

Performance of Cuckoo Hashing

One can show that “long eviction sequences” happen with very low probability.

Fact: Assuming hash values are uniformly randomly distributed, expected time of n put operations is $O(n)$ provided $N > 2n$

Fact: Cuckoo hashing achieves worst-case $O(1)$ time for lookups and removals

Another ADT: Set

A **set** is an unordered collection of elements, without duplicates that typically supports efficient membership tests.

Elements of a set are like keys of a map, but without any auxiliary values.

Set ADT

add(e): add the element e to S (if not already present)

remove(e): remove the element e from S (if present)

contains(e): returns whether e is present in S

iterator(): returns an iterator of the elements of S

There is also support for the traditional mathematical set operations **union**, **intersection**, and **subtraction** of two sets S and T:

$$S \cup T = \{e : e \text{ in } S \text{ or } e \text{ in } T\}$$

$$S \cap T = \{e : e \text{ in } S \text{ and } e \text{ in } T\}$$

$$S - T = \{e : e \text{ in } S \text{ and } e \text{ not in } T\}$$

addAll(T): Updates S to include all elements of T, effectively replacing S by $S \cup T$

retainAll(T): Updates S to keep only elements that are also elements in T, effectively replacing S by $S \cap T$

removeAll(T): Updates S to remove any elements that are also elements in T, effectively replacing S by $S - T$

Set implemented via Map

- Use a Map to store the keys, and ignore the value.
- Allows `contains(k)` to be answered by `get(k)`
- Similarly for add and remove
- Using HashMap for Map, gives main Set operations that usually can be performed in $O(1)$ time.

MultiSet

- Like a Set, but allows duplicates
 - also called a Bag
 - operation **count(e)** says how many occurrences of e in collection
 - **remove(e)** removes ONE occurrence (provided e is in the collection already)
- Implement by Map where the element is the key, and the associated value is the number of occurrences

Practice vs Theory

In practice hash tables implementation are usually fast and people use them **as though** put, get, and remove take $O(1)$ time.

The analyses we covered in lecture assume uniformly random hash values, which are not possible to implement in practice. Removing these assumptions is an active area of research well beyond the scope of this class.

In theory we do not know of an implementation of hash tables that can perform put, get, and remove $O(1)$ time in the worst case.

So you cannot use such a data structure in your assignments.

Theory of Hashing

There is rich theory of hashing beyond the basic hashing schemes we covered in class:

- Quadratic probing
- Double hashing
- Perfect hashing
- Universal hash families that are k-wise independent
- Cuckoo hashing with a stash
- Pseudorandom generators
- Cryptographic hash functions
- etc.

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Data structures and Algorithms

