Farida

Problem: Find the maximum sum of a subset of $m_1, m_2, \ldots, m_n$ which does not contain consecutive elements.

Farida

Problem: Find the maximum sum of a subset of $m_1, m_2, \ldots, m_n$ which does not contain consecutive elements.

- Let $g(k)$ be the answer if we only consider the first k elements

Farida

Problem: Find the maximum sum of a subset of $m_1, m_2, \ldots, m_n$ which does not contain consecutive elements.

- Let $g(k)$ be the answer if we only consider the first k elements
  - Base cases: $g(0) = 0$, $g(1) = m_1$

Farida

Problem: Find the maximum sum of a subset of $m_1, m_2, \ldots, m_n$ which does not contain consecutive elements.

- Let $g(k)$ be the answer if we only consider the first k elements
  - Base cases: $g(0) = 0$, $g(1) = m_1$
- We can either take the k-th element or don't

Farida

Problem: Find the maximum sum of a subset of $m_1, m_2, \ldots, m_n$ which does not contain consecutive elements.

- Let $g(k)$ be the answer if we only consider the first k elements
  - Base cases: $g(0) = 0$, $g(1) = m_1$
- We can either take the k-th element or don't
  - $g(k) = \max(m_k + g(k-2), g(k-1))$

Farida

Problem: Find the maximum sum of a subset of $m_1, m_2, \ldots, m_n$ which does not contain consecutive elements.

- Let $g(k)$ be the answer if we only consider the first k elements
  - Base cases: $g(0) = 0$, $g(1) = m_1$
- We can either take the k-th element or don't
  - $g(k) = \max(m_k + g(k-2), g(k-1))$
- Runs in $\mathcal{O}(n)$

Farida

- Can we solve this without using $\mathcal{O}(n)$ memory?
  - Reminder: $g(k) = \max(m_k + g(k-2), g(k-1))$

Farida

- Can we solve this without using $\mathcal{O}(n)$ memory?
  - Reminder: $g(k) = \max(m_k + g(k-2), g(k-1))$
- We're only accessing the two previous entries

```
prev = 0
prevprev = 0
for mi in m:
    prev, prevprev = max(prev, prevprev + m), prev
print(prev)
```

- You might know this trick from Fibonacci numbers

Farida

- How to notice possible integer overflows?

Farida

- How to notice possible integer overflows?
  - 32-bit integers: roughly up to 10^9
  - 64-bit integers: roughly up to 10^{18}

Farida

- How to notice possible integer overflows?
  - 32-bit integers: roughly up to 10^9
  - 64-bit integers: roughly up to 10^{18}
  - Roughly estimate largest possible value for variables in your program, always rounding up

Farida

- How to notice possible integer overflows?
  - 32-bit integers: roughly up to 10^9
  - 64-bit integers: roughly up to 10^{18}
  - Roughly estimate largest possible value for variables in your program, always rounding up
  - When in doubt: use larger integer type

Farida

- How to notice possible integer overflows?
  - 32-bit integers: roughly up to 10^9
  - 64-bit integers: roughly up to 10^{18}
  - Roughly estimate largest possible value for variables in your program, always rounding up
  - When in doubt: use larger integer type
- Example for this problem
  - There's 10^5 integers
  - Each integer is up to 10^5
  - Sum can be up to $10^{10} \to$ needs 64 bits

Hot Dog

Hot Dog

- Known as Rod Cutting

Hot Dog

- Known as Rod Cutting
- Greedily chosing longest/most expensive/highest value per length segment won't work

Hot Dog

- Known as Rod Cutting
- Greedily chosing longest/most expensive/highest value per length segment won't work
- Use dynamic programming instead

Hot Dog

- Known as Rod Cutting
- Greedily chosing longest/most expensive/highest value per length segment won't work
- Use dynamic programming instead
- Example table T:

Length l	1	3	5
Price p	4	20	70

Hot Dog

- Known as Rod Cutting
- Greedily chosing longest/most expensive/highest value per length segment won't work
- Use dynamic programming instead
- Example table T:

Length l	1	3	5
Price p	4	20	70

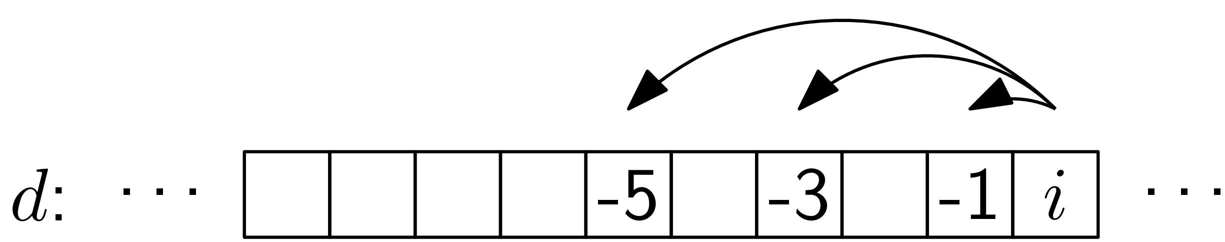
- $d[i] = \max_{t \in T} \left(d[i - t_l] + t_p \right)$

Hot Dog

- Known as Rod Cutting
- Greedily chosing longest/most expensive/highest value per length segment won't work
- Use dynamic programming instead
- Example table T:

Length l	1	3	5
Price p	4	20	70

- $d[i] = \max_{t \in T} \left(d[i - t_l] + t_p\right)$



Gamenight

Problem: 2D-nim game on a $w \times h$ grid. Two players take turns moving either 1 or 2 moves down, or 1 or 2 moves right, starting from the top left corner. Player with turn in the bottom right corner wins. Should we take the first turn to win?

Gamenight

Problem: 2D-nim game on a $w \times h$ grid. Two players take turns moving either 1 or 2 moves down, or 1 or 2 moves right, starting from the top left corner. Player with turn in the bottom right corner wins. Should we take the first turn to win?

- $\mathcal{O}(wh)$ DP works ($w \cdot h \approx 10^7$).

Gamenight

Problem: 2D-nim game on a $w \times h$ grid. Two players take turns moving either 1 or 2 moves down, or 1 or 2 moves right, starting from the top left corner. Player with turn in the bottom right corner wins. Should we take the first turn to win?

- $\mathcal{O}(wh)$ DP works $(w \cdot h \approx 10^7)$.
- Idea:
  - You win in the bottom-right cell
  - Otherwise: you win if you can bring the other player into a losing position

Gamenight

- Let $\mathrm{win}(x, y)$ be 1, if it's your turn at position (x, y), and 0 if you lose

Gamenight

- Let $\mathrm{win}(x, y)$ be 1, if it's your turn at position (x, y), and 0 if you lose
- Base case: $\mathrm{win}(w, h) = 1$

Gamenight

- Let $\mathrm{win}(x, y)$ be 1, if it's your turn at position (x, y), and 0 if you lose
- Base case: $\mathrm{win}(w, h) = 1$
- Otherwise, $\mathrm{win}(x, y)$ is 1 if $\mathrm{win}(x', y') = 0$ for any of the up to four reachable positions (x', y')

Gamenight

- Let $\mathrm{win}(x, y)$ be 1, if it's your turn at position (x, y), and 0 if you lose
- Base case: $\mathrm{win}(w, h) = 1$
- Otherwise, $\mathrm{win}(x, y)$ is 1 if $\mathrm{win}(x', y') = 0$ for any of the up to four reachable positions (x', y')
  - Ignore moves that lead outside of the board (since we explicitly handled the bottom-right corner, there's always at least one valid move)

Gamenight

- Let $\mathrm{win}(x, y)$ be 1, if it's your turn at position (x, y), and 0 if you lose
- Base case: $\mathrm{win}(w, h) = 1$
- Otherwise, $\mathrm{win}(x, y)$ is 1 if $\mathrm{win}(x', y') = 0$ for any of the up to four reachable positions (x', y')
  - Ignore moves that lead outside of the board (since we explicitly handled the bottom-right corner, there's always at least one valid move)
- Answer: $\mathrm{win}(1, 1)$

Gamenight

- Let $\text{win}(x, y)$ be 1, if it's your turn at position (x, y), and 0 if you lose
- Base case: $\text{win}(w, h) = 1$
- Otherwise, $\text{win}(x, y)$ is 1 if $\text{win}(x', y') = 0$ for any of the up to four reachable positions (x', y')
  - Ignore moves that lead outside of the board (since we explicitly handled the bottom-right corner, there's always at least one valid move)
- Answer: $\text{win}(1, 1)$
- You can also reverse the game to go up and left, which might be easier to implement

