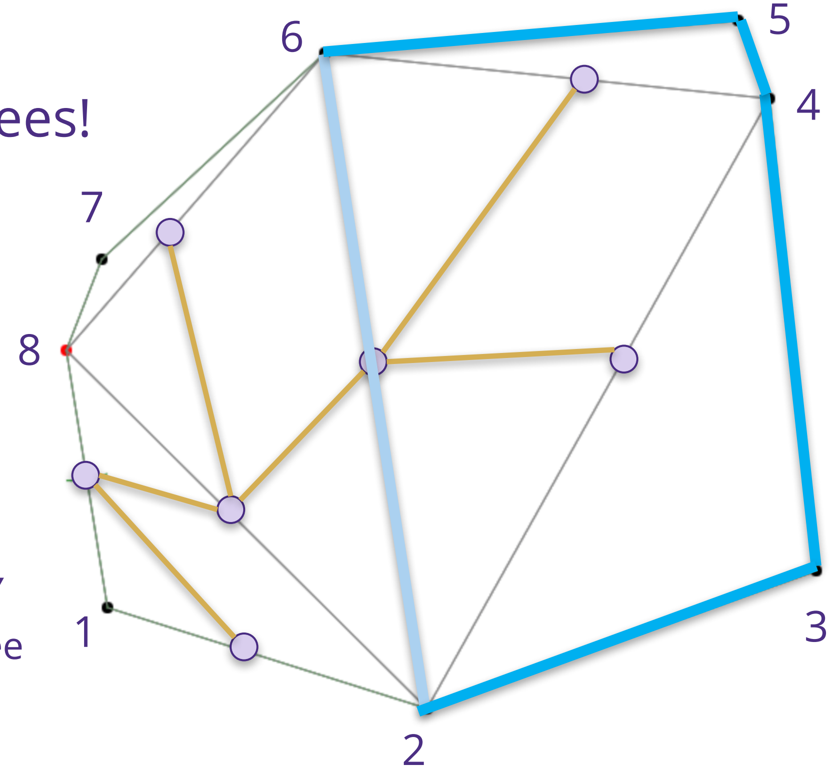


Optimal Polygon Triangulation

> Triangulations are trees!
  – leaves are edges
  – root is edge (1,n)

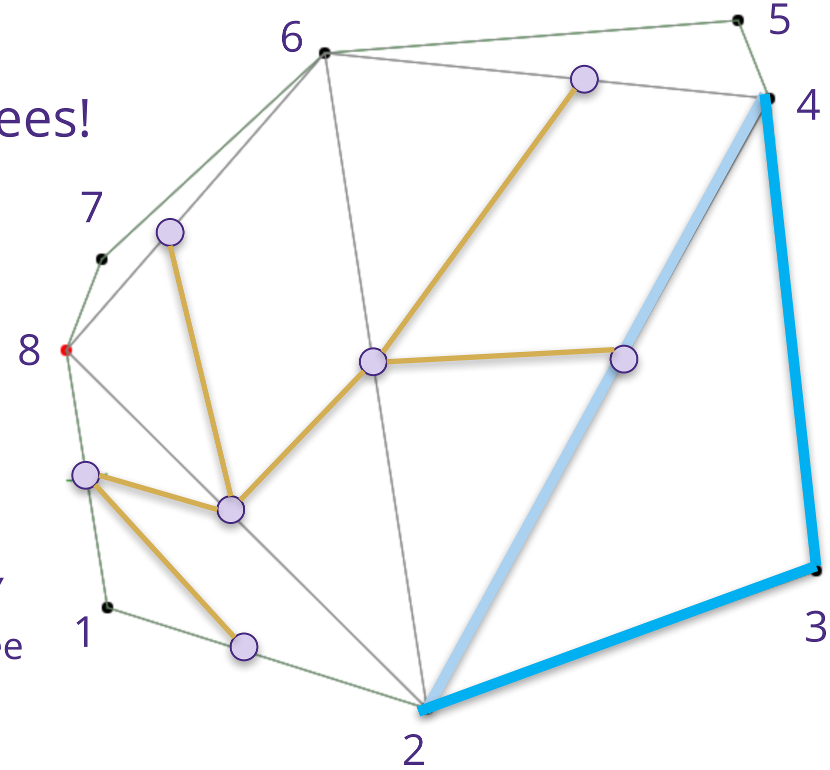
> 1 and n are both connected to some i

> 1 .. i and i .. n are triangulated *recursively*
  – (1,i) is root of one subtree
  – (i,n) is root of the other