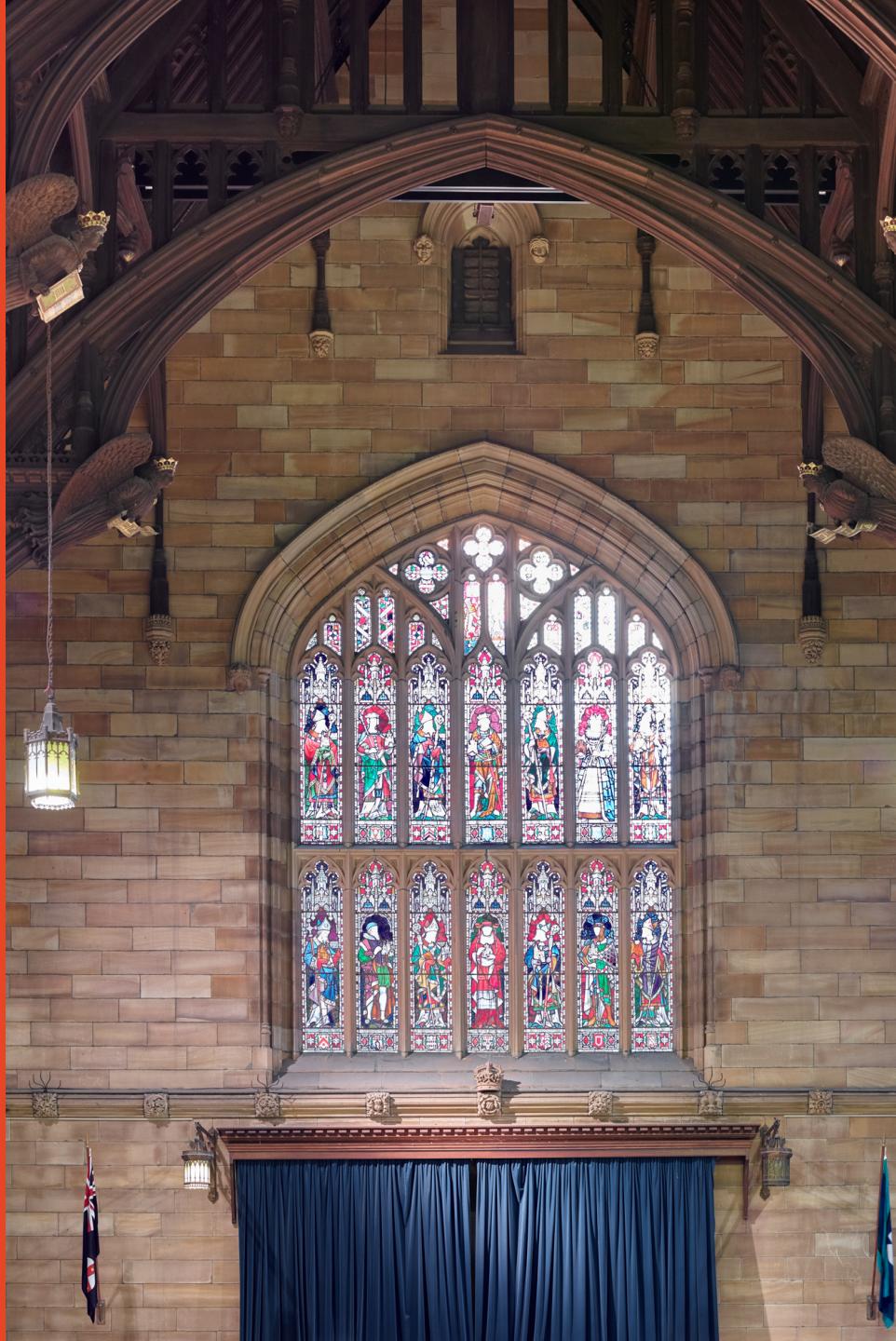
Lecture 9: Graphs [GT 13.1-3]

Dr. André van Renssen
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



THE UNIVERSITY OF
SYDNEY



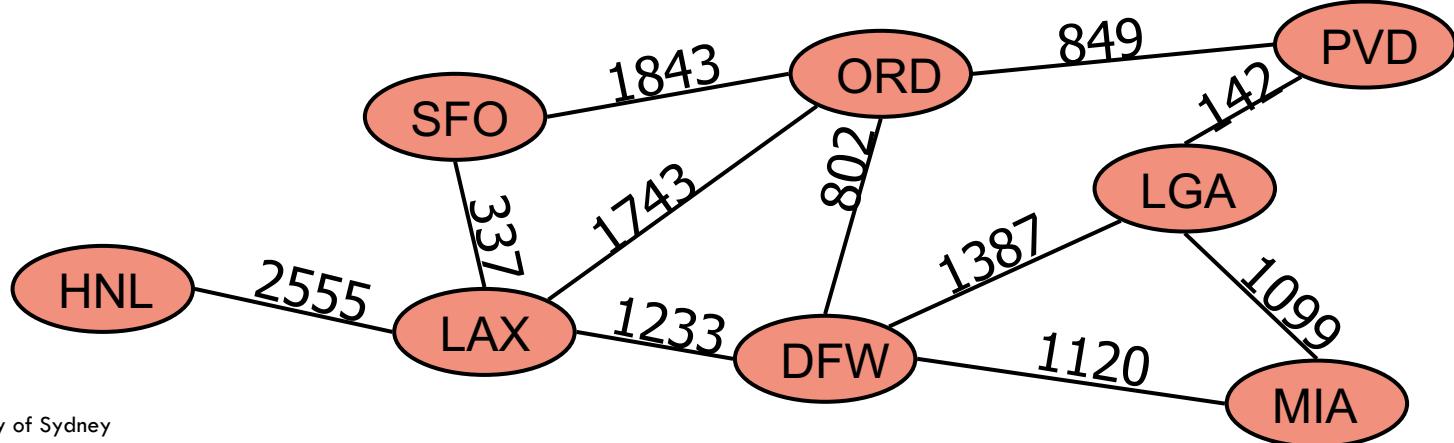
Graphs

A graph **G** is a pair (V, E) , where

- V is a set of nodes, called **vertices**
- E is a collection of pairs of vertices, called **edges**