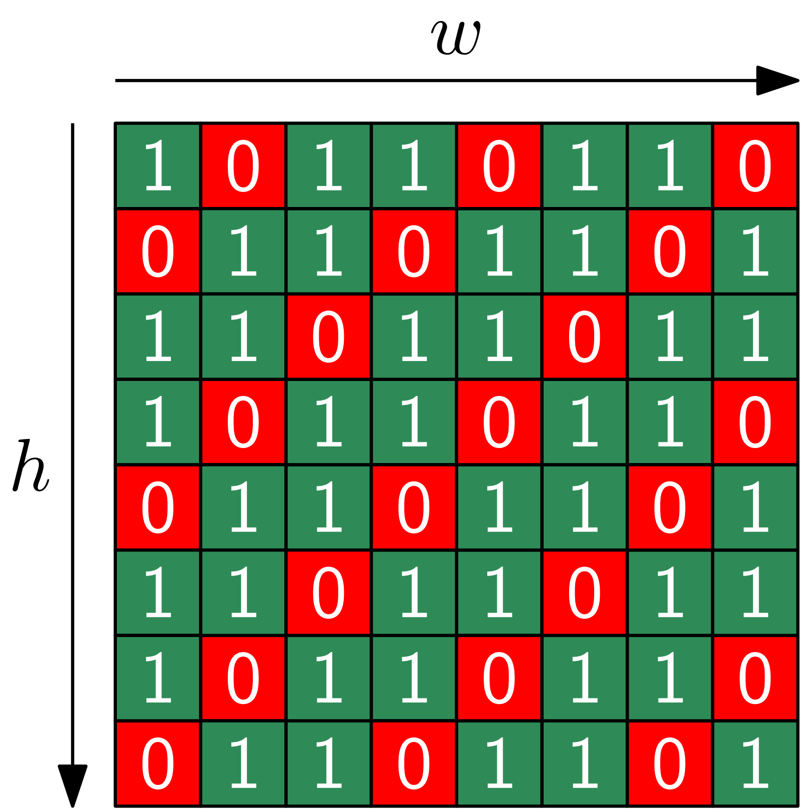
Gamenight

- What is up with those 1ms solutions?

C
0.001 s 1 try
0.001 s 3 tries
0.001 s 4 tries
1 try
0.089 s 6 tries
0.698 s 6 tries
4 tries
0.043 s 3 tries
0.128 s 2 tries

Gamenight

- What is up with those 1ms solutions?



C
0.001 s 1 try
0.001 s 3 tries
0.001 s 4 tries
1 try
0.089 s 6 tries
0.698 s 6 tries
4 tries
0.043 s 3 tries
0.128 s 2 tries

Gamenight

- What is up with those 1ms solutions?

$w \rightarrow$, $h \downarrow$

1	0	1	1	0	1	1	0
0	1	1	0	1	1	0	1
1	1	0	1	1	0	1	1
1	0	1	1	0	1	1	0
0	1	1	0	1	1	0	1
1	1	0	1	1	0	1	1
1	0	1	1	0	1	1	0
0	1	1	0	1	1	0	1

- Start is losing iff. $w + h \equiv 0 \bmod 3$

C
0.001 s 1 try
0.001 s 3 tries
0.001 s 4 tries
1 try
0.089 s 6 tries
0.698 s 6 tries
4 tries
0.043 s 3 tries
0.128 s 2 tries

Party

Problem: given a tree, find the maximum sum of node weights in an independent set

Party

Problem: given a tree, find the maximum sum of node weights in an independent set

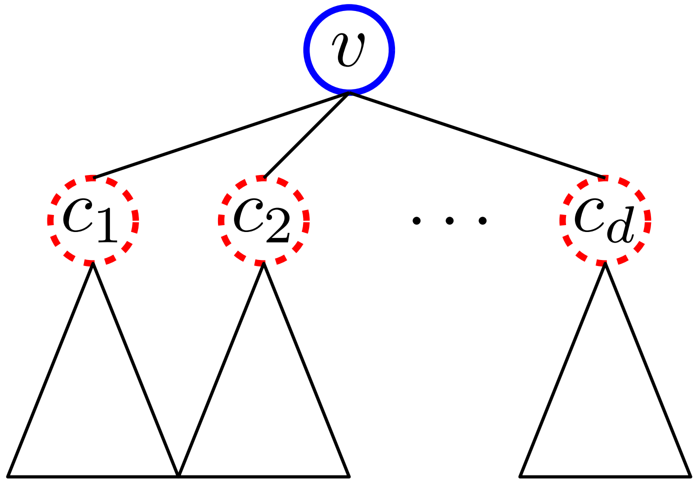
- For each node v, consider two cases

Party

Problem: given a tree, find the maximum sum of node weights in an independent set

- For each node v, consider two cases

take v



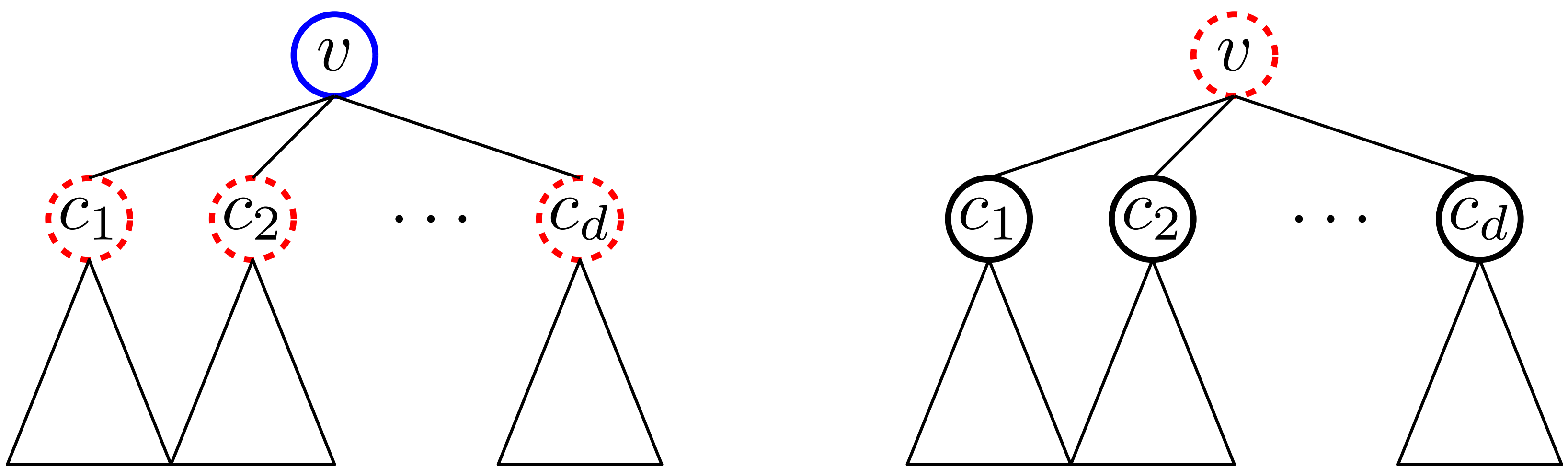
none of the children
can be taken

Party

Problem: given a tree, find the maximum sum of node weights in an independent set

- For each node v, consider two cases

take v | don't take v



none of the children can be taken | children can be taken, but don't have to be

Party

- For each node v, compute two values

Party

- For each node v, compute two values
  - $\mathrm{best}[v]$: best solution for the subtree rooted at v (which might or might not contain v)

Party

- For each node v, compute two values
  - $\text{best}[v]$: best solution for the subtree rooted at v (which might or might not contain v)
  - $\text{without}[v]$: best solution for the subtree rooted at v which does not contain v

Party

- For each node v, compute two values
  - $\text{best}[v]$: best solution for the subtree rooted at v (which might or might not contain v)
  - $\text{without}[v]$: best solution for the subtree rooted at v which does not contain v
- If v is a leaf, $\text{best}(v) = \text{fun}(v)$ and $\text{without}(v) = 0$

Party

- For each node v, compute two values
  - $\mathrm{best}[v]$: best solution for the subtree rooted at v (which might or might not contain v)
  - $\mathrm{without}[v]$: best solution for the subtree rooted at v which does not contain v
- If v is a leaf, $\mathrm{best}(v) = \mathrm{fun}(v)$ and $\mathrm{without}(v) = 0$
- In general:

Party

- For each node v, compute two values
  - $\text{best}[v]$: best solution for the subtree rooted at v (which might or might not contain v)
  - $\text{without}[v]$: best solution for the subtree rooted at v which does not contain v
- If v is a leaf, $\text{best}(v) = \text{fun}(v)$ and $\text{without}(v) = 0$
- In general:

$$\text{without}(v) = \sum_{\text{child } c} \text{best}(c)$$

Party

- For each node v, compute two values
  - $\text{best}[v]$: best solution for the subtree rooted at v (which might or might not contain v)
  - $\text{without}[v]$: best solution for the subtree rooted at v which does not contain v
- If v is a leaf, $\text{best}(v) = \text{fun}(v)$ and $\text{without}(v) = 0$
- In general:

$$\text{without}(v) = \sum_{\text{child } c} \text{best}(c)$$

$$\text{best}(v) = \max(\text{without}(v), \text{fun}(v) + \sum_{\text{child } c} \text{without}(c))$$

Party

- For DPs on trees, DFS is usually simpler than BFS

Party

- For DPs on trees, DFS is usually simpler than BFS
- Often we can directly return the DP value from the DFS function instead of putting it into a table

Party

- For DPs on trees, DFS is usually simpler than BFS
- Often we can directly return the DP value from the DFS function instead of putting it into a table

```
def dfs(v, adj, weight):
    without = 0
    with = weight[v]
    for v2 in adj[v]:
        (best2, without2) = dfs(v2, adj, weight)
        without += best2
        with += without2
    return max(with, without), without
```

Party

- For DPs on trees, DFS is usually simpler than BFS
- Often we can directly return the DP value from the DFS function instead of putting it into a table

```
def dfs(v, adj, weight):
    without = 0
    with = weight[v]
    for v2 in adj[v]:
        (best2, without2) = dfs(v2, adj, weight)
        without += best2
        with += without2
    return max(with, without), without
```

- The input is already in BFS order, so BFS is actually easy here

Market

Problem: Given an array a print the length of the longest strictly increasing subarray after at most one edit

Market

Problem: Given an array a print the length of the longest strictly increasing subarray after at most one edit

no equal elements

Market

Problem: Given an array a print the length of the longest strictly increasing subarray after at most one edit

no equal elements

- We can only change one element, so we should try to use existing strictly increasing subarrays

Market

Problem: Given an array a print the length of the longest strictly increasing subarray after at most one edit

no equal elements

- We can only change one element, so we should try to use existing strictly increasing subarrays

Market

Problem: Given an array a print the length of the longest strictly increasing subarray after at most one edit

no equal elements

- We can only change one element, so we should try to use existing strictly increasing subarrays

l r

always possible

Market

Problem: Given an array a print the length of the longest strictly increasing subarray after at most one edit

no equal elements

- We can only change one element, so we should try to use existing strictly increasing subarrays

always possible

Market

Problem: Given an array a print the length of the longest strictly increasing subarray after at most one edit

no equal elements

- We can only change one element, so we should try to use existing strictly increasing subarrays

always possible

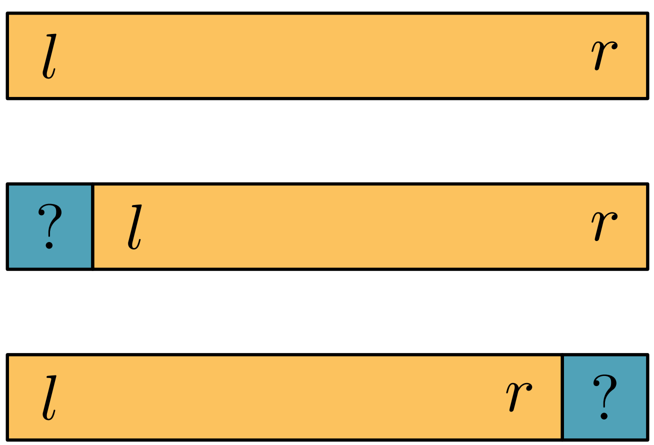
$l \neq 1$

Market

Problem: Given an array a print the length of the longest strictly increasing subarray after at most one edit

no equal elements

- We can only change one element, so we should try to use existing strictly increasing subarrays



always possible

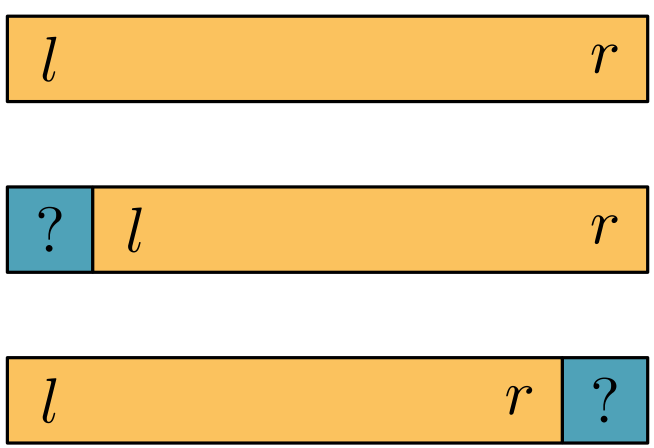
$l \neq 1$

Market

Problem: Given an array a print the length of the longest strictly increasing subarray after at most one edit

no equal elements

- We can only change one element, so we should try to use existing strictly increasing subarrays



always possible

$l \neq 1$

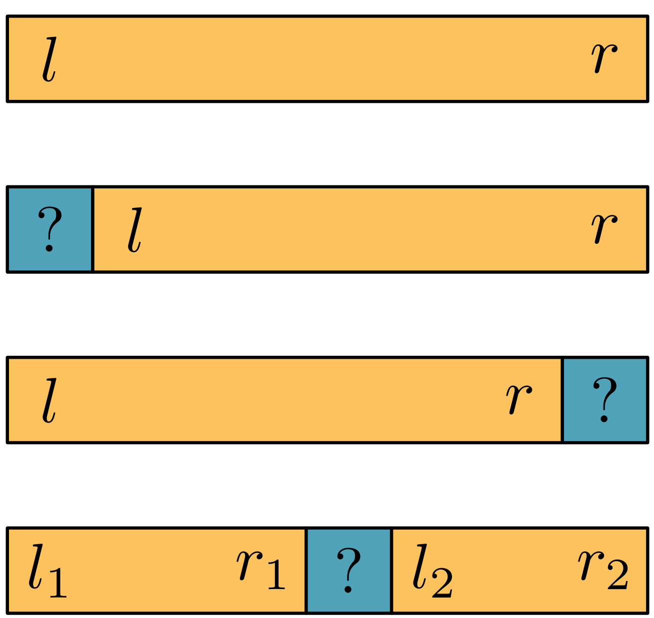
$r \neq n$

Market

Problem: Given an array a print the length of the longest strictly increasing subarray after at most one edit

no equal elements

- We can only change one element, so we should try to use existing strictly increasing subarrays



always possible

$l \neq 1$

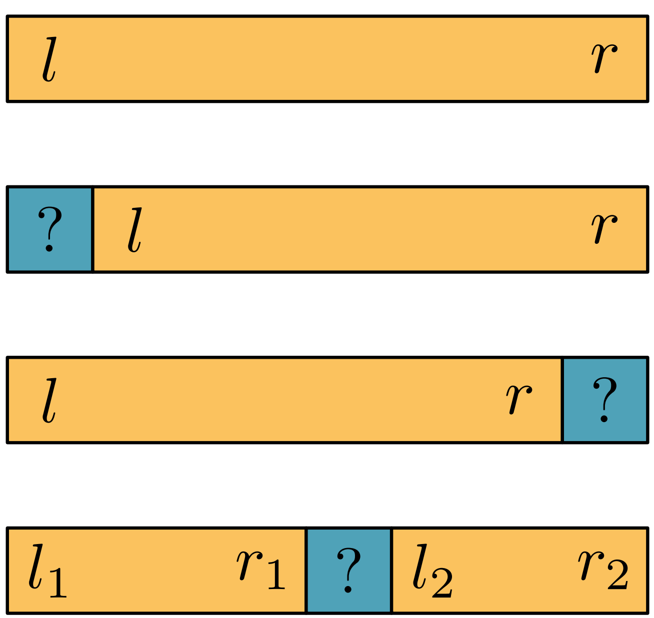
$r \neq n$

Market

Problem: Given an array a print the length of the longest strictly increasing subarray after at most one edit

no equal elements

- We can only change one element, so we should try to use existing strictly increasing subarrays



always possible

$l \neq 1$

$r \neq n$

$r_1 + 2 = l_2$ and $a_{r_1} + 1 < a_{l_2}$

Market

- Precompute information about existing subarrays

Market

- Precompute information about existing subarrays
- $\mathrm{ending\_at}[i]$: longest strictly increasing subarray ending at position i

