W04: Correctness and Divide & Conquer

CS3230 AY21/22 Sem 2

Table of Contents

Click on the link to jump to the relevant sections!

- Recap: Proving Correctness of Iterative Algorithms
- Question 1: Dijkstra's
  - Dijkstra's Algorithm walkthrough
  - Dijkstra's Correctness Proof:
    - Intuition & Formalisation
    - Maintenance Proof
- Question 2: 2D Peak-finding
- Question 3: Finding useful property for better 2D Peak-finding
- Question 4: Better 2D Peak-finding

Recap: Proving Correctness of Iterative Algorithms

First, you need to **define some invariants** (can be many and different things!)

Recap: Proving Correctness of Iterative Algorithms

First, you need to **define some invariants** (can be many and different things!). Choose some invariant which will be **useful** to show what we want **at the end of the loop**.

Recap: Proving Correctness of Iterative Algorithms

First, you need to **define some invariants** (can be many and different things!). Choose some invariant which will be **useful** to show what we want **at the end of the loop**.

Show:

- **Initialisation**: Invariant is true before the first iteration of the loop

Recap: Proving Correctness of Iterative Algorithms

First, you need to **define some invariants** (can be many and different things!). Choose some invariant which will be **useful** to show what we want **at the end of the loop**.

Show:

- **Initialisation**: Invariant is true before the first iteration of the loop
- **Maintenance**: If invariant true before an iteration, it remains true before the next iteration

Recap: Proving Correctness of Iterative Algorithms

First, you need to **define some invariants** (can be many and different things!). Choose some invariant which will be **useful** to show what we want **at the end of the loop**.

Show:

- **Initialisation**: Invariant is true before the first iteration of the loop
- **Maintenance**: If invariant true before an iteration, it remains true before the next iteration
- **Termination**: When the algorithm terminates, the invariant gives us a useful property for showing correctness

Question 1: Dijkstra’s Correctness

Dijkstra's Algorithm

G = (V, E) is an undirected graph. With *s* as the start node. **All edges in G are positive weights**.

```
def Dijkstra(G, s):
For all node u except s, d(u) = inf
d(s) = 0; R = {};
while R != V:
  pick u not in R, with the smallest d(u)
  R = R union {u}
  for v in neighbours(u):
    d(v) = min(d(v), d(u) + w(u, v))
```

Dijkstra's Algorithm

G = (V, E) is an undirected graph. With *s* as the start node. **All edges in G are positive weights**.

```
def Dijkstra(G, s):
For all node u except s, d(u) = inf
d(s) = 0; R = {};
while R != V:
  pick u not in R, with the smallest d(u)
  R = R union {u}
  for v in neighbours(u):
    d(v) = min(d(v), d(u) + w(u, v))
```

Basic idea:

- Start with an empty 'distance to everyone' -- I have no idea what's the distance to everyone - except myself!

Dijkstra's Algorithm

G = (V, E) is an undirected graph. With *s* as the start node. **All edges in G are positive weights**.

```
def Dijkstra(G, s):
1. For all node u except s, d(u) = inf
2. d(s) = 0; R = {};
3. while R != V:
4.   pick u not in R, with the smallest d(u)
5.   R = R union {u}
6.   for v in neighbours(u):
7.     d(v) = min(d(v), d(u) + w(u, v))
```

Also known as: Relaxation Step

Basic idea:

- Start with an empty 'distance to everyone' -- I have no idea what's the distance to everyone - except myself!
- Use *d(u)* as an *estimate* -- "Right now, I can get to *d(u)* with this much cost. Can it be better?"
- Repeat:
  - Consider the vertex with *minimum* estimate
  - Add it to our R (results, what we are sure of the distance)
  - Relax all outgoing edges -- maybe we can go to other nodes faster!

Dijkstra's Algorithm

G = (V, E) is an undirected graph. With *s* as the start node. **All edges in G are positive weights**.

```
def Dijkstra(G, s):

1. For all node u except s, d(u) = inf

2. d(s) = 0; R = {};

3. while R != V:

4.   pick u not in R, with the smallest d(u)

5.   R = R union {u}

6.   for v in neighbours(u):

7.     d(v) = min(d(v), d(u) + w(u, v))
```

Basic idea:

- Start with an empty 'distance to everyone' -- I have no idea what's the distance to everyone - except myself!
- Use *d(u)* as an *estimate* -- "Right now, I can get to *d(u)* with this much cost. Can it be better?"
- Repeat:
    - Consider the vertex with *minimum* estimate

Dijkstra's Algorithm

G = (V, E) is an undirected graph. With *s* as the start node. **All edges in G are positive weights**.

```
def Dijkstra(G, s):
For all node u except s, d(u) = inf
d(s) = 0; R = {};
while R != V:
  pick u not in R, with the smallest d(u)
  R = R union {u}
  for v in neighbours(u):
    d(v) = min(d(v), d(u) + w(u, v))
```

Basic idea:

- Start with an empty 'distance to everyone' -- I have no idea what's the distance to everyone - except myself!
- Use *d(u)* as an *estimate* -- "Right now, I can get to *d(u)* with this much cost. Can it be better?"
- Repeat:
  - Consider the vertex with *minimum* estimate
  - Add it to our R (results, what we are sure of the distance)

Dijkstra's Algorithm

G = (V, E) is an undirected graph. With *s* as the start node. **All edges in G are positive weights**.

```
def Dijkstra(G, s):
For all node u except s, d(u) = inf
d(s) = 0; R = {};
while R != V:
  pick u not in R, with the smallest d(u)
  R = R union {u}
  for v in neighbours(u):
    d(v) = min(d(v), d(u) + w(u, v))
```

Also known as: Relaxation Step

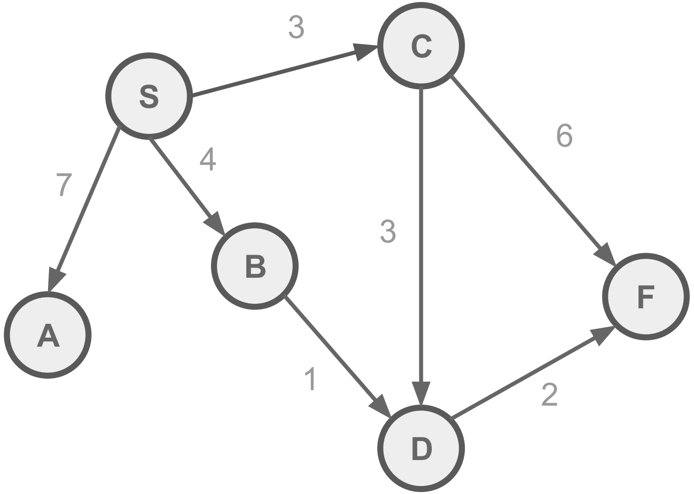
Basic idea:

- Start with an empty 'distance to everyone' -- I have no idea what's the distance to everyone - except myself!
- Use *d(u)* as an *estimate* -- "Right now, I can get to *d(u)* with this much cost. Can it be better?"
- Repeat:
  - Consider the vertex with *minimum* estimate
  - Add it to our R (results, what we are sure of the distance)
  - Relax all outgoing edges -- maybe we can go to other nodes faster!

Question 1

Walkthrough

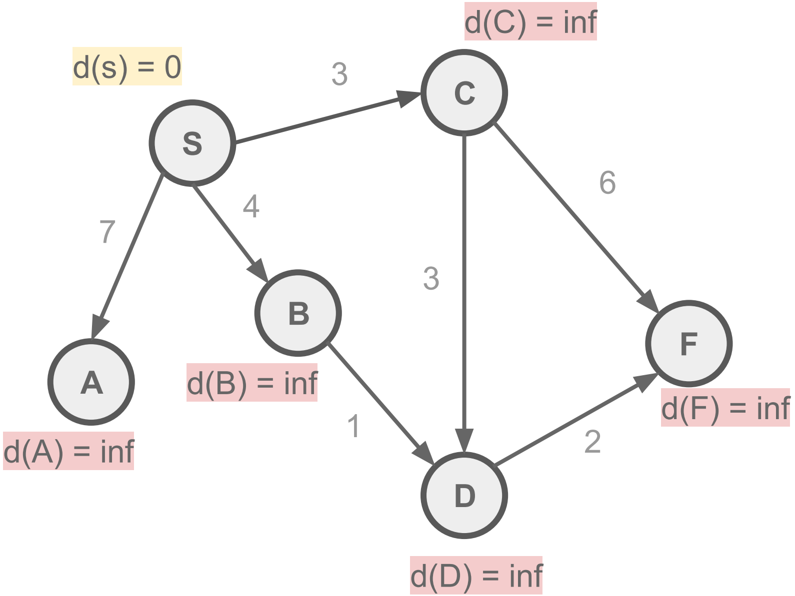
```
def Dijkstra(G, s):
For all node u except s, d(u) = inf
d(s) = 0; R = {};
while R != V:
  pick u not in R, with the smallest d(u)
  R = R union {u}
  for v in neighbours(u):
    d(v) = min(d(v), d(u) + w(u, v))
```



Question 1

Initialisation of d(u)

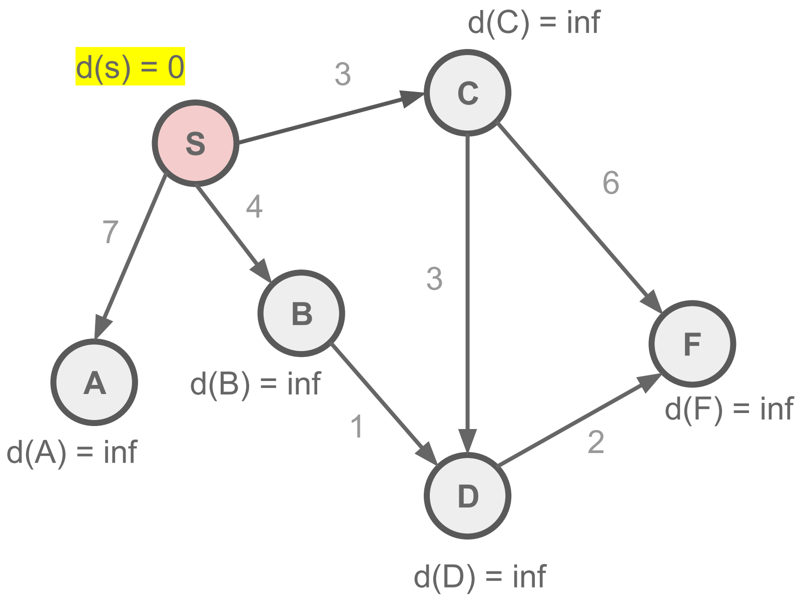
```
def Dijkstra(G, s):
1. For all node u except s, d(u) = inf
2. d(s) = 0; R = {};
3. while R != V:
4.   pick u not in R, with the smallest d(u)
5.   R = R union {u}
6.   for v in neighbours(u):
7.     d(v) = min(d(v), d(u) + w(u, v))
```