Example:

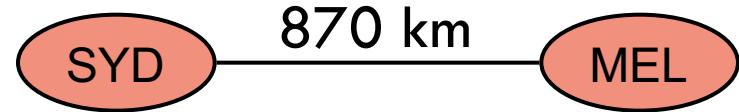
- A vertex represents an airport and stores the three-letter airport code
- An edge represents a flight route between two airports and stores the mileage of the route



Edge Types

Directed edge

- ordered pair of vertices (u, v)
- u is the origin/tail
- v is the destination/head
- e.g., a flight



Undirected edge

- unordered pair of vertices (u, v)
- e.g., a two-way road

Applications

Electronic circuits

- Printed circuit board
- Integrated circuit

Transportation networks

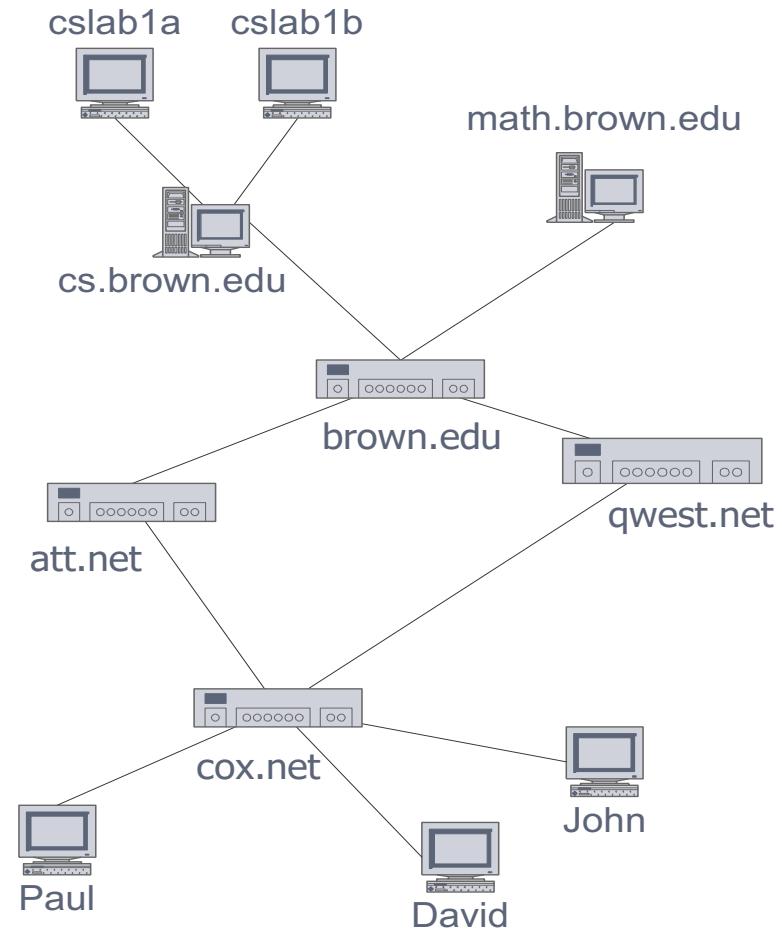
- Highway network
- Flight network

Computer networks

- Internet
- Web

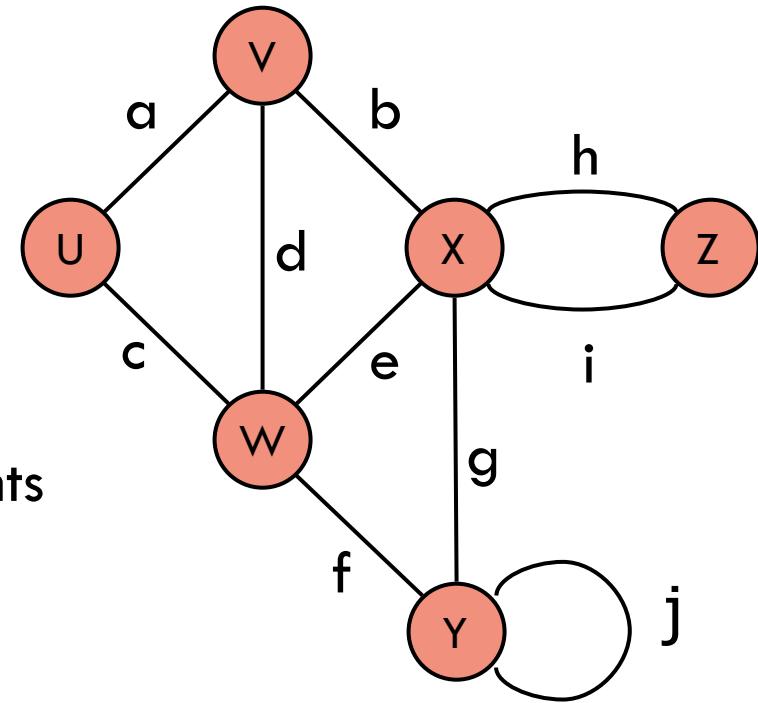
Modeling

- Entity-relationship diagram
- Gantt precedence constraints



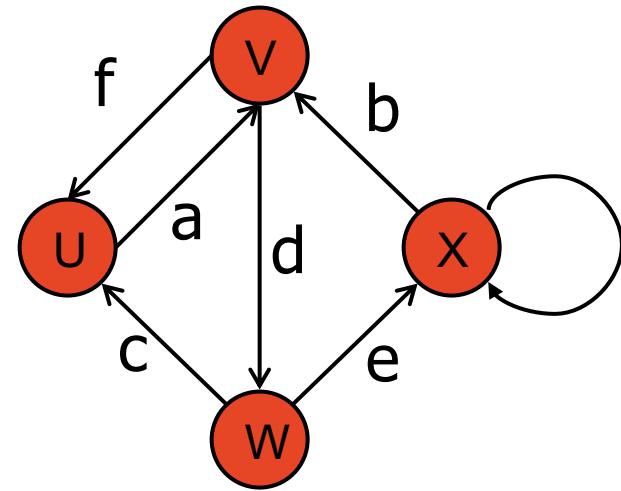
Terminology (Undirected graphs)

- Edges connect **endpoints**
e.g., W and Y for edge f
- Edges are **incident** on endpoints
e.g., a, d, and b are incident on V
- **Adjacent** vertices are connected
e.g., U and V are adjacent
- **Degree** is # of edges on a vertex
e.g., X has degree 5
- **Parallel edges** share same endpoints
e.g., h and i are parallel
- **Self-loop** have only one endpoint
e.g., j is a self-loop
- **Simple** graphs have no parallel or self-loops



Terminology (Directed graphs)

- Edges go from **tail** to **head**
e.g., W is the tail of c and U its head
- **Out-degree** is # of edges out of a vertex
e.g., W has out-degree 2
- **In-degree** is # of edges into a vertex
e.g., W has in-degree 1
- **Parallel edges** share tail and head
e.g., no parallel edge on the right
- **Self-loop** have same head and tail
e.g., X has a self-loop
- **Simple** directed graphs have no parallel or self-loops, but are allowed to have anti-parallel loops like f and a