Market

- Precompute information about existing subarrays
- $\mathrm{ending\_at}[i]$: longest strictly increasing subarray ending at position i
- Can be computed with dynamic programming

Market

- Precompute information about existing subarrays
- $\text{ending\_at}[i]$: longest strictly increasing subarray ending at position i
- Can be computed with dynamic programming
  - If $a_{i-1} < a_i$, we can extend the subarray ending at $i - 1$ by one element

Market

- Precompute information about existing subarrays
- $\text{ending\_at}[i]$: longest strictly increasing subarray ending at position i
- Can be computed with dynamic programming
  - If $a_{i-1} < a_i$, we can extend the subarray ending at $i-1$ by one element
  - Otherwise, the longest subarray only consists of a_i

Market

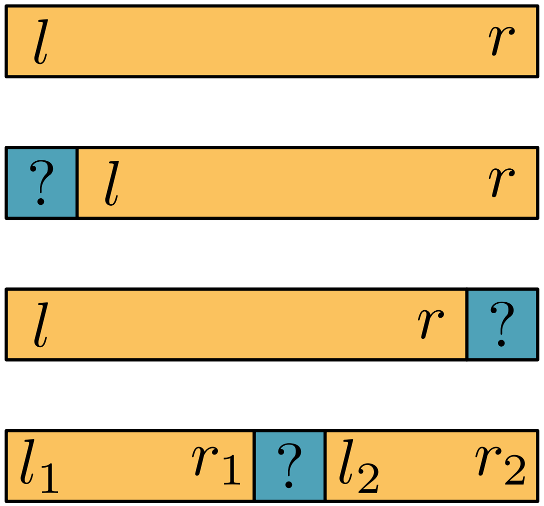
- Precompute information about existing subarrays
- $\text{ending\_at}[i]$: longest strictly increasing subarray ending at position i
- Can be computed with dynamic programming
  - If $a_{i-1} < a_i$, we can extend the subarray ending at $i - 1$ by one element
  - Otherwise, the longest subarray only consists of a_i
    - Also the base case for $i = 1$

Market

- Precompute information about existing subarrays
- $\text{ending\_at}[i]$: longest strictly increasing subarray ending at position i
- Can be computed with dynamic programming
  - If $a_{i-1} < a_i$, we can extend the subarray ending at $i - 1$ by one element
  - Otherwise, the longest subarray only consists of a_i
    - Also the base case for $i = 1$
- $\text{starting\_at}[i]$ can be defined & computed analogously

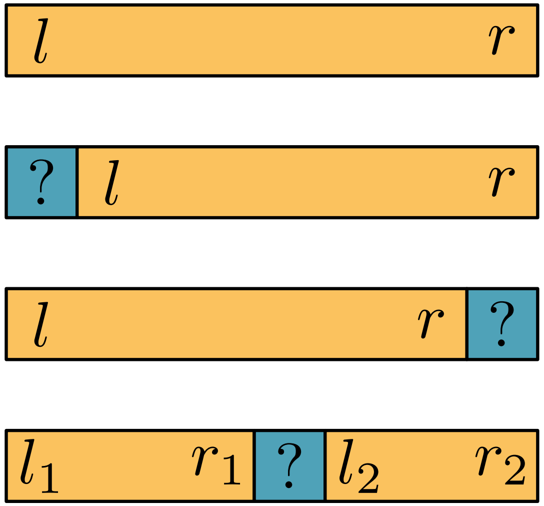
Market

- For each i, consider 4 possibilities



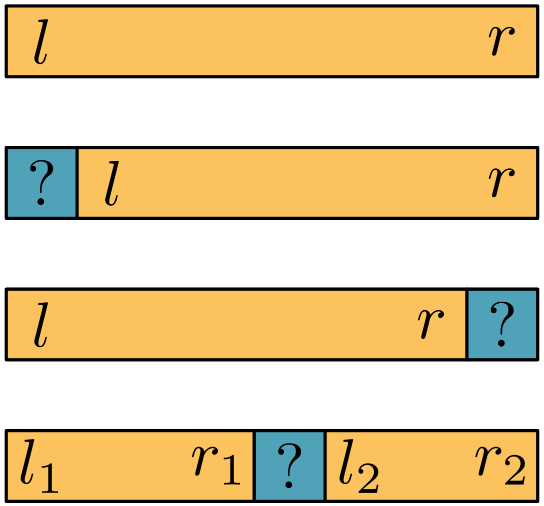
Market

- For each i, consider 4 possibilities
  - $\text{starting\_at}[i]$



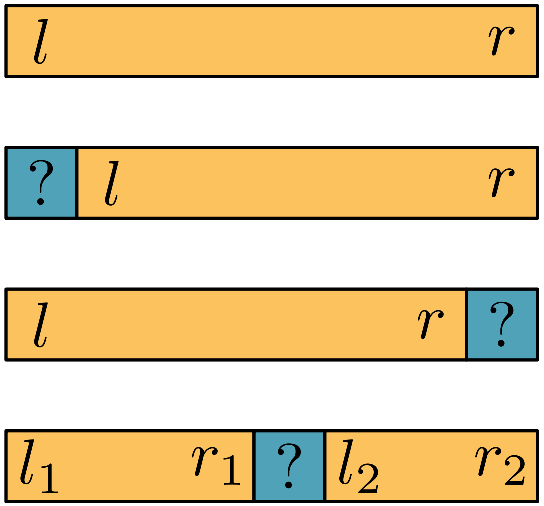
Market

- For each i, consider 4 possibilities
  - $\text{starting\_at}[i]$
  - $1 + \text{starting\_at}[i+1]$

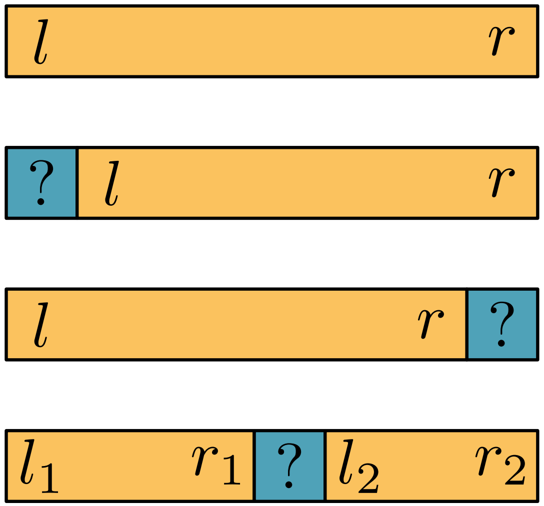


Market

- For each i, consider 4 possibilities
  - $\text{starting\_at}[i]$
  - $1 + \text{starting\_at}[i+1]$
  - $\text{ending\_at}[i-1] + 1$



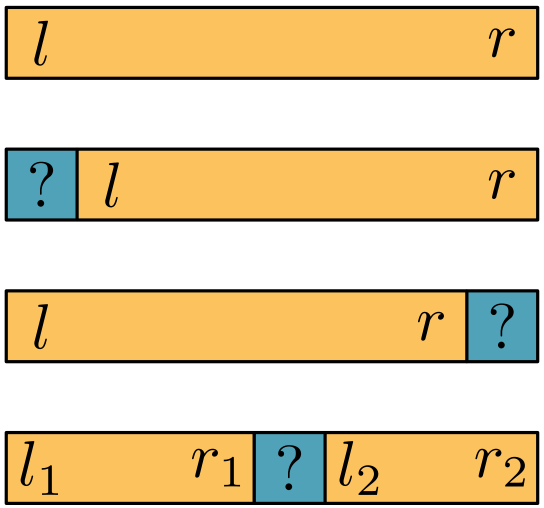
Market



- For each i, consider 4 possibilities
  - $\text{starting\_at}[i]$
  - $1 + \text{starting\_at}[i+1]$
  - $\text{ending\_at}[i-1] + 1$
  - $\text{ending\_at}[i-1] + 1 + \text{starting\_at}[i+1]$ if $a_{i-1} + 1 < a_{i+1}$