Question 1

From this slide onwards, R = set of red nodes

```
def Dijkstra(G, s):
1. For all node u except s, d(u) = inf
2. d(s) = 0; R = {};
3. while R != V:
4.    pick u not in R, with the smallest d(u)
5.    R = R union {u}
6.    for v in neighbours(u):
7.      d(v) = min(d(v), d(u) + w(u, v))
```



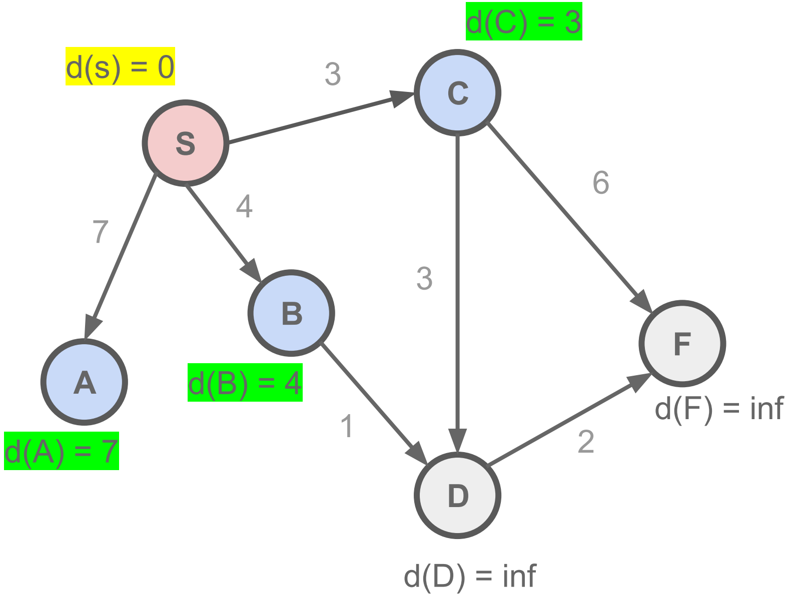
Question 1

Relax all the neighbours!

```
def Dijkstra(G, s):
1. For all node u except s, d(u) = inf
2. d(s) = 0; R = {};
3. while R != V:
4.   pick u not in R, with the smallest d(u)
5.   R = R union {u}
6.   for v in neighbours(u):
7.     d(v) = min(d(v), d(u) + w(u, v))
```

Existing estimate is good

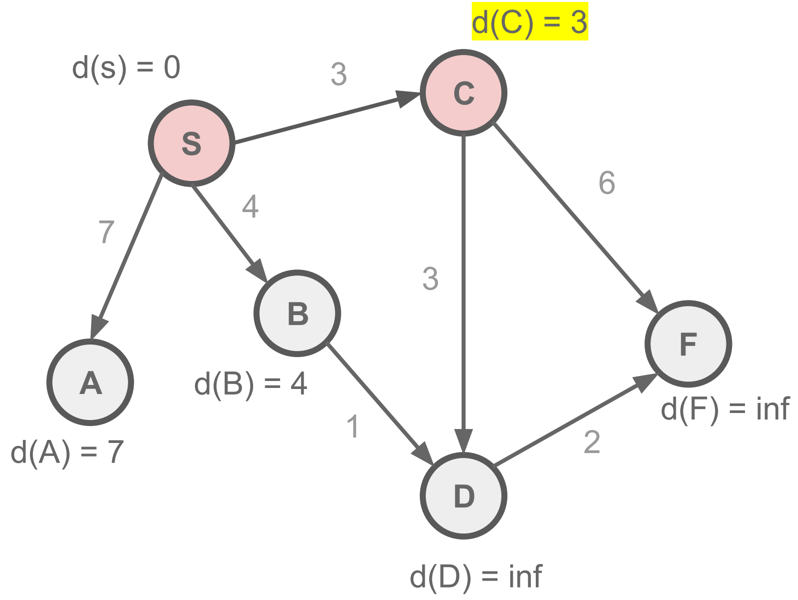
Can do better by going through node u



Question 1

Repeat the process!

```
def Dijkstra(G, s):
1. For all node u except s, d(u) = inf
2. d(s) = 0; R = {};
3. while R != V:
4.    pick u not in R, with the smallest d(u)
5.    R = R union {u}
6.    for v in neighbours(u):
7.      d(v) = min(d(v), d(u) + w(u, v))
```



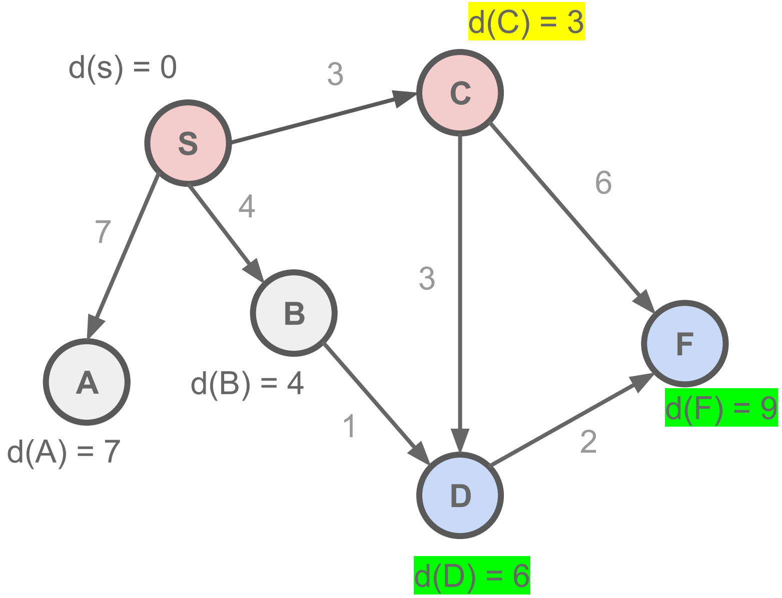
Question 1

Relax neighbours of C - now note that we *think* we can get to D and F with cost 6 and 9 respectively

```
def Dijkstra(G, s):
1. For all node u except s, d(u) = inf
2. d(s) = 0; R = {};
3. while R != V:
4.   pick u not in R, with the smallest d(u)
5.   R = R union {u}
6.   for v in neighbours(u):
7.     d(v) = min(d(v), d(u) + w(u, v))
```

Existing estimate is good

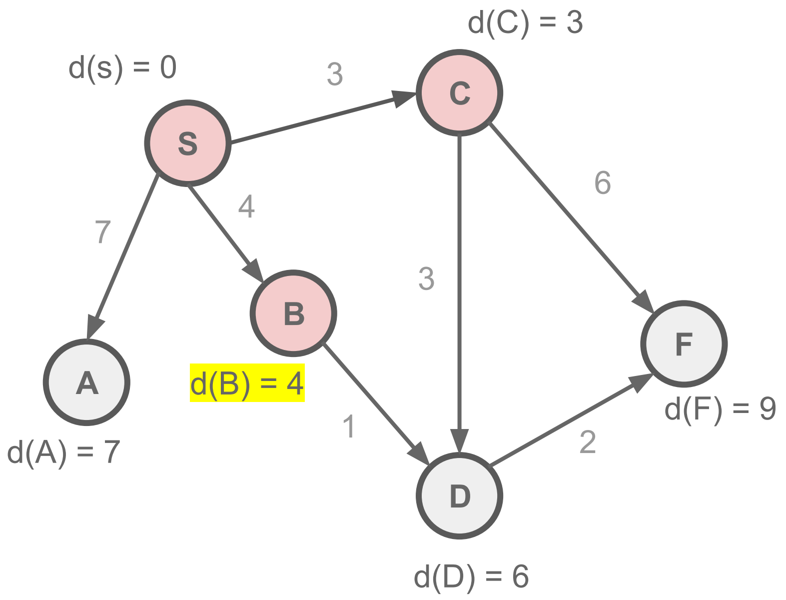
Can do better by going through node u



Question 1

Repeat the process!

```
def Dijkstra(G, s):
1. For all node u except s, d(u) = inf
2. d(s) = 0; R = {};
3. while R != V:
4.    pick u not in R, with the smallest d(u)
5.    R = R union {u}
6.    for v in neighbours(u):
7.      d(v) = min(d(v), d(u) + w(u, v))
```



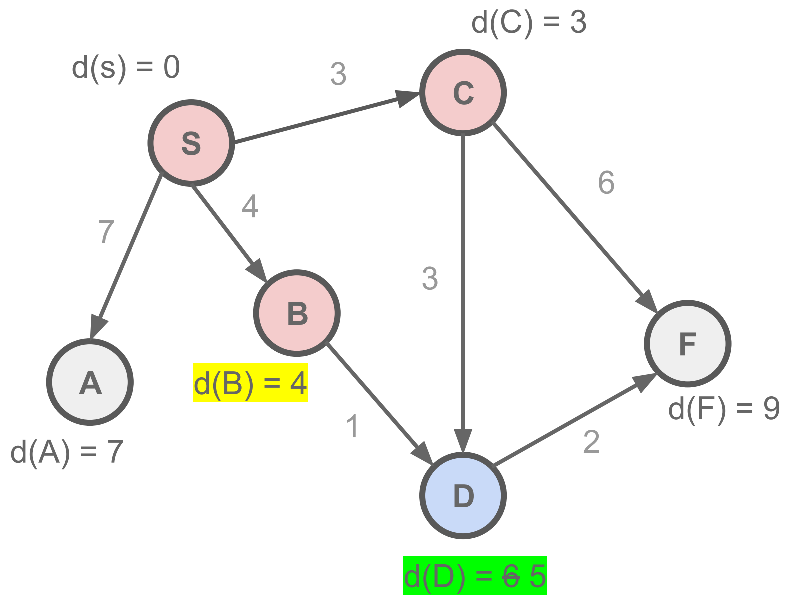
Question 1

IMPORTANT: Now we have a **better** estimate to reach node D. Instead of going through C (prev estimate)

```
def Dijkstra(G, s):

1. For all node u except s, d(u) = inf

2. d(s) = 0; R = {};

3. while R != V:

4.   pick u not in R, with the smallest d(u)

5.   R = R union {u}

6.   for v in neighbours(u):

7.     d(v) = min(d(v), d(u) + w(u, v))
```

Existing estimate is good

Can do better by going through node u



Question 1

Dijkstra(G, s)

1. *For all* $u \in V \setminus \{s\}$, $d(u)=\infty$;
2. $d(s)=0$; $R=\{\}$;
3. *While* $R \neq V$
4.   *pick* $u \notin R$ *with the smallest* $d(u)$
5. $R = R \cup \{u\}$
6.   *for all neighbor* v *of* u,
7. $d(v) = \min\{ d(v), d(u)+w(u,v))\}$

G=(V,E) is an undirected graph.
s is the start node
Assume all edges in G are of positive weights.

What is the invariant for the while loop?

Can you show that this algorithm correctly compute the shortest distance from s to all nodes?