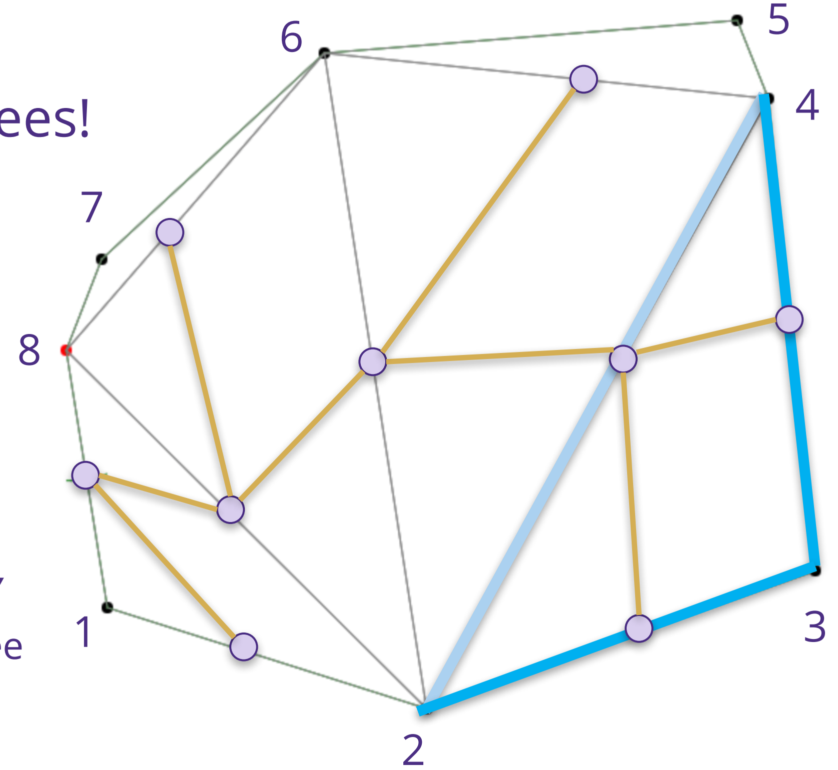
Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- 1 and n are both connected to some i
- 1 .. i and i .. n are triangulated *recursively*
  - (1,i) is root of one subtree
  - (i,n) is root of the other



Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- 1 and n are both connected to some i
- 1 .. i and i .. n are triangulated *recursively*
  - (1,i) is root of one subtree
  - (i,n) is root of the other