Terminology

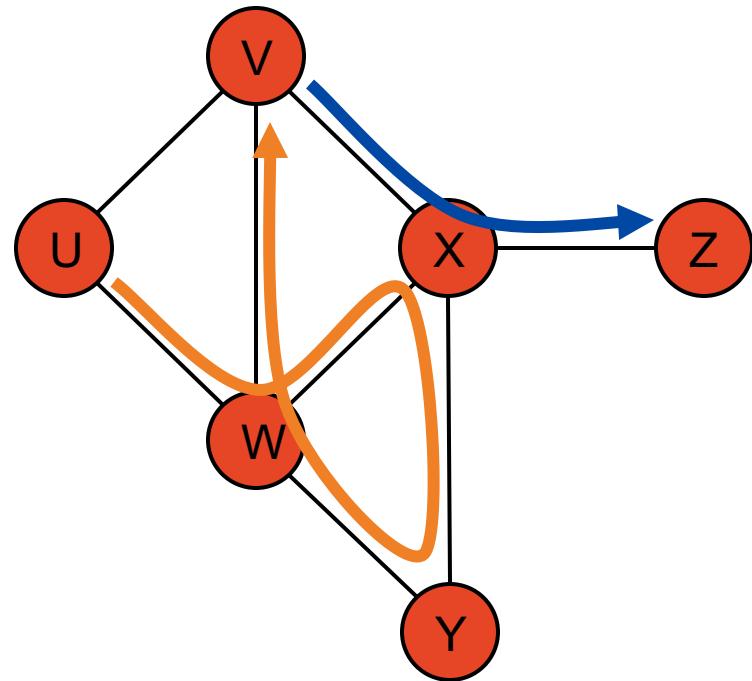
A **path** is a sequence of vertices such that every pair of consecutive vertices is connected by an edge.

A simple path is one where all vertices are distinct

Examples

- (V, X, Z) is a simple path
- (U, W, X, Y, W, V) is a path that is not simple

A (simple) path from s to t is also called an s - t path.



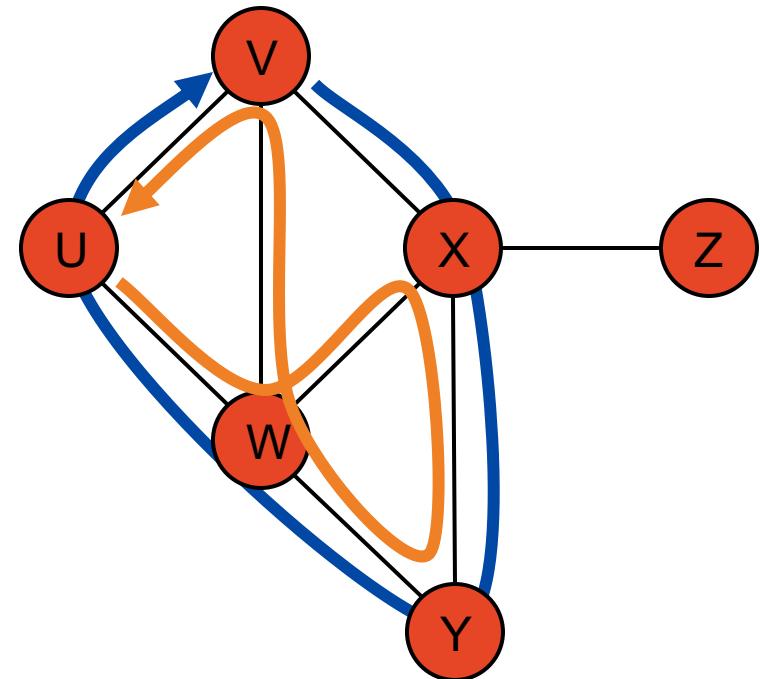
Terminology

A **cycle** is defined by a path that starts and ends at the same vertex

A **simple cycle** is one where all vertices are distinct

Examples

- (V, X, Y, W, U, V) is a simple cycle
- (U, W, X, Y, W, V, U) is a cycle that is not simple