Dijkstra’s Correctness Proof (Intuition)

What happens when the algorithm terminates?

```
def Dijkstra(G, s):
1. For all node u except s, d(u) = inf
2. d(s) = 0; R = {};
3. while R != V:
4.   pick u not in R, with the smallest d(u)
5.   R = R union {u}
6.   for v in neighbours(u):
7.     d(v) = min(d(v), d(u) + w(u, v))
```

Dijkstra’s Correctness Proof (Intuition)

What happens when the algorithm terminates?

- We should have all *d(u)* correctly updated as our shortest path cost!

```
def Dijkstra(G, s):

For all node u except s, d(u) = inf

d(s) = 0; R = {};

while R != V:

  pick u not in R, with the smallest d(u)

  R = R union {u}

  for v in neighbours(u):

    d(v) = min(d(v), d(u) + w(u, v))
```

Dijkstra's Correctness Proof (Intuition)

What happens when the algorithm terminates?

- We should have all *d(u)* correctly updated as our shortest path cost!

Why did the algorithm terminate?

```
def Dijkstra(G, s):

For all node u except s, d(u) = inf

d(s) = 0; R = {};

while R != V:

  pick u not in R, with the smallest d(u)

  R = R union {u}

  for v in neighbours(u):

    d(v) = min(d(v), d(u) + w(u, v))
```

Dijkstra's Correctness Proof (Intuition)

What happens when the algorithm terminates?

- We should have all *d(u)* correctly updated as our shortest path cost!

Why did the algorithm terminate?

- Because we have placed all the nodes in R

```
def Dijkstra(G, s):

For all node u except s, d(u) = inf

d(s) = 0; R = {};

while R != V:

  pick u not in R, with the smallest d(u)

  R = R union {u}

  for v in neighbours(u):

    d(v) = min(d(v), d(u) + w(u, v))
```

Dijkstra’s Correctness Proof (Intuition)

What happens when the algorithm terminates?

- We should have all *d(u)* correctly updated as our shortest path cost!

Why did the algorithm terminate?

- Because we have placed all the nodes in R
- Every iteration adds node u to R

```
def Dijkstra(G, s):

For all node u except s, d(u) = inf

d(s) = 0; R = {};

while R != V:

  pick u not in R, with the smallest d(u)

  R = R union {u}

  for v in neighbours(u):

    d(v) = min(d(v), d(u) + w(u, v))
```

Dijkstra's Correctness Proof (Intuition)

What happens when the algorithm terminates?

- We should have all *d(u)* correctly updated as our shortest path cost!

Why did the algorithm terminate?

- Because we have placed all the nodes in R
- Every iteration adds node u to R

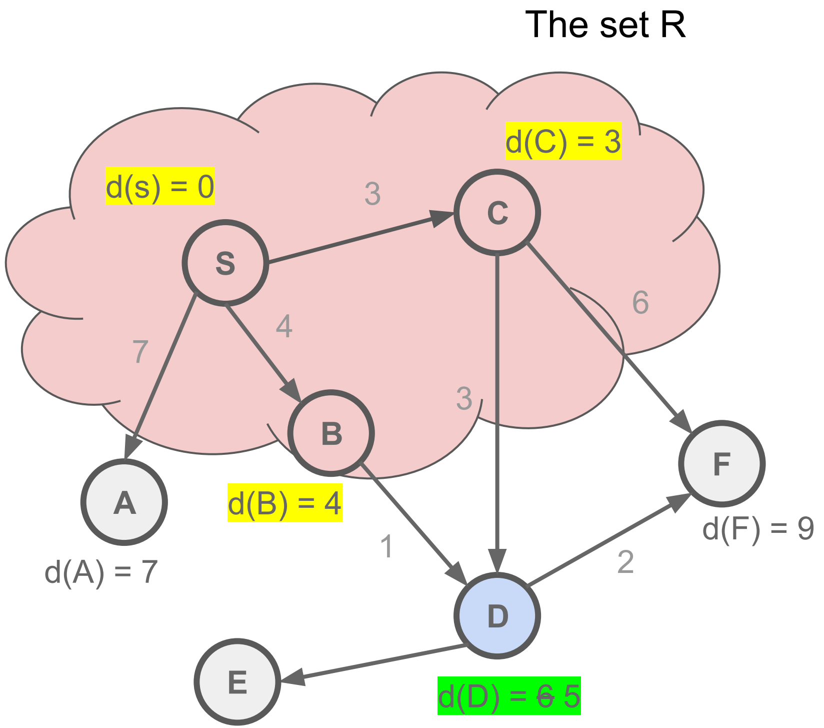
Invariant idea:

Relate the nodes already in R with "already have the correct shortest path cost"

```
def Dijkstra(G, s):
1. For all node u except s, d(u) = inf
2. d(s) = 0; R = {};
3. while R != V:
4.   pick u not in R, with the smallest d(u)
5.   R = R union {u}
6.   for v in neighbours(u):
7.     d(v) = min(d(v), d(u) + w(u, v))
```

Dijkstra’s Correctness Proof (Intuition)

Notice that all the nodes in set R would already have their true shortest path cost set!

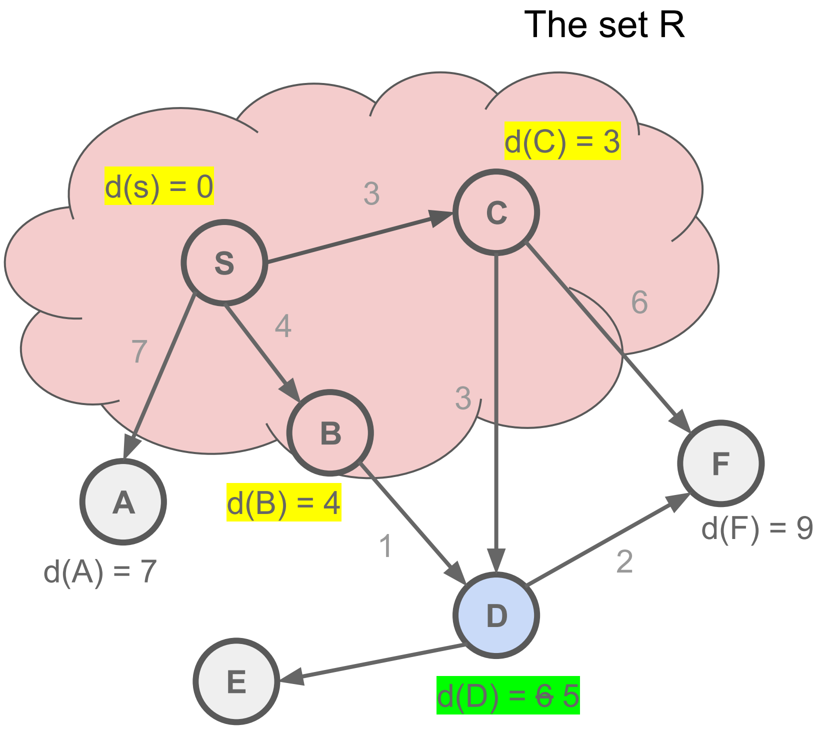


Dijkstra’s Correctness Proof (Intuition)

Notice that all the nodes in set R would already have their true shortest path cost set!

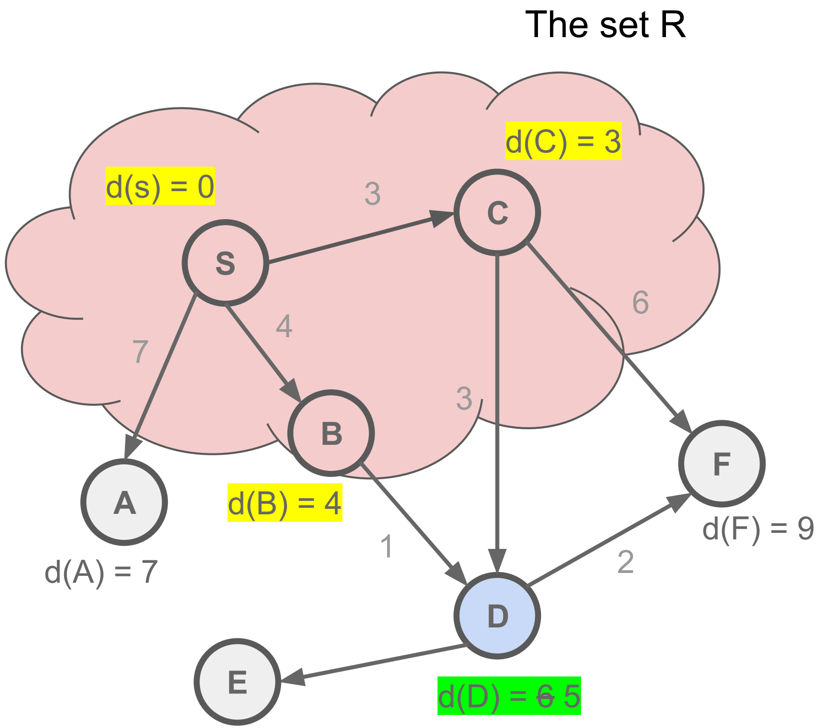
Furthermore, recall that in the algorithm, we simply add the next node with the minimum estimate

```
while R != V:
    pick u not in R, with the smallest d(u)
    R = R union {u}
```



Dijkstra’s Correctness Proof (Formalisation)

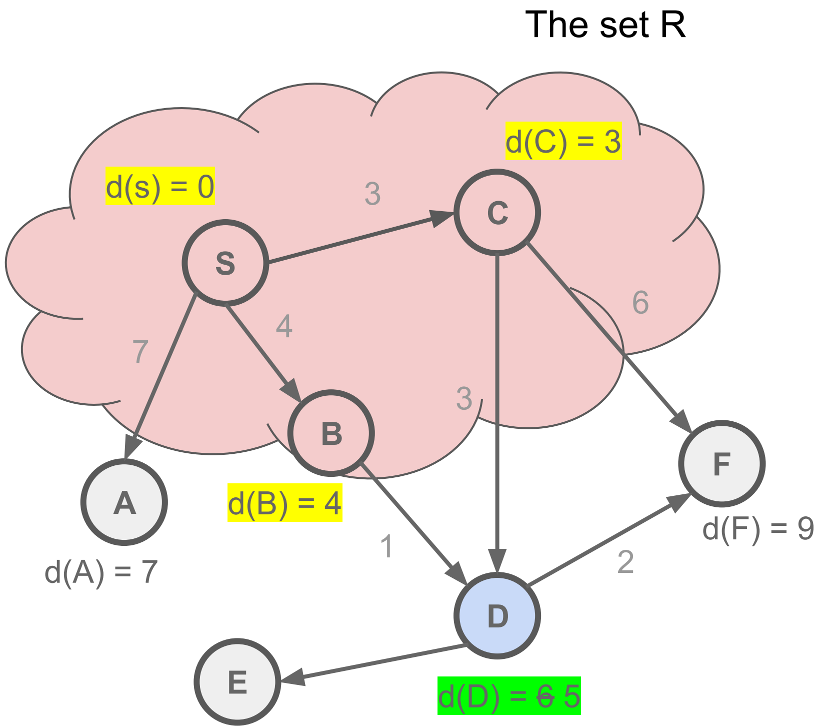
Let ***δ(a → b)*** be the (true) shortest distance from a to b in G