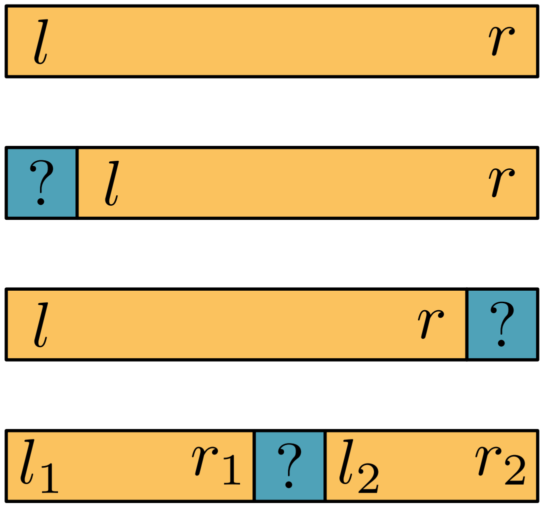
Market

- For each i, consider 4 possibilities
  - $\text{starting\_at}[i]$
  - $1 + \text{starting\_at}[i+1]$
  - $\text{ending\_at}[i-1] + 1$
  - $\text{ending\_at}[i-1] + 1 + \text{starting\_at}[i+1]$ if $a_{i-1} + 1 < a_{i+1}$
- Skip those that would go out of bounds

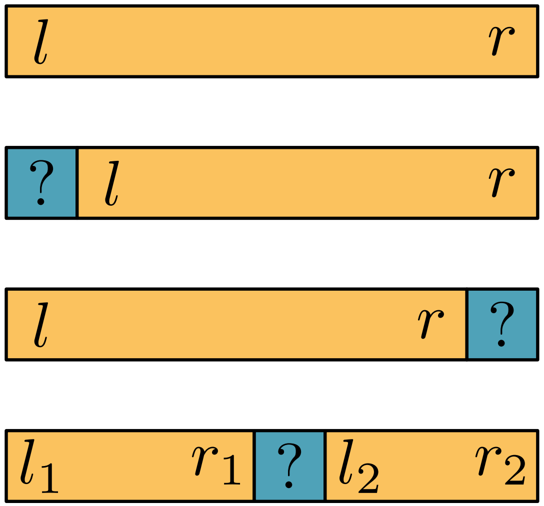


Market

- For each i, consider 4 possibilities
  - $\text{starting\_at}[i]$
  - $1 + \text{starting\_at}[i+1]$
  - $\text{ending\_at}[i-1] + 1$
  - $\text{ending\_at}[i-1] + 1 + \text{starting\_at}[i+1]$ if $a_{i-1} + 1 < a_{i+1}$
- Skip those that would go out of bounds
- Take maximum over all possibilities of all i

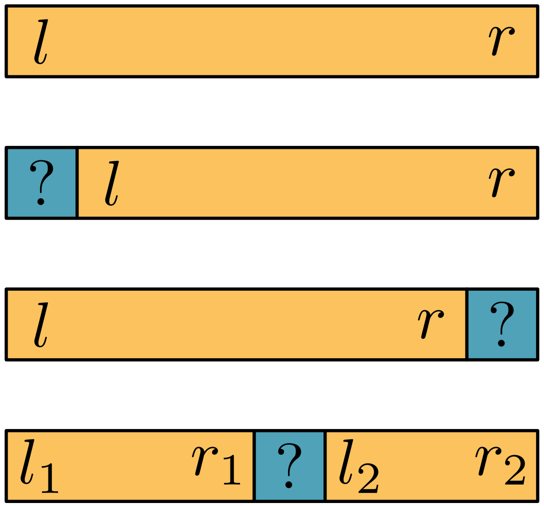


Market



- For each i, consider 4 possibilities
  - $\text{starting\_at}[i]$
  - $1 + \text{starting\_at}[i+1]$
  - $\text{ending\_at}[i-1] + 1$
  - $\text{ending\_at}[i-1] + 1 + \text{starting\_at}[i+1]$ if $a_{i-1} + 1 < a_{i+1}$
- Skip those that would go out of bounds
- Take maximum over all possibilities of all i
- You can prove the first case is unnecessary

Market



- For each i, consider 4 possibilities
  - $\text{starting\_at}[i]$
  - $1 + \text{starting\_at}[i+1]$
  - $\text{ending\_at}[i-1] + 1$
  - $\text{ending\_at}[i-1] + 1 + \text{starting\_at}[i+1]$ if $a_{i-1} + 1 < a_{i+1}$
- Skip those that would go out of bounds
- Take maximum over all possibilities of all i
- You can prove the first case is unnecessary
- You can also remove the second and third case if you handle empty intervals in the fourth case

Laning

Problem: find the minimum diameter of a tree formed by adding edges to a given forest

Laning

Problem: find the minimum diameter of a tree formed by adding edges to a given forest

- it's only optimal to add edges to centers of trees

Laning

Problem: find the minimum diameter of a tree formed by adding edges to a given forest

- it's only optimal to add edges to centers of trees
  - graph center: $\arg\min_{v \in V} \max\{d(v, u) \mid u \in V\}$

Laning

Problem: find the minimum diameter of a tree formed by adding edges to a given forest

- it's only optimal to add edges to centers of trees
  - graph center: $\arg\min_{v \in V} \max\{d(v,u) \mid u \in V\}$
  - graph centroid: $\arg\min_{v \in V} \sum_{u \in V} d(v,u)$

Laning

Problem: find the minimum diameter of a tree formed by adding edges to a given forest

- it's only optimal to add edges to centers of trees
  - graph center: $\arg\min_{v \in V} \max\{d(v,u) \mid u \in V\}$
  - graph centroid: $\arg\min_{v \in V} \sum_{u \in V} d(v,u)$
- in a tree: center is middle of longest path

Laning

Problem: find the minimum diameter of a tree formed by adding edges to a given forest

- it's only optimal to add edges to centers of trees
  - graph center: $\arg\min_{v \in V} \max\{d(v,u) \mid u \in V\}$
  - graph centroid: $\arg\min_{v \in V} \sum_{u \in V} d(v,u)$
- in a tree: center is middle of longest path
- max. distance from center: $\left\lceil \frac{\text{diameter}}{2} \right\rceil$

Laning

Problem: find the minimum diameter of a tree formed by adding edges to a given forest

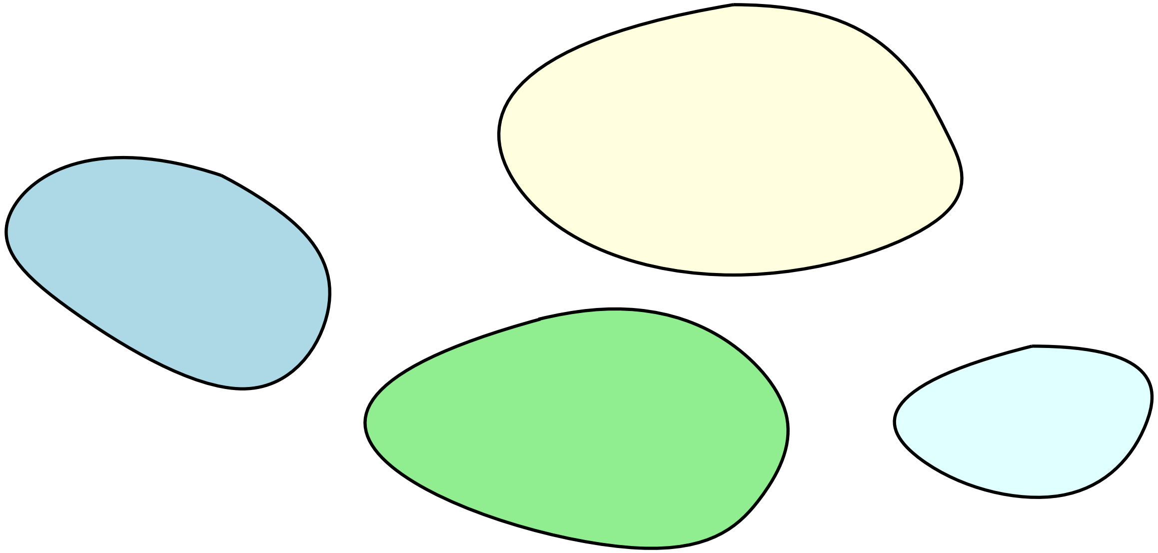
- it's only optimal to add edges to centers of trees
  - graph center: $\arg\min_{v \in V} \max\{d(v, u) \mid u \in V\}$
  - graph centroid: $\arg\min_{v \in V} \sum_{u \in V} d(v, u)$
- in a tree: center is middle of longest path
- max. distance from center: $\left\lceil \frac{\text{diameter}}{2} \right\rceil$
- how do we connect tree centers?

Laning

- star with highest diameter tree in the middle!