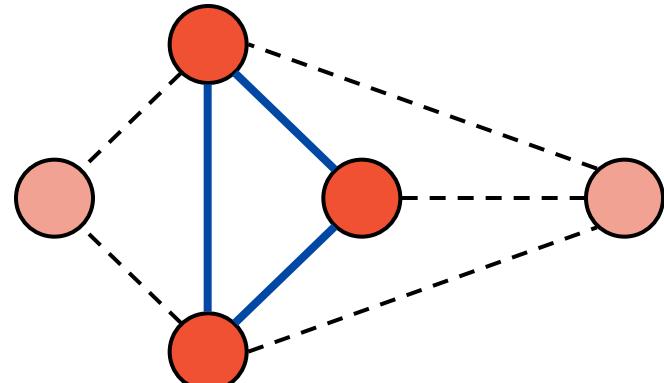
An **acyclic graph** has no cycles

Subgraphs

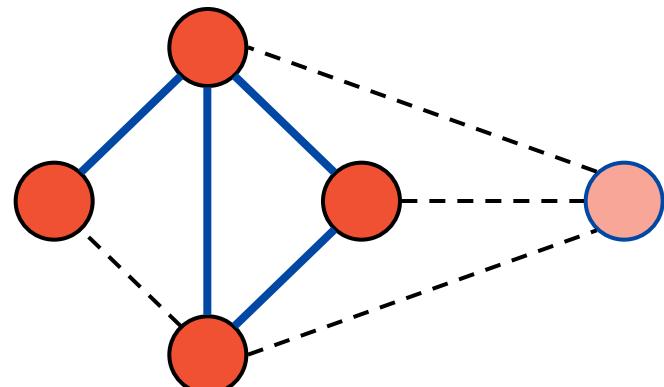
Let $G=(V, E)$ be a graph. We say $S=(U, F)$ is a subgraph of G if $U \subseteq V$ and $F \subseteq E$

A subset $U \subseteq V$ induces a graph $G[U] = (U, E[U])$ where $E[U]$ are the edges in E with endpoints in U

A subset $F \subseteq E$ induces a graph $G[F] = (V[F], F)$ where $V[F]$ are the endpoints of edges in F



Subgraph induced by red vertices

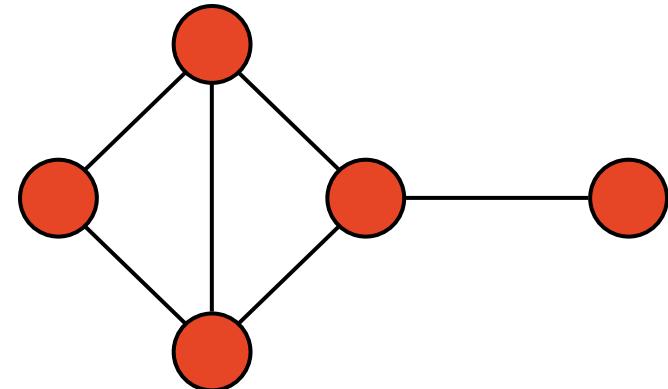


Subgraph induced by blue edges

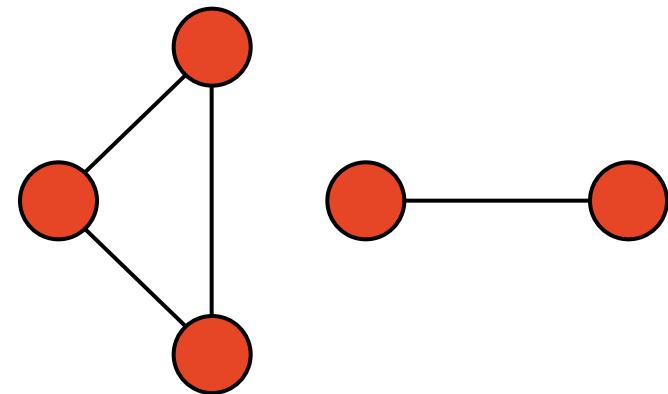
Connectivity

A graph $G=(V, E)$ is connected if there is a path between every pair of vertices in V

A connected component of a graph G is a maximal connected subgraph of G



Connected graph



Graph with two connected components

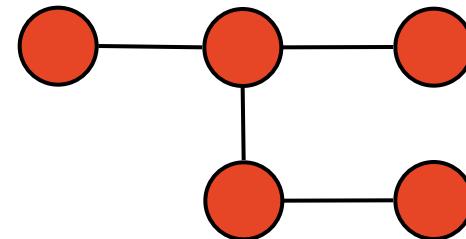
Trees and Forests

An unrooted tree T is a graph such that

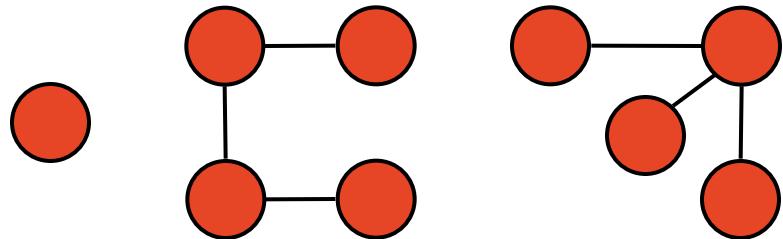
- T is connected
- T has no cycles

A forest is a graph without cycles. In other words, its connected components are trees