Dijkstra's Correctness Proof (Formalisation)

Let $\delta(\boldsymbol{a} \rightarrow \boldsymbol{b})$ be the (true) shortest distance from a to b in G

Invariant 1 (INV1):
For all node u in R, d(u) = $\delta(s \rightarrow u)$ [d(u) is the shortest distance from s to u already]

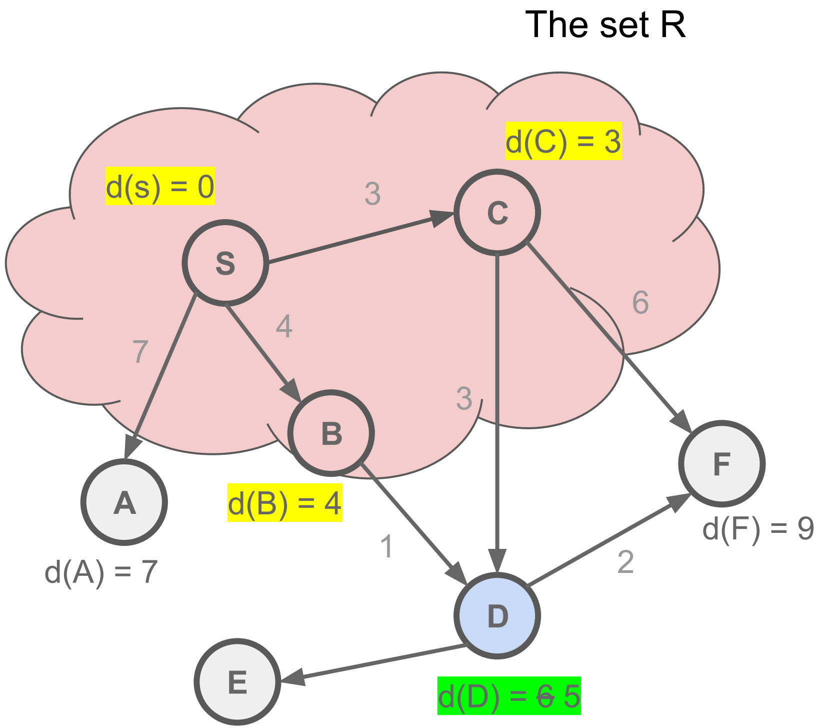


Dijkstra's Correctness Proof (Formalisation)

Let ***δ(a → b)*** be the (true) shortest distance from a to b in G

Invariant 1 (INV1):
For all node u in R, d(u) = *δ(s → u)* [d(u) is the shortest distance from s to u already]

Let ***Δ(X, u)*** be *min{ δ(s → v) + w(v, u) for v in X }*



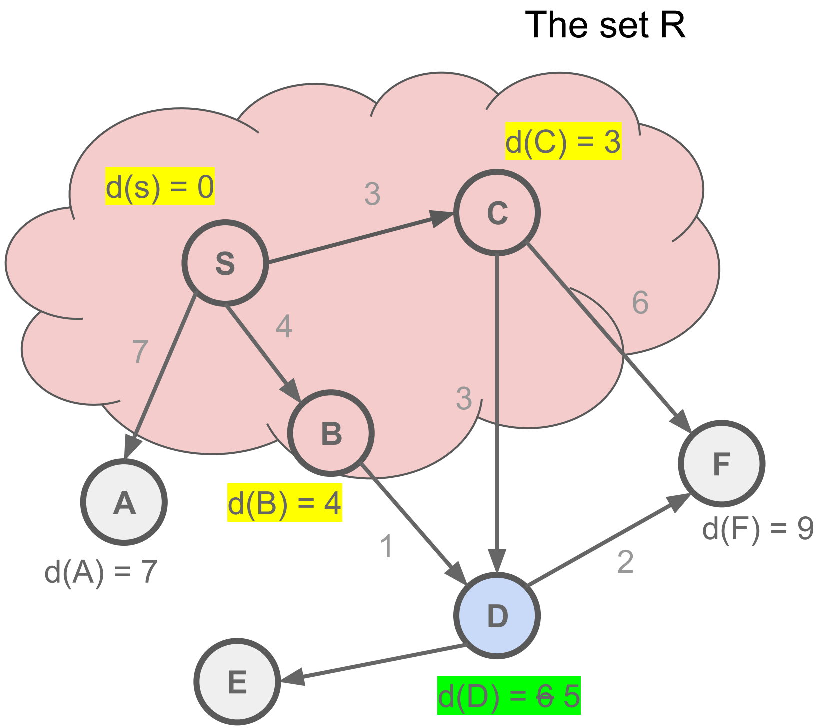
Dijkstra’s Correctness Proof (Formalisation)

Let ***δ(a → b)*** be the (true) shortest distance from a to b in G

Invariant 1 (INV1):
For all node u in R, d(u) = *δ(s → u)* [d(u) is the shortest distance from s to u already]

Let ***Δ(X, u)*** be *min{ δ(s → v) + w(v, u) for v in X }*

Read it as: “from all the nodes in X to me, take the ***best estimate***”. e.g. node D on the right can either have a distance 6 (through C) or 5 (through B)



Dijkstra’s Correctness Proof (Formalisation)

Let ***δ(a → b)*** be the (true) shortest distance from a to b in G

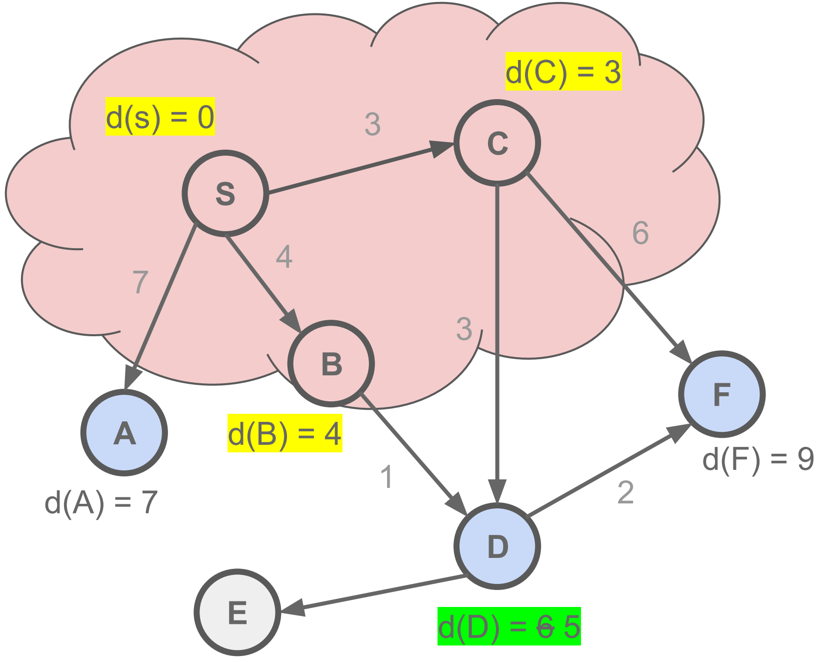
Invariant 1 (INV1):
For all node u in R, d(u) = *δ(s → u)* [d(u) is the shortest distance from s to u already]

Let ***Δ(X, u)*** be *min{ δ(s → v) + w(v, u) for v in X }*

Read it as: “from all the nodes in X to me, take the ***best estimate***”. e.g. node D on the right can either have a distance 6 (through C) or 5 (through B)

Invariant 2 (INV2):
For every neighbour v of all nodes in R,
d(v) = Δ(R, v)

The set R



e.g. E is not a neighbour of node in R

Dijkstra **Initialisation**

Before the first iteration of the while loop, we set $R = \{\}$.

INV1 and INV2 are trivially true (there is no node in R)

Let $\delta(a \rightarrow b)$ be the (true) shortest distance from a to b in G

Invariant 1 (INV1):
For all node u in R, d(u) = $\delta(s \rightarrow u)$ [d(u) is the shortest distance from s to u already]

Let $\Delta(X, u)$ be $min\{ \delta(s \rightarrow v) + w(v, u)$ *for v in X* $\}$

Invariant 2 (INV2):
For every neighbour v of all nodes in R, $d(v) = \Delta(R, v)$

```
1. For all node u except s, d(u) = inf
2. d(s) = 0; R = {};
3. while R != V:
```

Dijkstra **Maintenance** (INV1)

In steps 4 and 5, we add the node u with smallest d(u) into R

To maintain INV1, we need to show that this node u that we pick has d(u) = $\delta(s \rightarrow u)$ [Recall what it means to be inside R]

Let $\delta(a \rightarrow b)$ be the (true) shortest distance from a to b in G

Invariant 1 (INV1):
For all node u in R, d(u) = $\delta(s \rightarrow u)$ [d(u) is the shortest distance from s to u already]

Let $\Delta(X, u)$ be $\min\{ \delta(s \rightarrow v) + w(v, u)$ for v in $X \}$

Invariant 2 (INV2):
For every neighbour v of all nodes in R, $d(v) = \Delta(R, v)$

```
4.   pick u not in R, with the smallest d(u)

5.   R = R union {u}
```

Dijkstra **Maintenance** (INV1)

In steps 4 and 5, we add the node u with smallest d(u) into R

To maintain INV1, we need to show that this node u that we pick has d(u) = $\delta(s \to u)$ [Recall what it means to be inside R]

Proof by Contradiction! Assume that the node u **does not have** d(u) = $\delta(s \to u)$

Let ***δ(a → b)*** be the (true) shortest distance from a to b in G

Invariant 1 (INV1):
For all node u in R, d(u) = *$\delta(s \to u)$* [d(u) is the shortest distance from s to u already]

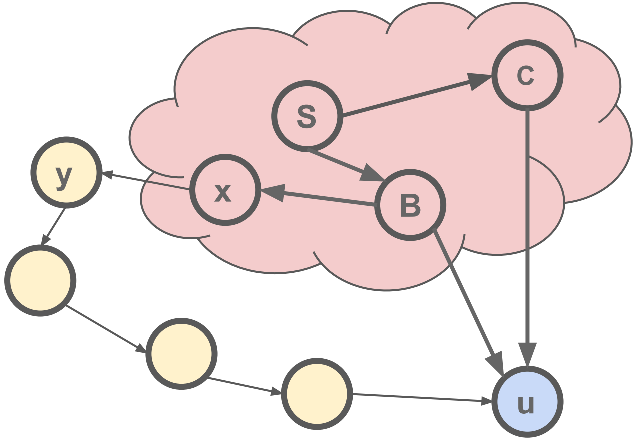
Let ***Δ(X, u)*** be *min{ $\delta(s \to v) + w(v, u)$ for v in X }*

Invariant 2 (INV2):
For every neighbour v of all nodes in R, *d(v) = Δ(R, v)*

```
4.   pick u not in R, with the smallest d(u)

5.   R = R union {u}
```

Proof by Contradiction! Assume that the node u **does not have** d(u) = δ (s → u). There is some other path from S to reach u that is shorter (via edge x → y). Call this path Q.