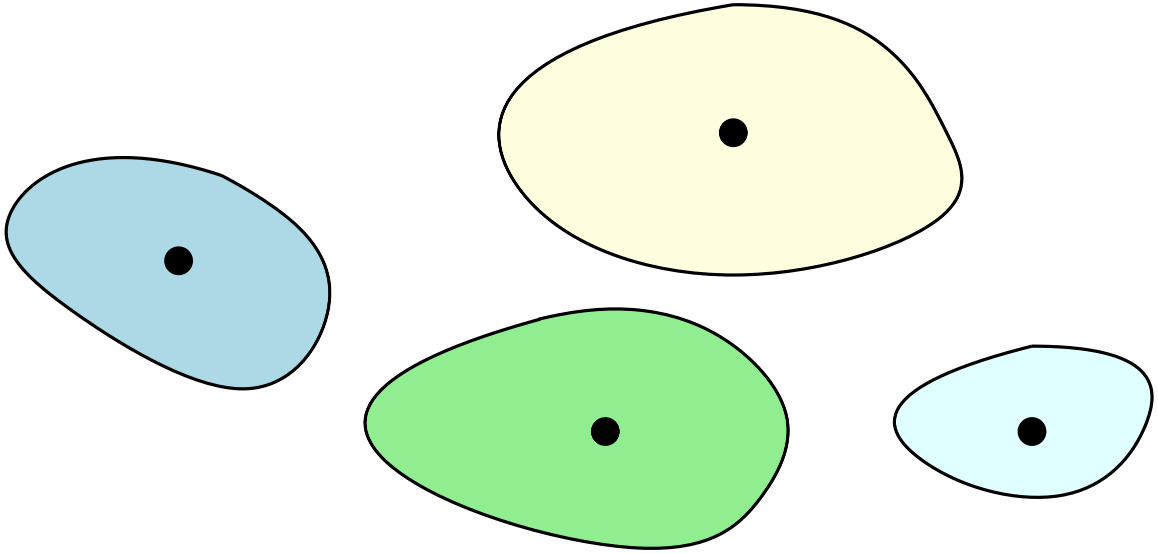
Laning

- star with highest diameter tree in the middle!



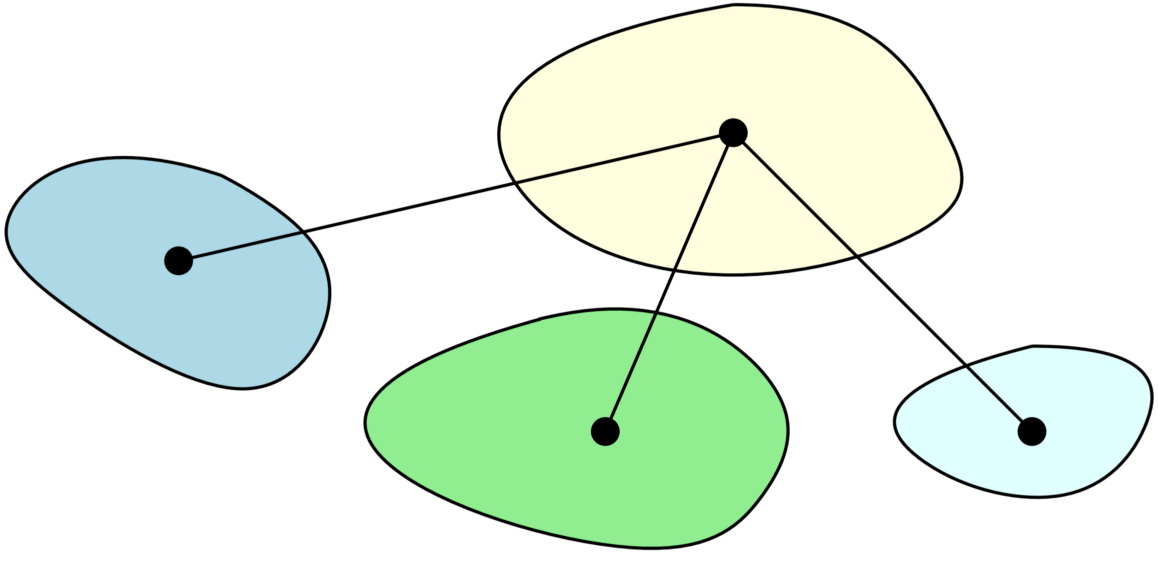
Laning

- star with highest diameter tree in the middle!



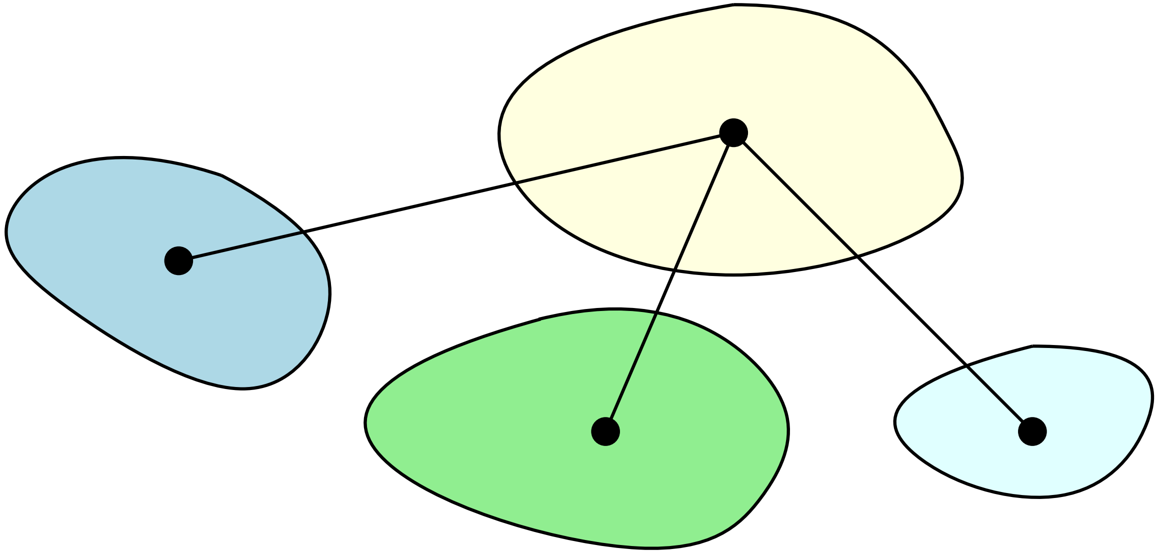
Laning

- star with highest diameter tree in the middle!



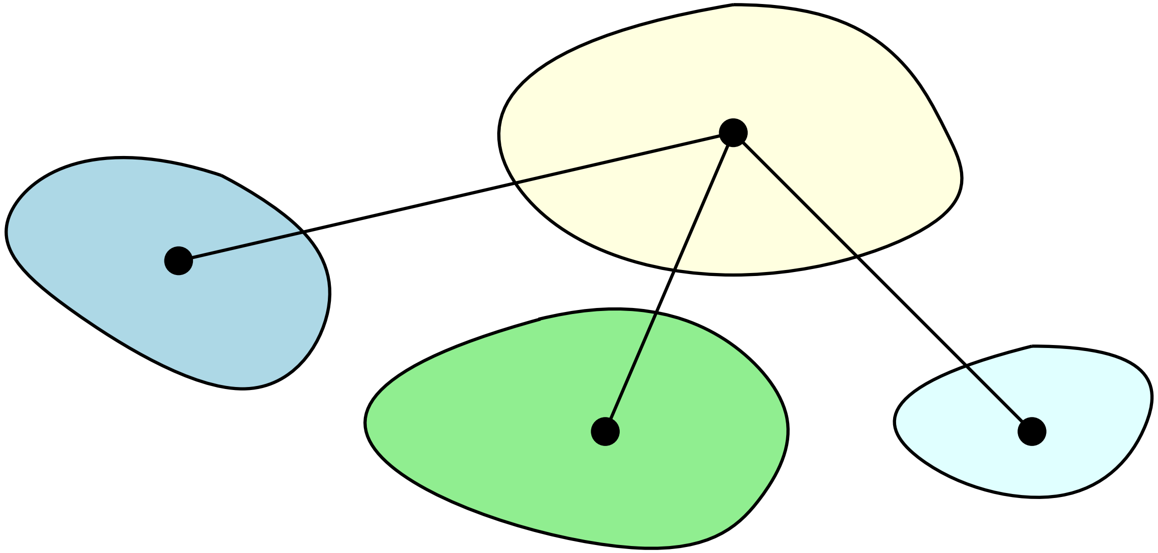
Laning

- star with highest diameter tree in the middle!
- let d_1, d_2, d_3 be the 3 highest diameters