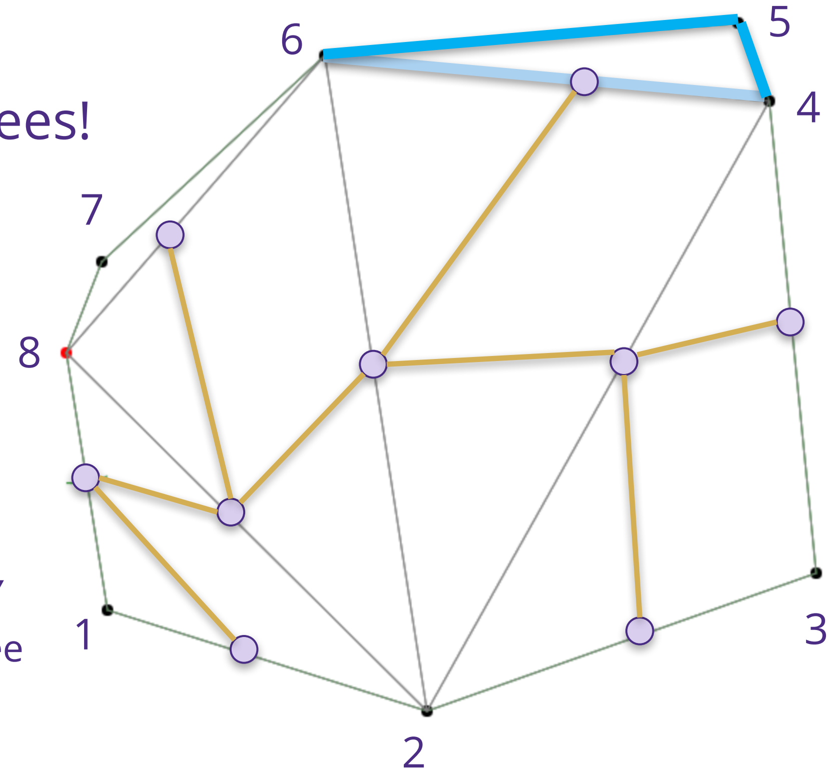


Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)

- 1 and n are both connected to some i

- 1 .. i and i .. n are triangulated *recursively*
  - (1,i) is root of one subtree
  - (i,n) is root of the other

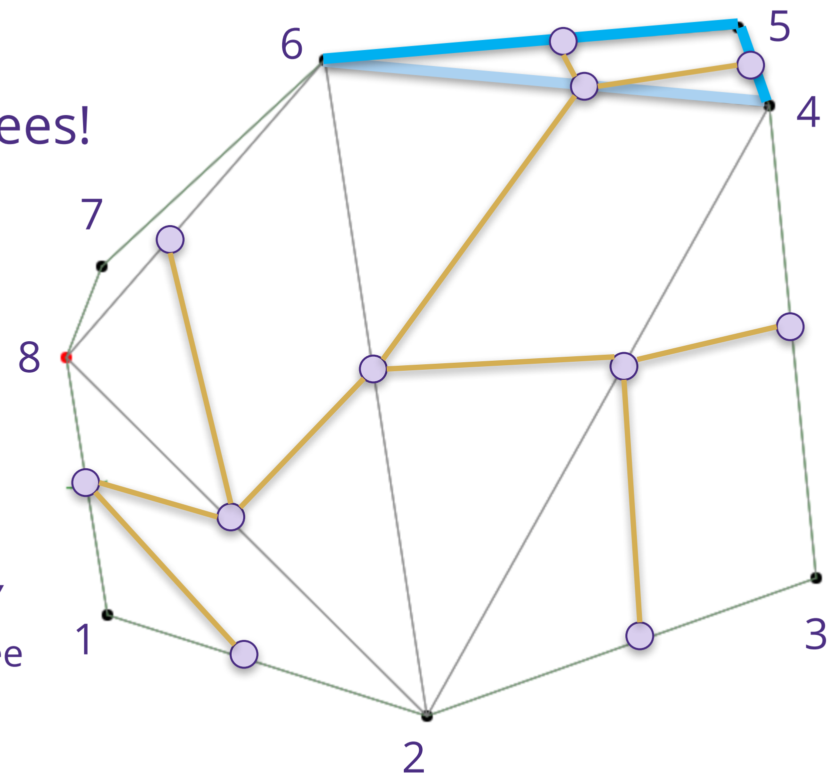


Optimal Polygon Triangulation

> Triangulations are trees!
  – leaves are edges
  – root is edge (1,n)