The set R



Let δ***(a → b)*** be the (true) shortest distance from a to b in G

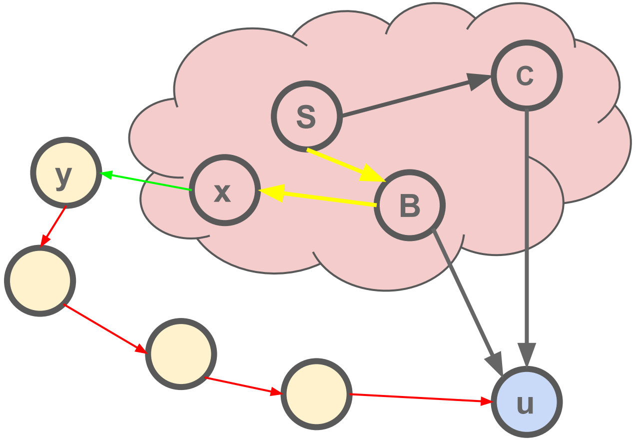
Let ***Δ(X, u)*** be *min{ δ(s → v) + w(v, u) for v in X }*

Invariant 2 (INV2):
For every neighbour v of all nodes in R,
d(v) = Δ(R, v)

Proof by Contradiction! Assume that the node u **does not have** d(u) = δ (s $\rightarrow$ u). There is some other path from S to reach u that is shorter (via edge x $\rightarrow$ y). Call this path Q.

Weight of Q: $\delta(s \rightarrow x)$ + $w(x, y)$ + $\delta(y \rightarrow u)$ < $d(u)$ [by assumption]

The set R



Let $\boldsymbol{\delta(a \rightarrow b)}$ be the (true) shortest distance from a to b in G

Let $\boldsymbol{\Delta(X, u)}$ be $min\{\ \delta(s \rightarrow v) + w(v, u) \text{ for } v \text{ in } X\ \}$

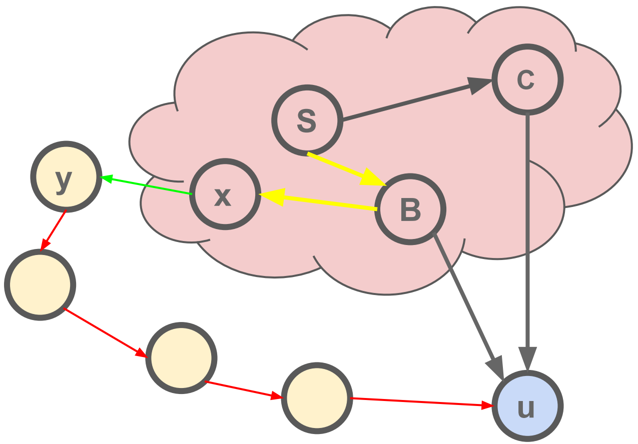
Invariant 2 (INV2):
For every neighbour v of all nodes in R,
$d(v) = \Delta(R, v)$

Proof by Contradiction! Assume that the node u **does not have** d(u) = δ (s → u). There is some other path from S to reach u that is shorter (via edge x → y). Call this path Q.

Weight of Q: $\delta(s \to x) + w(x, y) + \delta(y \to u) < d(u)$ [by assumption]

$d(y) = \Delta(R, y) = \min\{ \delta(s \to r) + w(r, y) \text{ for } r \text{ in } R \}$ [by invariant 2]

The set R



Let δ***(a → b)*** be the (true) shortest distance from a to b in G

Let ***Δ(X, u)*** be $\min\{ \delta(s \to v) + w(v, u) \text{ for } v \text{ in } X \}$

Invariant 2 (INV2):
For every neighbour v of all nodes in R,
$d(v) = \Delta(R, v)$

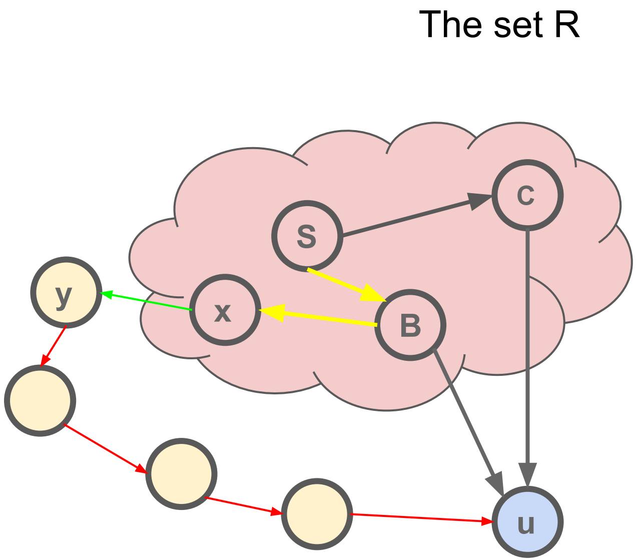
Proof by Contradiction! Assume that the node u **does not have** d(u) = $\delta(s \rightarrow u)$. There is some other path from S to reach u that is shorter (via edge $x \rightarrow y$). Call this path Q.

Weight of Q: $\delta(s \rightarrow x) + w(x, y) + \delta(y \rightarrow u) < d(u)$ [by assumption]

$d(y) = \Delta(R, y) = \min\{ \delta(s \rightarrow r) + w(r, y) \text{ for } r \text{ in } R \}$ [by invariant 2]

$d(y) \leq \delta(s \rightarrow x) + w(x, y)$ [d(y) is the min, so anything else must be ≤]

e.g. d(y) = min {100, 3, 50, 42 }
so d(y) ≤ 100 or d(y) ≤ 3 or d(y) ≤ 50 or d(y) ≤ 42



Let $\delta(a \rightarrow b)$ be the (true) shortest distance from a to b in G

Let $\Delta(X, u)$ be $\min\{ \delta(s \rightarrow v) + w(v, u) \text{ for } v \text{ in } X \}$

Invariant 2 (INV2):
For every neighbour v of all nodes in R,
$d(v) = \Delta(R, v)$

Proof by Contradiction! Assume that the node u **does not have** $d(u) = \delta(s \rightarrow u)$. There is some other path from S to reach u that is shorter (via edge $x \rightarrow y$). Call this path Q.

Weight of Q: $\delta(s \rightarrow x) + w(x, y) + \delta(y \rightarrow u) < d(u)$ [by assumption]

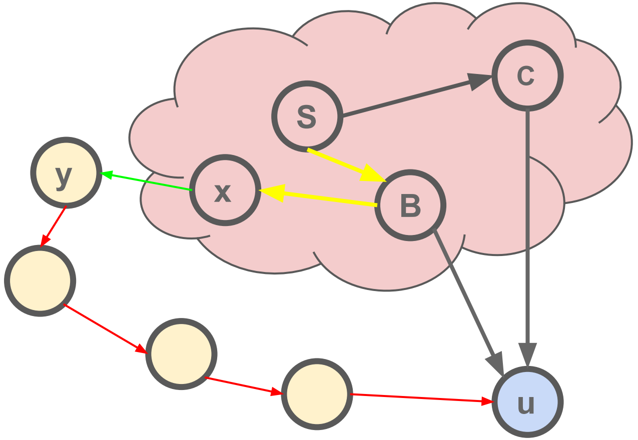
$d(y) = \Delta(R, y) = \min\{ \delta(s \rightarrow r) + w(r, y) \text{ for } r \text{ in } R \}$ [by invariant 2]

$d(y) \leq \delta(s \rightarrow x) + w(x, y)$ [d(y) is the min, so anything else must be ≤]

$\leq \delta(s \rightarrow x) + w(x, y) + \delta(y \rightarrow u)$ [edges are positive. Just add them in]

It's like saying 100 ≤ 100 + 3

The set R



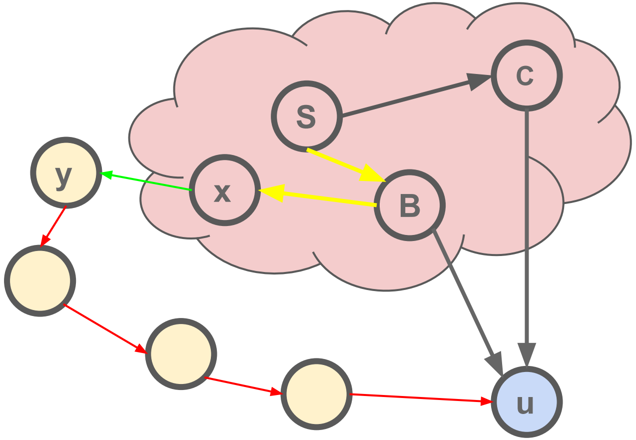
Let $\delta(a \rightarrow b)$ be the (true) shortest distance from a to b in G

Let $\Delta(X, u)$ be $\min\{ \delta(s \rightarrow v) + w(v, u) \text{ for } v \text{ in } X \}$

Invariant 2 (INV2):
For every neighbour v of all nodes in R,
$d(v) = \Delta(R, v)$

Proof by Contradiction! Assume that the node u **does not have** $d(u) = \delta(s \rightarrow u)$. There is some other path from S to reach u that is shorter (via edge $x \rightarrow y$). Call this path Q.

The set R



Weight of Q: $\delta(s \rightarrow x) + w(x, y) + \delta(y \rightarrow u) < d(u)$ [by assumption]

$d(y) = \Delta(R, y) = \min\{ \delta(s \rightarrow r) + w(r, y) \text{ for } r \text{ in } R \}$ [by invariant 2]

$d(y) \leq \delta(s \rightarrow x) + w(x, y)$ [d(y) is the min, so anything else must be ≤]

$\leq \delta(s \rightarrow x) + w(x, y) + \delta(y \rightarrow u)$ [edges are positive. Just add them in]

$< d(u)$

Let $\delta(a \rightarrow b)$ be the (true) shortest distance from a to b in G

Let $\Delta(X, u)$ be $\min\{ \delta(s \rightarrow v) + w(v, u) \text{ for } v \text{ in } X \}$

Invariant 2 (INV2):
For every neighbour v of all nodes in R,
$d(v) = \Delta(R, v)$

Proof by Contradiction! Assume that the node u **does not have** d(u) = $\delta(s \rightarrow u)$. There is some other path from S to reach u that is shorter (via edge $x \rightarrow y$). Call this path Q.