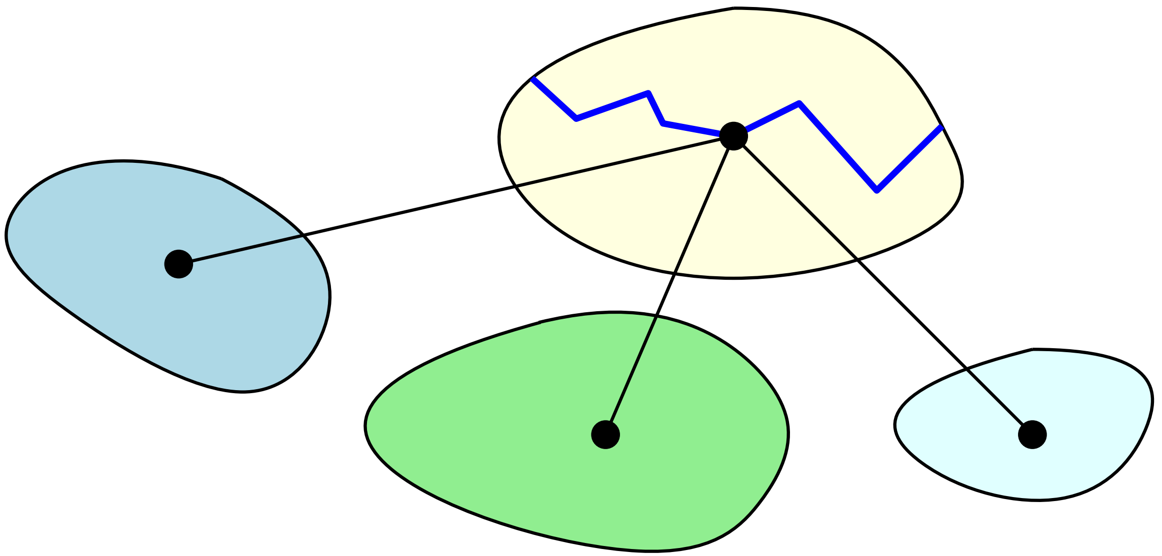
Laning

- star with highest diameter tree in the middle!
- let d_1, d_2, d_3 be the 3 highest diameters
- resulting diameter is the max of:



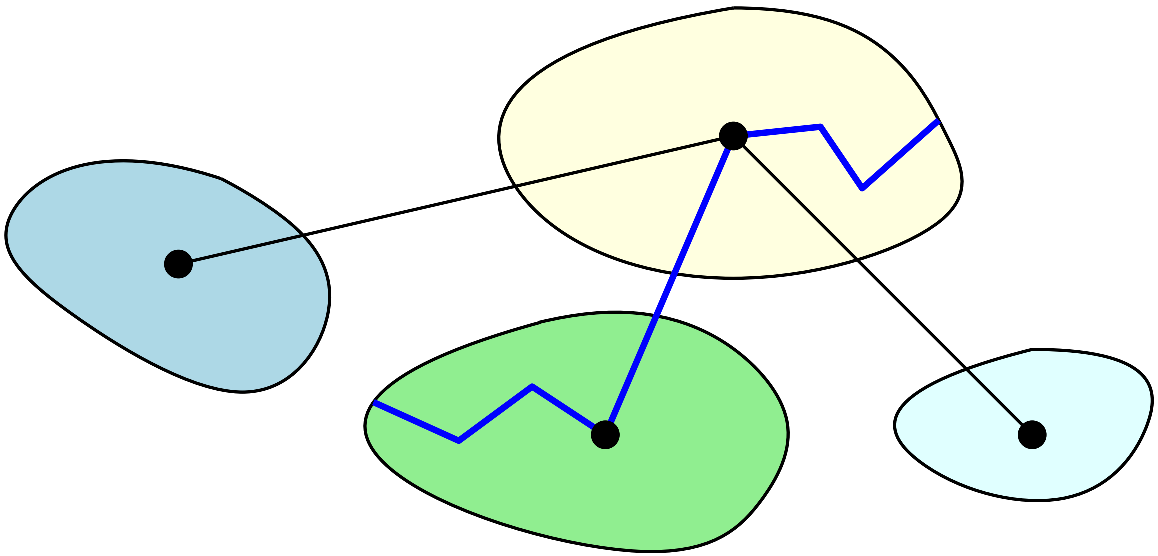
Laning

- star with highest diameter tree in the middle!
- let d_1, d_2, d_3 be the 3 highest diameters
- resulting diameter is the max of:
  - d_1



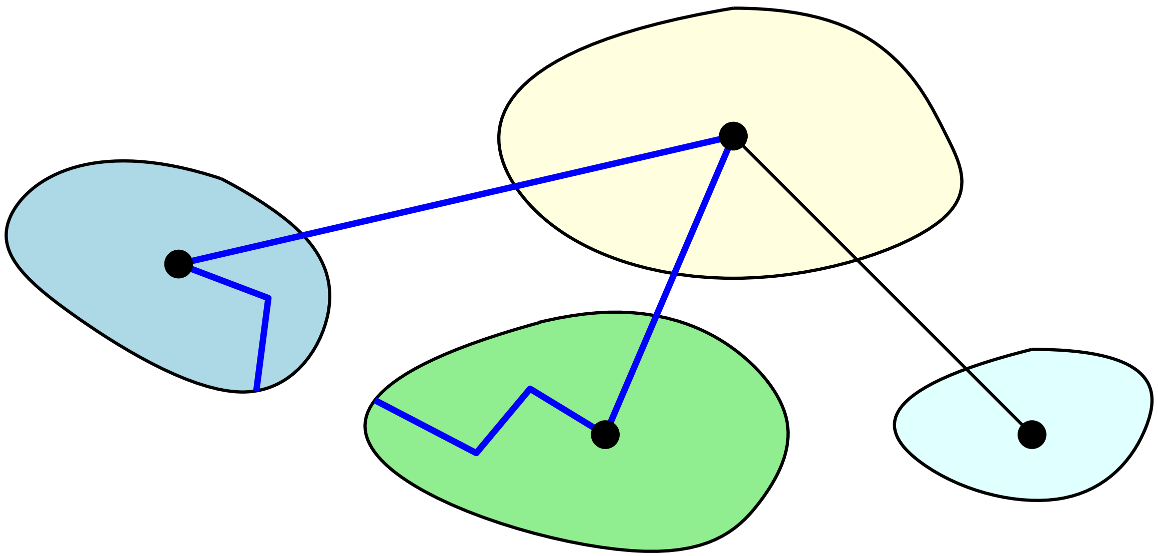
Laning

- star with highest diameter tree in the middle!
- let d_1, d_2, d_3 be the 3 highest diameters
- resulting diameter is the max of:
  - d_1
  - $\left\lceil \frac{d_1}{2} \right\rceil + 1 + \left\lceil \frac{d_2}{2} \right\rceil$



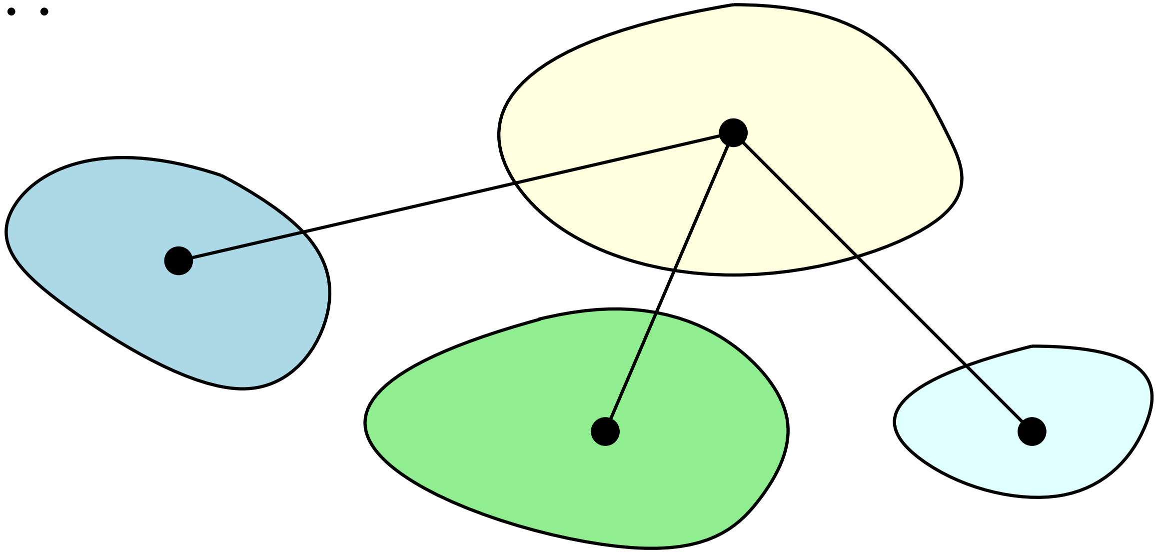
Laning

- star with highest diameter tree in the middle!
- let d_1, d_2, d_3 be the 3 highest diameters
- resulting diameter is the max of:
  - d_1
  - $\lceil \frac{d_1}{2} \rceil + 1 + \lceil \frac{d_2}{2} \rceil$
  - $\lceil \frac{d_2}{2} \rceil + 2 + \lceil \frac{d_3}{2} \rceil$