Fact: Every tree on n vertices has $n-1$ edges



Tree



Forest

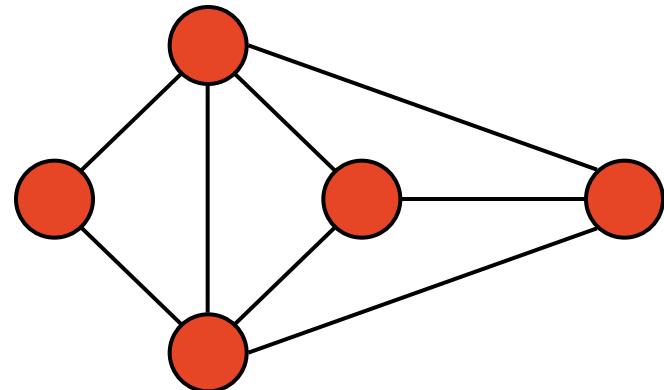
Spanning Trees and Forests

A spanning tree is a connected subgraph on the same vertex set

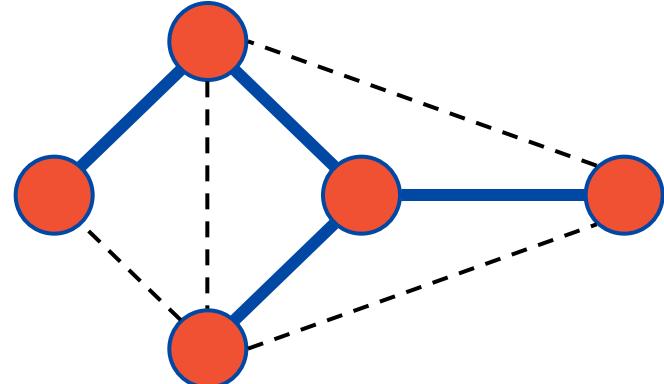
A spanning tree is not unique unless the graph is a tree

Spanning trees have applications to the design of communication networks

A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

Properties

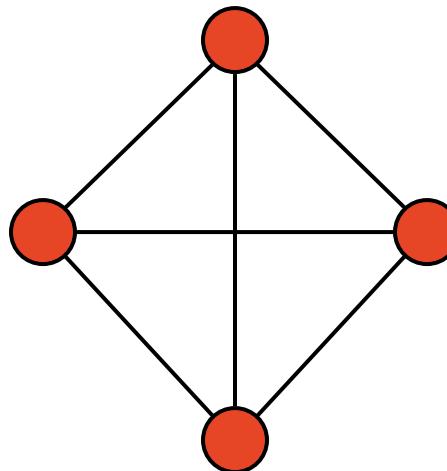
Fact: $\sum_{v \in V} \deg(v) = 2m$

Fact: In a simple undirected graph $m \leq n(n - 1)/2$

Fact: In a simple directed graph $m \leq n(n - 1)$

Notation

n	number of vertices
m	number of edges
Δ	maximum degree



Example: K_4

$$n = 4$$

$$m = 6$$

$$\max \deg = 3$$

Graph ADT

We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.

A Vertex stores an associated object (e.g., an airport code) that is retrieved with a getElement() method.

An Edge stores an associated object (e.g., a flight number, travel distance) that is retrieved with a getElement() method.