Weight of Q: $\delta(s \rightarrow x) + w(x, y) + \delta(y \rightarrow u) < d(u)$ [by assumption]

$d(y) = \Delta(R, y) = \min\{ \delta(s \rightarrow r) + w(r, y) \text{ for } r \text{ in } R \}$ [by invariant 2]

$d(y) \leq \delta(s \rightarrow x) + w(x, y)$ [d(y) is the min, so anything else must be ≤]

$\leq \delta(s \rightarrow x) + w(x, y) + \delta(y \rightarrow u)$ [edges are positive. Just add them in]

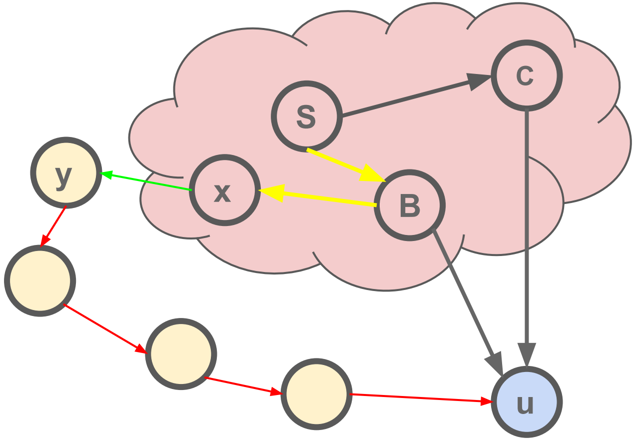
$< d(u)$

Our algorithm: chooses smallest d(u) → d(u) ≤ d(y)
Our math: d(y) < d(u)

Contradiction!
Thus, d(u) is already the shortest distance ($\delta(s \rightarrow u)$)

```
4.   pick u not in R, with the smallest d(u)

5.   R = R union {u}
```

The set R



Let $\delta(a \rightarrow b)$ be the (true) shortest distance from a to b in G

Let $\Delta(X, u)$ be $\min\{ \delta(s \rightarrow v) + w(v, u) \text{ for } v \text{ in } X \}$

Invariant 2 (INV2):
For every neighbour v of all nodes in R,
$d(v) = \Delta(R, v)$

Maintenance - Aside

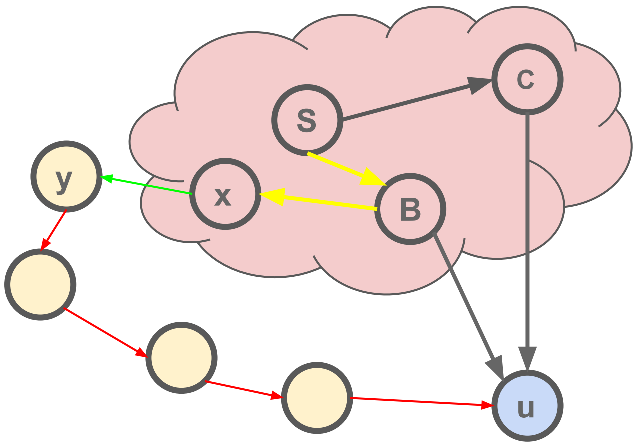
You can also derive the contradiction by starting off with the fact that d(y) ≥ d(u) [by our algorithm].

Then show that d(y) + some edges > d(u) [bcs the edge are positive]

This implies that the path via y is longer (contradicting that we assumed there was a shorter path via y)

```
4.   pick u not in R, with the smallest d(u)

5.   R = R union {u}
```

The set R



Let *δ(a → b)* be the (true) shortest distance from a to b in G

Let ***Δ(X, u)*** be *min{ δ(s → v) + w(v, u) for v in X }*

Invariant 2 (INV2):
For every neighbour v of all nodes in R,
d(v) = Δ(R, v)

Dijkstra **Maintenance** (INV2)

To maintain INV2, observe that we **just added** node u. Now we need to make sure all the neighbours of u are correctly updated

Let $\delta(a \rightarrow b)$ be the (true) shortest distance from a to b in G

Invariant 1 (INV1):
For all node u in R, d(u) = $\delta(s \rightarrow u)$ [d(u) is the shortest distance from s to u already]

Let $\Delta(X, u)$ be $\min\{ \delta(s \rightarrow v) + w(v, u)$ *for v in X* $\}$

Invariant 2 (INV2):
For every neighbour v of all nodes in R, $d(v) = \Delta(R, v)$

```
for v in neighbours(u):
  d(v) = min(d(v), d(u) + w(u, v))
```

Dijkstra **Maintenance** (INV2)

To maintain INV2, observe that we **just added** node u. Now we need to make sure all the neighbours of u are correctly updated

But hey! That's exactly our algorithm in steps 6-7

Let $\boldsymbol{\delta(a \rightarrow b)}$ be the (true) shortest distance from a to b in G

Invariant 1 (INV1):
For all node u in R, d(u) = $\delta(s \rightarrow u)$ [d(u) is the shortest distance from s to u already]

Let $\boldsymbol{\Delta(X, u)}$ be $\min\{\, \delta(s \rightarrow v) + w(v, u)$ for v in $X \,\}$

Invariant 2 (INV2):
For every neighbour v of all nodes in R, $d(v) = \Delta(R, v)$

```
6.   for v in neighbours(u):
7.     d(v) = min(d(v), d(u) + w(u, v))
```

Dijkstra **Termination** (INV1)

After the last iteration of the loop, **R** includes all vertices in **G**.

Let $\delta(a \to b)$ be the (true) shortest distance from a to b in G

Invariant 1 (INV1):
For all node u in R, d(u) = $\delta(s \to u)$ [d(u) is the shortest distance from s to u already]

Let $\Delta(X, u)$ be $min\{ \delta(s \to v) + w(v, u)$ for v in $X \}$

Invariant 2 (INV2):
For every neighbour v of all nodes in R, $d(v) = \Delta(R, v)$

```
3. while R != V:
```

Dijkstra **Termination** (INV1)

After the last iteration of the loop, **R** includes all vertices in **G**.

So **d(u)** is the shortest distance from **s** for every node **u** in G (by INV1)

Let *δ(**a** $\rightarrow$ **b**)* be the (true) shortest distance from a to b in G

Invariant 1 (INV1):
For all node u in R, d(u) = *δ(s $\rightarrow$ u)* [d(u) is the shortest distance from s to u already]

Let ***Δ(X, u)*** be *min{ δ(s $\rightarrow$ v) + w(v, u) for v in X }*

Invariant 2 (INV2):
For every neighbour v of all nodes in R, *d(v) = Δ(R, v)*

```
3. while R != V:
```

Question 2 - 2D Peak-finding

Question 2 - 2D Peak-finding

Given a 2D-array *A* of size *m* rows and *n* columns. *A[i][j]* is called a “peak” if it is **greater than or equal to** its adjacent neighbours
Note: just a peak is enough, it does not have to be a global peak!

n = 4

m = 3

0	23	4	7
1	18	18	1
6	15	0	8

0	23	4	7
1	18	18	1
6	15	0	8

0	23	4	7
1	18	18	1
6	15	0	8

Question 2 - Proposed Pseudocode

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)

    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)

        if candidate_peak is peak:
            return candidate_peak
        else:
            p1 = Find2DPeak(A, but only the right side)
            p2 = Find2DPeak(A, but only the left side)
            if p1 is a peak return p1 else return p2
```

Question 2

Let’s try on this A

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)

    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)

        if candidate_peak is peak:
            return candidate_peak
        else:
            p1 = Find2DPeak(A, but only the right side)
            p2 = Find2DPeak(A, but only the left side)
            if p1 is a peak return p1 else return p2
```

0	23	4	7	23	4	7
1	18	18	1	18	18	1
6	15	0	8	15	0	8

Question 2

Take the middle column

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)

    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)

        if candidate_peak is peak:
            return candidate_peak
        else:
            p1 = Find2DPeak(A, but only the right side)
            p2 = Find2DPeak(A, but only the left side)
            if p1 is a peak return p1 else return p2
```

0	23	4	7	23	4	7
1	18	18	1	18	18	1
6	15	0	8	15	0	8

Question 2

8 is the max on this column

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)

    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)

        if candidate_peak is peak:
            return candidate_peak
        else:
            p1 = Find2DPeak(A, but only the right side)
            p2 = Find2DPeak(A, but only the left side)
            if p1 is a peak return p1 else return p2
```

0	23	4	7	23	4	7
1	18	18	1	18	18	1
6	15	0	8	15	0	8

Question 2

15 is greater than 8. 8 is not a peak!

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)

    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)

        if candidate_peak is peak:
            return candidate_peak
        else:
            p1 = Find2DPeak(A, but only the right side)
            p2 = Find2DPeak(A, but only the left side)
            if p1 is a peak return p1 else return p2
```

0	23	4	7	23	4	7
1	18	18	1	18	18	1
6	15	0	8	15	0	8

Question 2

Recurse on the right side

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)

    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)

        if candidate_peak is peak:
            return candidate_peak
        else:
            p1 = Find2DPeak(A, but only the right side)
            p2 = Find2DPeak(A, but only the left side)
            if p1 is a peak return p1 else return p2
```

0	23	4	7	23	4	7
1	18	18	1	18	18	1
6	15	0	8	15	0	8

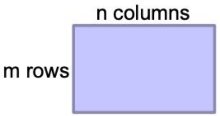
Question 2

Recurse on the left side

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)

    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)

        if candidate_peak is peak:
            return candidate_peak
        else:
            p1 = Find2DPeak(A, but only the right side)
            p2 = Find2DPeak(A, but only the left side)
            if p1 is a peak return p1 else return p2
```

0	23	4	7	23	4	7
1	18	18	1	18	18	1
6	15	0	8	15	0	8

Question 2



Suppose we are given a 2D-array A of size m rows by n columns. An element in the array $A[i][j]$ is called a "peak" if it is **greater than or equal to** its adjacent neighbours (if they exist -- an element is always considered greater than or equal to non-existent elements). For example, the 8 in the middle is a peak in the following array.

$$\begin{array}{cccc} * & * & 5 & * \\ * & 8 & 8 & 3 \\ * & * & 2 & * \end{array}$$

Consider the following algorithm to return any "peak":

```
Find2DPeak(A):
    If A only has a column, return the maximal element of the column
    Otherwise:
        Select the middle column of the A
        Find the maximal element of the column
        If the maximal element is a peak, return that element
        Else
            p1 = Find2DPeak(right half of A excluding middle col)
            p2 = Find2DPeak(left half of A excluding middle col)
            If p1 or p2 is a peak, return either one, otherwise return None
```

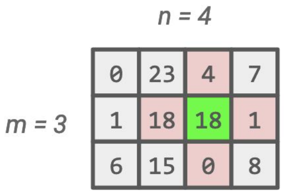
What is the runtime of the algorithm?

- $\Theta(mn)$
- $\Theta(m \lg n)$
- $\Theta(\lg m \lg n)$
- $\Theta(m^2 \lg n)$

Time Complexity - Intuition

- You are dividing the array into two, but you are doing recursion on **both sides**
- You are still checking every columns!

Time Complexity - Method one



Recall: m rows and n columns.

Consider the **recurrence with two parameters**:

Time Complexity - Method one

$n = 4$

$m = 3$

0	23	4	7
1	18	18	1
6	15	0	8

Recall: m rows and n columns.

Consider the **recurrence with two parameters**:

$T(m, 1) = cm$

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)

    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)

        if candidate_peak is peak:
            return candidate_peak
        else:
            p1 = Find2DPeak(A, but only the right side)
            p2 = Find2DPeak(A, but only the left side)
            if p1 or p2 is a peak return p1 else None
```

Time Complexity - Method one

Recall: m rows and n columns.