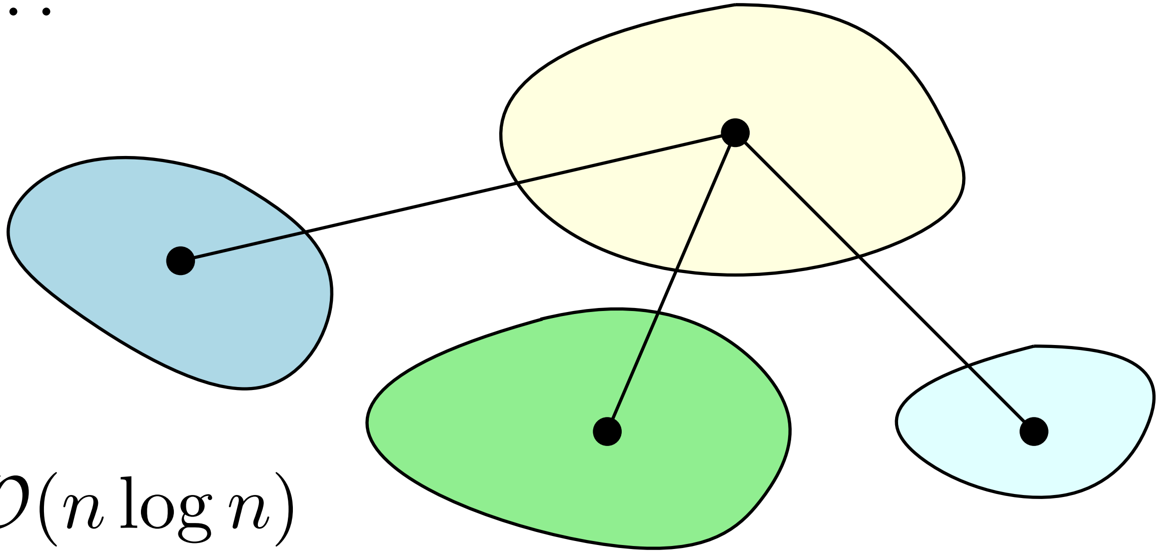
Laning

- star with highest diameter tree in the middle!
- let d_1, d_2, d_3 be the 3 highest diameters
- resulting diameter is the max of:
  - d_1
  - $\left\lceil \frac{d_1}{2} \right\rceil + 1 + \left\lceil \frac{d_2}{2} \right\rceil$
  - $\left\lceil \frac{d_2}{2} \right\rceil + 2 + \left\lceil \frac{d_3}{2} \right\rceil$
- alternative: build tree and calculate diameter...



Laning

- star with highest diameter tree in the middle!
- let d_1, d_2, d_3 be the 3 highest diameters
- resulting diameter is the max of:
  - d_1
  - $\lceil \frac{d_1}{2} \rceil + 1 + \lceil \frac{d_2}{2} \rceil$
  - $\lceil \frac{d_2}{2} \rceil + 2 + \lceil \frac{d_3}{2} \rceil$
- alternative: build tree and calculate diameter...



- runtime: $\mathcal{O}(n)$ or $\mathcal{O}(n \log n)$

Palindrom

Problem: convert a string s to a palindrome using the minimum number of operations, where each operation is a deletion, insertion, or change of a single character

Palindrom

Problem: convert a string s to a palindrome using the minimum number of operations, where each operation is a deletion, insertion, or change of a single character

- First insight: insertions are not needed

Palindrom

Problem: convert a string s to a palindrome using the minimum number of operations, where each operation is a deletion, insertion, or change of a single character

- First insight: insertions are not needed
- In the final palindrome, the inserted character is either

Palindrom

Problem: convert a string s to a palindrome using the minimum number of operations, where each operation is a deletion, insertion, or change of a single character

- First insight: insertions are not needed
- In the final palindrome, the inserted character is either
  - At the center $\rightarrow$ just don't insert it

Palindrom

Problem: convert a string s to a palindrome using the minimum number of operations, where each operation is a deletion, insertion, or change of a single character

- First insight: insertions are not needed
- In the final palindrome, the inserted character is either
  - At the center $\rightarrow$ just don't insert it
  - Equal to some other character on the other side of the string $\rightarrow$ delete that character instead

Palindrom

- Given a string $x \ldots y$, what's the minimal changes we can make to reduce to a smaller subproblem?

Palindrom

- Given a string $x \ldots y$, what's the minimal changes we can make to reduce to a smaller subproblem?
  - Delete x, convert remaining string (1 operation)

Palindrom

- Given a string $x \ldots y$, what's the minimal changes we can make to reduce to a smaller subproblem?
  - Delete x, convert remaining string (1 operation)
  - Delete y, convert remaining string (1 operation)

Palindrom

- Given a string $x \dots y$, what's the minimal changes we can make to reduce to a smaller subproblem?
  - Delete x, convert remaining string (1 operation)
  - Delete y, convert remaining string (1 operation)
  - Make x equal to y, convert remaining inner string
    - 1 operation, or 0, if they are already equal

Palindrom

- Given a string $x \dots y$, what's the minimal changes we can make to reduce to a smaller subproblem?
  - Delete x, convert remaining string (1 operation)
  - Delete y, convert remaining string (1 operation)
  - Make x equal to y, convert remaining inner string
    - 1 operation, or 0, if they are already equal
- Compute $\mathrm{cost}(l, r)$: minimum operations to convert substring of s from l to r to a palindrome

Palindrom

- Given a string $x \dots y$, what's the minimal changes we can make to reduce to a smaller subproblem?
  - Delete x, convert remaining string (1 operation)
  - Delete y, convert remaining string (1 operation)
  - Make x equal to y, convert remaining inner string
    - 1 operation, or 0, if they are already equal
- Compute $\text{cost}(l, r)$: minimum operations to convert substring of s from l to r to a palindrome
  - Use the above as transitions

Palindrom

- Given a string $x \dots y$, what's the minimal changes we can make to reduce to a smaller subproblem?
  - Delete x, convert remaining string (1 operation)
  - Delete y, convert remaining string (1 operation)
  - Make x equal to y, convert remaining inner string
    - 1 operation, or 0, if they are already equal
- Compute $\mathrm{cost}(l, r)$: minimum operations to convert substring of s from l to r to a palindrome
  - Use the above as transitions
- Answer: $\mathrm{cost}(1, n)$

Palindrom

- Prime examples of DPs being very compact to code

```
for w in range(2, n + 1):
    for l in range(n - w + 1):
        r = l + w
        dp[l][r] = min(dp[l+1][r] + 1, dp[l][r-1] + 1,
                       dp[l+1][r-1] + int(s[l]!=s[r-1]))
```

Palindrom

- Prime examples of DPs being very compact to code

```
for w in range(2, n + 1):
    for l in range(n - w + 1):
        r = l + w
        dp[l][r] = min(dp[l+1][r] + 1, dp[l][r-1] + 1,
                       dp[l+1][r-1] + int(s[l]!=s[r-1]))
```

- The structure of having one value per substring is extremely common for string DPs

Palindrom

- Prime examples of DPs being very compact to code

```
for w in range(2, n + 1):
    for l in range(n - w + 1):
        r = l + w
        dp[l][r] = min(dp[l+1][r] + 1, dp[l][r-1] + 1,
                       dp[l+1][r-1] + int(s[l]!=s[r-1]))
```

- The structure of having one value per substring is extremely common for string DPs

- Fun fact: the answer is equivalent to half the edit distance of the string to its own reverse