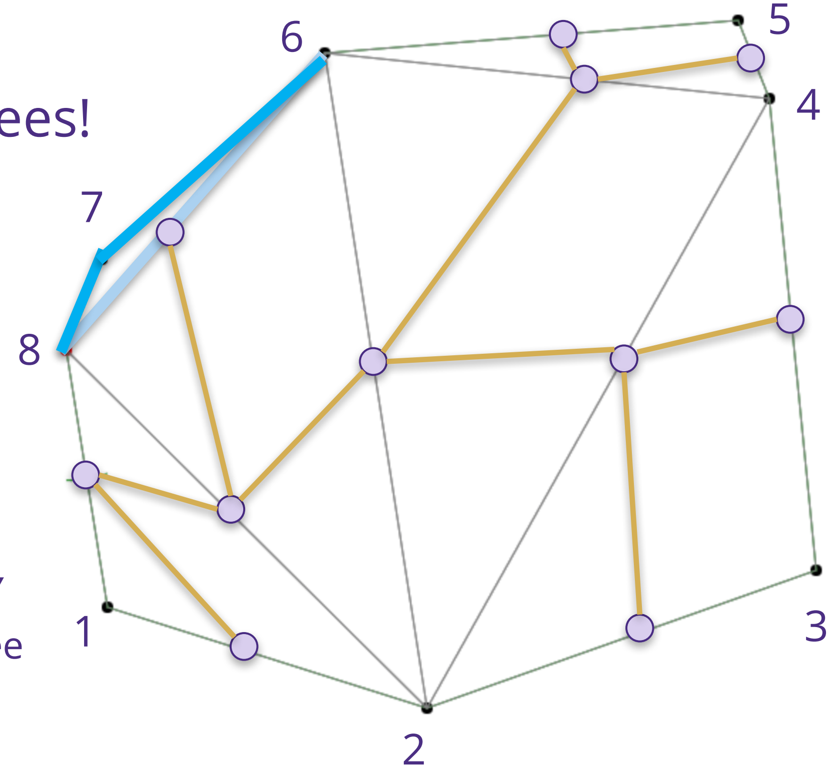
> 1 and n are both connected to some i

> 1 .. i and i .. n are triangulated *recursively*
  – (1,i) is root of one subtree
  – (i,n) is root of the other



Optimal Polygon Triangulation

- Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)
- 1 and n are both connected to some i
- 1 .. i and i .. n are triangulated *recursively*
  - (1,i) is root of one subtree
  - (i,n) is root of the other

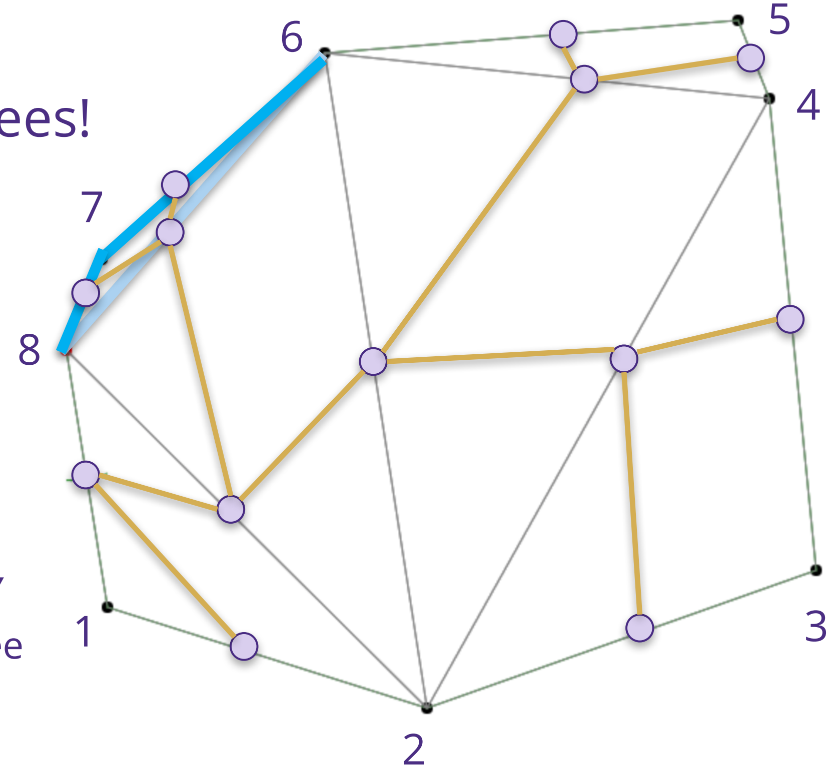


Optimal Polygon Triangulation

> Triangulations are trees!
  – leaves are edges
  – root is edge (1,n)