Consider the **recurrence with two parameters**:

$T(m, 1) = cm$

$T(m, n) = 2T(m, n/2) + cm$

Possible to solve this with recursion tree!
(Exercise)

n = 4

m = 3

0	23	4	7
1	18	18	1
6	15	0	8

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)

    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)

        if candidate_peak is peak:
            return candidate_peak
        else:
            p1 = Find2DPeak(A, but only the right side)
            p2 = Find2DPeak(A, but only the left side)
            if p1 or p2 is a peak return p1 else None
```

Time Complexity - Method two

Observation: The time to process every column is the same: *θ(m)*

n = 4

m = 3

0	23	4	7
1	18	18	1
6	15	0	8

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)

    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)

        if candidate_peak is peak:
            return candidate_peak
        else:
            p1 = Find2DPeak(A, but only the right side)
            p2 = Find2DPeak(A, but only the left side)
            if p1 or p2 is a peak return p1 else None
```

Time Complexity - Method two

n = 4

m = 3

0	23	4	7
1	18	18	1
6	15	0	8

Observation: The time to process every column is the same: *θ(m)*

Idea: Count the number of columns processed! Multiply processing time later

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)

    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)

        if candidate_peak is peak:
            return candidate_peak
        else:
            p1 = Find2DPeak(A, but only the right side)
            p2 = Find2DPeak(A, but only the left side)
            if p1 or p2 is a peak return p1 else None
```

Time Complexity - Method two

$n = 4$

$m = 3$

0	23	4	7
1	18	18	1
6	15	0	8

Observation: The time to process every column is the same: $\theta(m)$

Idea: Count the number of columns processed! Multiply processing time later

- In the divide step, the middle column is processed, and no column is processed in the combined step - $\theta(1)$
- **Two** subproblems - each **half** the size!

Recurrence:

$C(n) = 2C(n/2) + \theta(1)$

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

Time Complexity - Method two

$n = 4$, $m = 3$

0	23	4	7
1	18	18	1
6	15	0	8

Observation: The time to process every column is the same: *θ(m)*

Idea: Count the number of columns processed! Multiply processing time later

- In the divide step, the middle column is processed, and no column is processed in the combined step - *θ(1)*
- **Two** subproblems - each **half** the size!

Recurrence:

$C(n) = 2C(n/2) + \theta(1)$

Master Theorem (Case 1):

$\log_b a = \log_2 2 = 1$

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

Time Complexity - Method two

n = 4

m = 3

0	23	4	7
1	18	18	1
6	15	0	8

Observation: The time to process every column is the same: $\theta(m)$

Idea: Count the number of columns processed! Multiply processing time later

- In the divide step, the middle column is processed, and no column is processed in the combined step - $\theta(1)$
- **Two** subproblems - each **half** the size!

Recurrence:

$C(n) = 2C(n/2) + \theta(1)$

Master Theorem (Case 1):

$\log_b a = \log_2 2 = 1$

$f(n) = \theta(1) = O(n^{1-\varepsilon})$

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

Time Complexity - Method two

n = 4

m = 3

0	23	4	7
1	18	18	1
6	15	0	8

Observation: The time to process every column is the same: *θ(m)*

Idea: Count the number of columns processed! Multiply processing time later

- In the divide step, the middle column is processed, and no column is processed in the combined step - *θ(1)*
- **Two** subproblems - each **half** the size!

Recurrence:

$C(n) = 2C(n/2) + \theta(1)$

Master Theorem (Case 1):

$\log_b a = \log_2 2 = 1$

$f(n) = \theta(1) = O(n^{1-\varepsilon})$

Therefore, $C(n) = \theta(n^1)$

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

Time Complexity - Method two

n = 4

m = 3

0	23	4	7
1	18	18	1
6	15	0	8

Observation: The time to process every column is the same: $\theta(m)$

Idea: Count the number of columns processed! Multiply processing time later

- In the divide step, the middle column is processed, and no column is processed in the combined step - $\theta(1)$
- **Two** subproblems - each **half** the size!

Recurrence:

$C(n) = 2C(n/2) + \theta(1)$

Master Theorem (Case 1):

$\log_b a = \log_2 2 = 1$

$f(n) = \theta(1) = O(n^{1-\varepsilon})$

Therefore, $C(n) = \theta(n^1)$

Total time:

$\theta(mn)$

Time Complexity - Method two

Total time of *$\theta(mn)$* is ***no better*** than just iterating the 2D array normally! Does this mean that this algorithm is bad?

Time Complexity - Method two

Total time of $\theta(mn)$ is ***no better*** than just iterating the 2D array normally! Does this mean that this algorithm is bad?

Not exactly -- we can get new ideas from it on how to solve the problem!

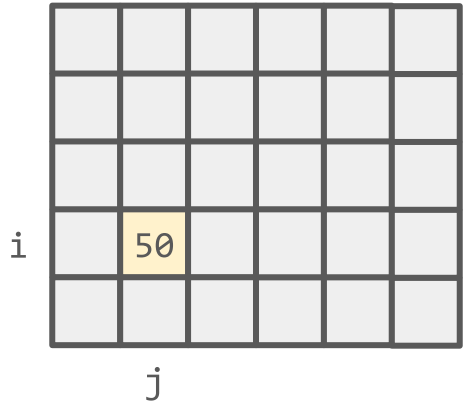
Question 3: Finding useful property for better 2D Peak-finding

Question 3

Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.

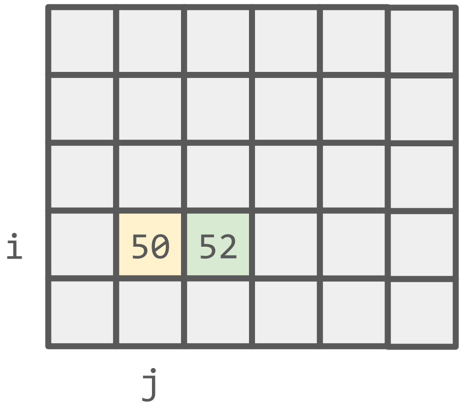
Question 3

Let $A[i][j]$ be the largest element in column j. Assume that $A[i][j + 1] \geq A[i][j]$. Argue in the subarray B, where B is A but column $j + 1$ onwards, any peak that is the largest element in its column is also a peak of the entire array A.



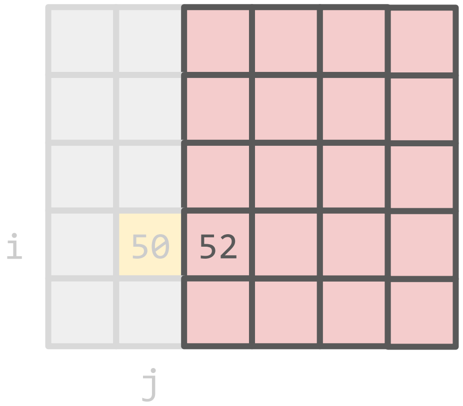
Question 3

Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.



Question 3

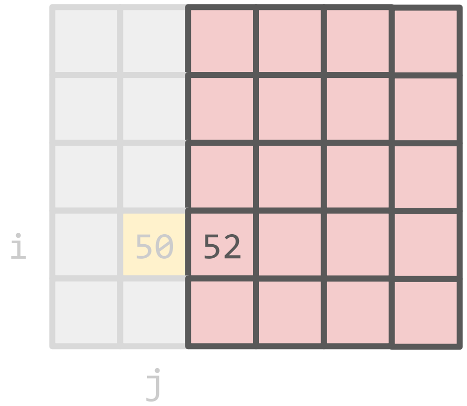
Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.



This question was added later on for completeness of q4. Hence it was not in the qn sheet

Question 3b (not in tutorial qn sheet)

Argue that this sub-array will always have a peak.

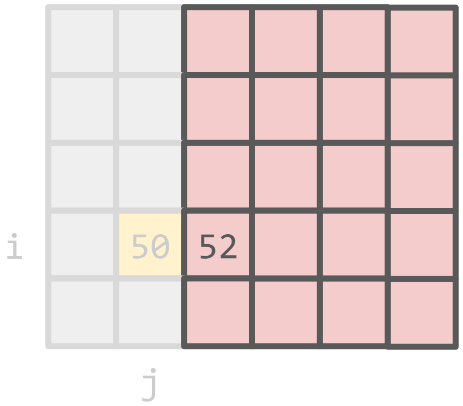


This question was added later on for completeness of q4. Hence it was not in the qn sheet

Question 3b (not in tutorial qn sheet)

Argue that this sub-array will always have a peak:

Any 2D array must have a peak. Simply take the maximum element in the 2D array, which will be a peak in this array.



Question 3

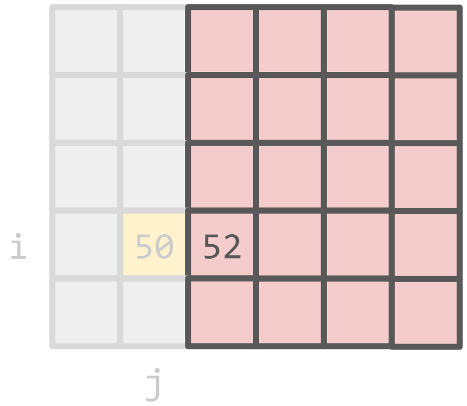
Suppose we are given a 2D-array A of size *m* rows by *n* columns. An element in the array $A[i][j]$ is called a "peak" if it is **greater than or equal to** its adjacent neighbours (if they exist -- an element is always considered greater than or equal to non-existent elements). For example, the 8 in the middle is a peak in the following array.

$$\begin{array}{cccc} * & * & 5 & * \\ * & 8 & 8 & 3 \\ * & * & 2 & * \end{array}$$

Let $A[i][j]$ be the largest element in column j. Assume that $A[i][j+1] \geq A[i][j]$. Argue that any peak in the the subarray $A[1..m][j+1..n]$ that is the largest element in its column is also a peak of the entire array A.

Question 3

Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.



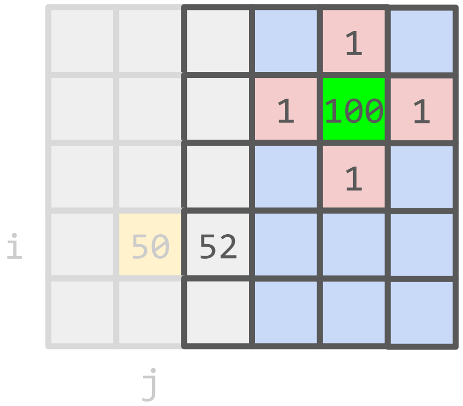
Now back to the original question at hand!

Q3 - Case 1

Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.

A peak in the columns ***j* + 2** onwards will be a peak in the overall array!

Intuition: Regardless of whether you add back the part you “removed”, whether this is a peak or not is **unaffected**

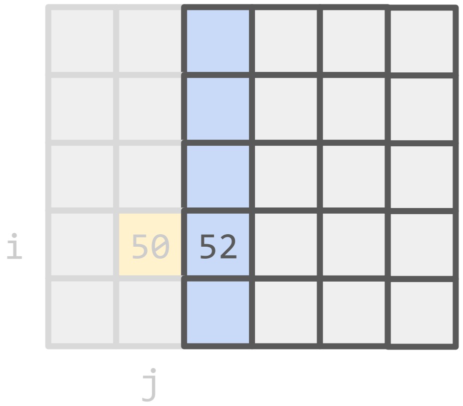


Q3 - Case 2

Let $A[i][j]$ be the largest element in column j. Assume that $A[i][j + 1] \geq A[i][j]$. Argue in the subarray B, where B is A but column $j + 1$ onwards, any peak that is the largest element in its column is also a peak of the entire array A.

A peak in the column ***j* + 1** needs a bit more work

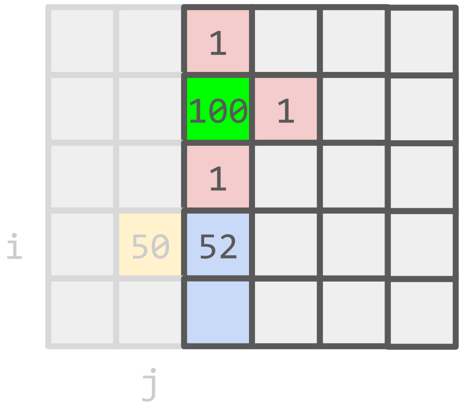
Intuition: If you add back the part of A that you removed, **will that peak still remain a peak**?



Q3 - Case 2

Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.

Let's say that the element 100 (note, larger than 52) here is a peak. Within the subarray, it's clearly a peak

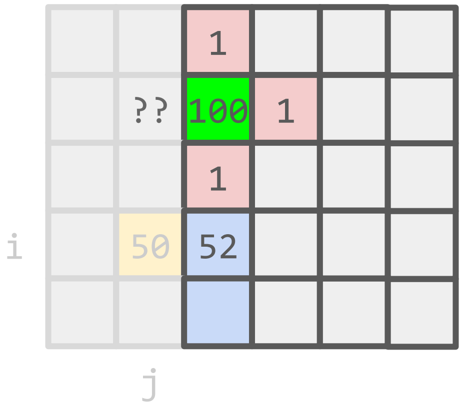


Q3 - Case 2

Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.

Let's say that the element 100 (note, larger than 52) here is a peak. Within the subarray, it's clearly a peak

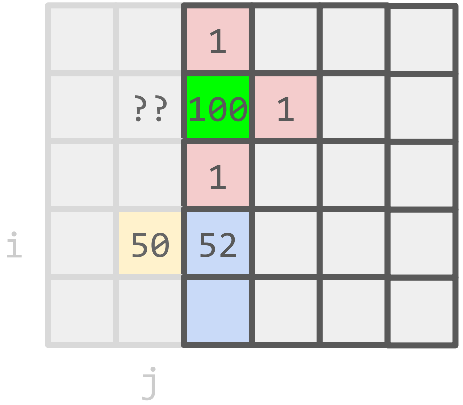
But what about after considering the element to its left?? Will 100 stay a peak?



Q3 - Case 2

Let $A[i][j]$ be the largest element in column j. Assume that $A[i][j + 1] \geq A[i][j]$. Argue in the subarray B, where B is A but column $j + 1$ onwards, any peak that is the largest element in its column is also a peak of the entire array A.

Observe: 50 is the largest element in column j. **So '??' has to be ≤ 50**

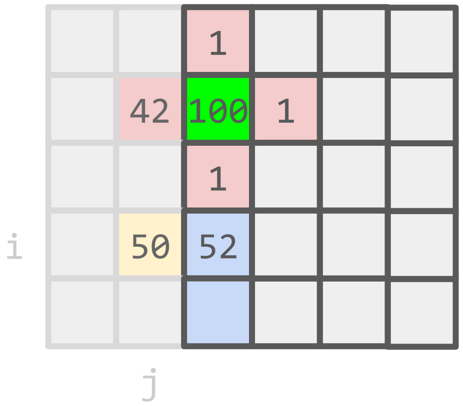


Q3 - Case 2

Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.

Observe: 50 is the largest element in column j. **So '??' has to be ≤ 50**

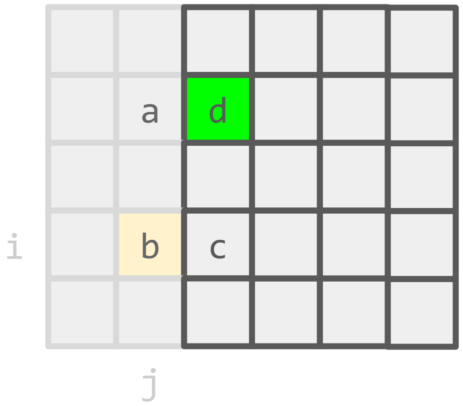
Let's say we put 42. Then 100 is **still a peak**, even when we consider the bigger array!



Q3 - Case 2

Let $A[i][j]$ be the largest element in column j. Assume that $A[i][j + 1] \geq A[i][j]$. Argue in the subarray B, where B is A but column $j + 1$ onwards, any peak that is the largest element in its column is also a peak of the entire array A.

Generalising: Let d be the peak in the subarray.

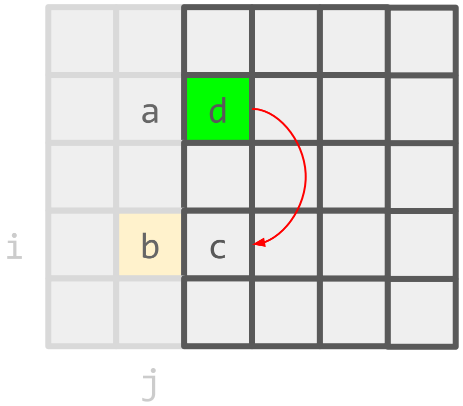


Q3 - Case 2

Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.

Generalising: Let d be the peak in the subarray.

d
≥ c (the peak should be the largest element in the column)

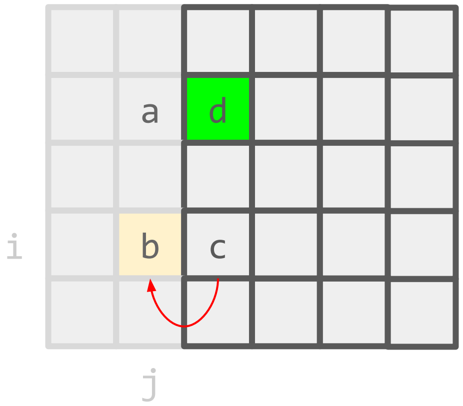


Q3 - Case 2

Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.

Generalising: Let d be the peak in the subarray.

d
≥ c (the peak should be the largest element in the column)
≥ b (*A[i][j+1] ≥ A[i][j]*)



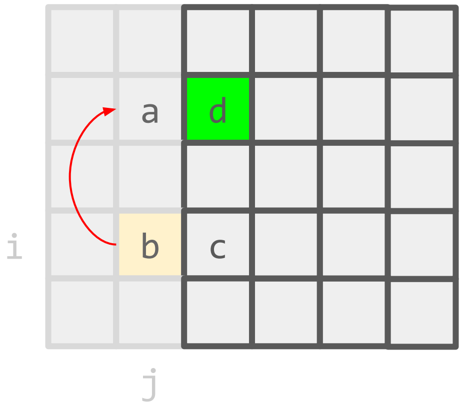
Q3 - Case 2

Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.

Generalising: Let d be the peak in the subarray.

d
≥ c (the peak should be the largest element in the column)
≥ b (*A[i][j+1] ≥ A[i][j]*)
≥ a (*A[i][j]* is the largest in col j)

Thus, d ≥ a



Question 4: Better 2D Peak-finding

Question 4

Suppose we are given a 2D-array A of size *m* rows by *n* columns. An element in the array $A[i][j]$ is called a "peak" if it is **greater than or equal to** its adjacent neighbours (if they exist -- an element is always considered greater than or equal to non-existent elements). For example, the 8 in the middle is a peak in the following array.

$$\begin{array}{cccc} * & * & 5 & * \\ * & 8 & 8 & 3 \\ * & * & 2 & * \end{array}$$

Using the idea in Question 3, describe an algorithm that is asymptotically faster that the one given in Question 2. What is it's runtime?

Question 4 (Solution)

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)
    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)
        if candidate_peak is peak:
            return candidate_peak
        else if candidate_peak <= element to its right:
            return Find2DPeak(A, but only the right side)
        else:
            return Find2DPeak(A, but only the left side)
```

Either side must contain a peak, by qn 3b

Question 4 (Solution)

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)
    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)
        if candidate_peak is peak:
            return candidate_peak
        else if candidate_peak <= element to its right:
            return Find2DPeak(A, but only the right side)
        else:
            return Find2DPeak(A, but only the left side)
```

If element to its left and to its right are smaller than `candidate_peak`, this case has been accounted for by the check if `candidate_peak is peak`

Question 4 (Solution)

Let *A[i][j]* be the largest element in column *j*. Assume that *A[i][j + 1] ≥ A[i][j]*. Argue in the subarray B, where B is A but column *j + 1* onwards, any peak that is the largest element in its column is also a peak of the entire array A.

```
def Find2DPeak(A):
    if num_of_cols(A) == 1:
        return max(A)
    else:
        mid_col = middle column of A
        candidate_peak = max(mid_col)
        if candidate_peak is peak:
            return candidate_peak
        else if candidate_peak <= element to its right:
            return Find2DPeak(A, but only the right side)
        else:
            return Find2DPeak(A, but only the left side)
```

Important: return the max in the column! Otherwise, claim in Q3 does not hold

Q4 Analysis

Similar as before, but this time the recurrence for the number of columns processed is: $C(n) = C(n/2) + \theta(1)$

Q4 Analysis

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Similar as before, but this time the recurrence for the number of columns processed is: $C(n) = C(n/2) + \theta(1)$

By Master Theorem (case 2):

$log_b a = log_2 1 = 0$

Q4 Analysis

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Similar as before, but this time the recurrence for the number of columns processed is: $C(n) = C(n/2) + \theta(1)$

By Master Theorem (case 2):

$log_b a = log_2 1 = 0$

$f(n) = \theta(1) = \theta(n^0 (logn)^0) = \theta(1)$ [our choice of k = 0]

Q4 Analysis

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Similar as before, but this time the recurrence for the number of columns processed is: $C(n) = C(n/2) + \theta(1)$

By Master Theorem (case 2):
$\log_b a = \log_2 1 = 0$
$f(n) = \theta(1) = \theta(n^0 (\log n)^0) = \theta(1)$ [our choice of k = 0]

So $C(n) = \theta(n^0 (\log n)^{0+1}) = \theta(\log n)$ [the +1 is because of master theorem]

Q4 Analysis

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Similar as before, but this time the recurrence for the number of columns processed is: $C(n) = C(n/2) + \theta(1)$

By Master Theorem (case 2):
$\log_b a = \log_2 1 = 0$
$f(n) = \theta(1) = \theta(n^0 (\log n)^0) = \theta(1)$ [our choice of k = 0]

So $C(n) = \theta(n^0 (\log n)^{0+1}) = \theta(\log n)$ [the +1 is because of master theorem]

Total running time: $\theta(m\log n)$