> 1 and n are both connected to some i

> 1 .. i and i .. n are triangulated *recursively*
  – (1,i) is root of one subtree
  – (i,n) is root of the other

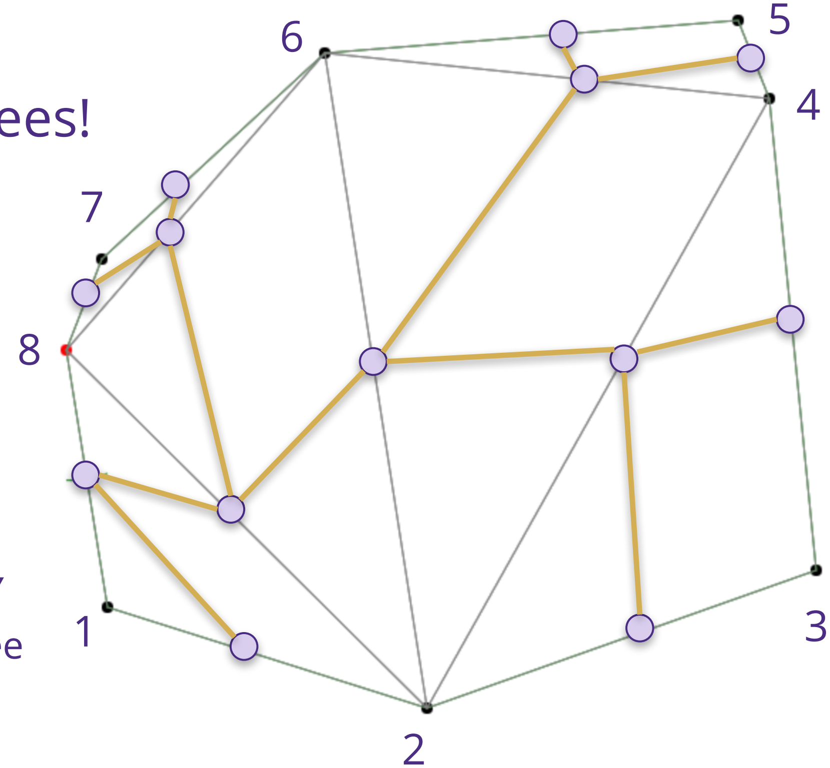


Optimal Polygon Triangulation

> Triangulations are trees!
  - leaves are edges
  - root is edge (1,n)

> 1 and n are both connected to some i

> 1 .. i and i .. n are triangulated *recursively*
  - (1,i) is root of one subtree
  - (i,n) is root of the other



Optimal Polygon Triangulation

- Any triangulation is a tree.
- Likewise, any tree with edges as leaves is a triangulation.
  - (not hard to check)
- Hence, this is essentially the same problem as the other two, so the same algorithm should work...

Optimal Polygon Triangulation

> Apply dynamic programming…

1. Can find opt triangulation of 1 .. n given opt for each 1 .. i and i .. n (see below)
2. To apply this recursively, we need opt on every range i .. j
3. Solve sub-problems starting from ranges of size 3:
   - opt value on i i+1 i+2 = value for that triangle (there's only one triangulation)
   - opt value on i .. j =
     min (opt value on i .. k) + (opt value on k .. j) + value of triangle (i, k, j)
     over k = i+1 .. j-1

> Can replace "+" with any associative op
> Can replace "min" with "max"
> Value on individual triangles is arbitrary

} actually generalizes the other two problems

Optimal Polygon Triangulation

> Apply dynamic programming...

1. Can find opt triangulation of 1 .. n given opt for each 1 .. i and i .. n (see below)
2. To apply this recursively, we need opt on every range i .. j
3. Solve sub-problems starting from ranges of size 3:
   - opt value on i i+1 i+2 = value for that triangle (there's only one triangulation)
   - opt value on i .. j =
     min (opt value on i .. k) + (opt value on k .. j) + value of triangle (i, k, j)
     over k = i+1 .. j-1

> Total running time is $O(n^3)$ as before

Optimal Polygon Triangulation

- You will solve this problem (on paper) in HW5
  - (actually, you can use Excel / Google Docs)