Directed Graph ADT

Undirected
Graph
alternatives

degree(v) ←

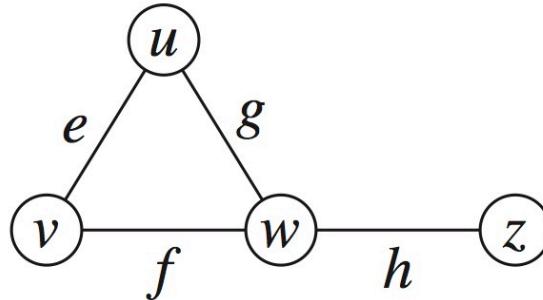
incidentEdges(v) ←

- numVertices()**: Returns the number of vertices of the graph.
- vertices()**: Returns an iteration of all the vertices of the graph.
- numEdges()**: Returns the number of edges of the graph.
- edges()**: Returns an iteration of all the edges of the graph.
- getEdge(u, v)**: Returns the edge from vertex u to vertex v , if one exists; otherwise return null. For an undirected graph, there is no difference between $\text{getEdge}(u, v)$ and $\text{getEdge}(v, u)$.
- endVertices(e)**: Returns an array containing the two endpoint vertices of edge e . If the graph is directed, the first vertex is the origin and the second is the destination.
- opposite(v, e)**: For edge e incident to vertex v , returns the other vertex of the edge; an error occurs if e is not incident to v .
- outDegree(v)**: Returns the number of outgoing edges from vertex v .
- inDegree(v)**: Returns the number of incoming edges to vertex v . For an undirected graph, this returns the same value as does $\text{outDegree}(v)$.
- outgoingEdges(v)**: Returns an iteration of all outgoing edges from vertex v .
- incomingEdges(v)**: Returns an iteration of all incoming edges to vertex v . For an undirected graph, this returns the same collection as does $\text{outgoingEdges}(v)$.
- insertVertex(x)**: Creates and returns a new Vertex storing element x .
- insertEdge(u, v, x)**: Creates and returns a new Edge from vertex u to vertex v , storing element x ; an error occurs if there already exists an edge from u to v .
- removeVertex(v)**: Removes vertex v and all its incident edges from the graph.
- removeEdge(e)**: Removes edge e from the graph.

Edge List Structure

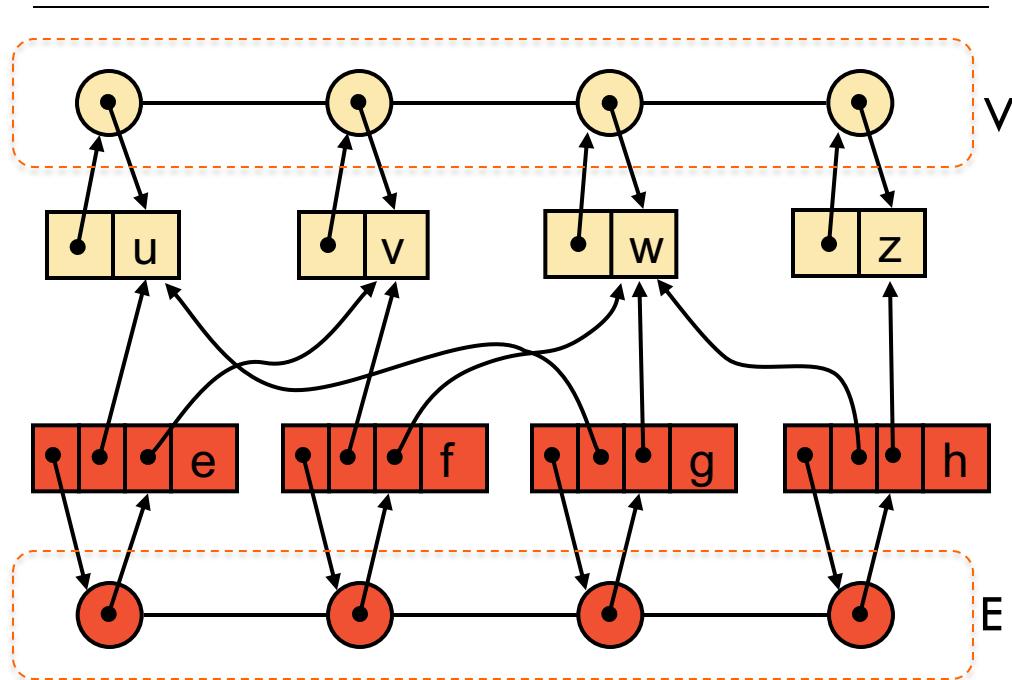
Vertex sequence holds

- sequence of vertices
- vertex object keeps track of its position in the sequence



Edge sequence

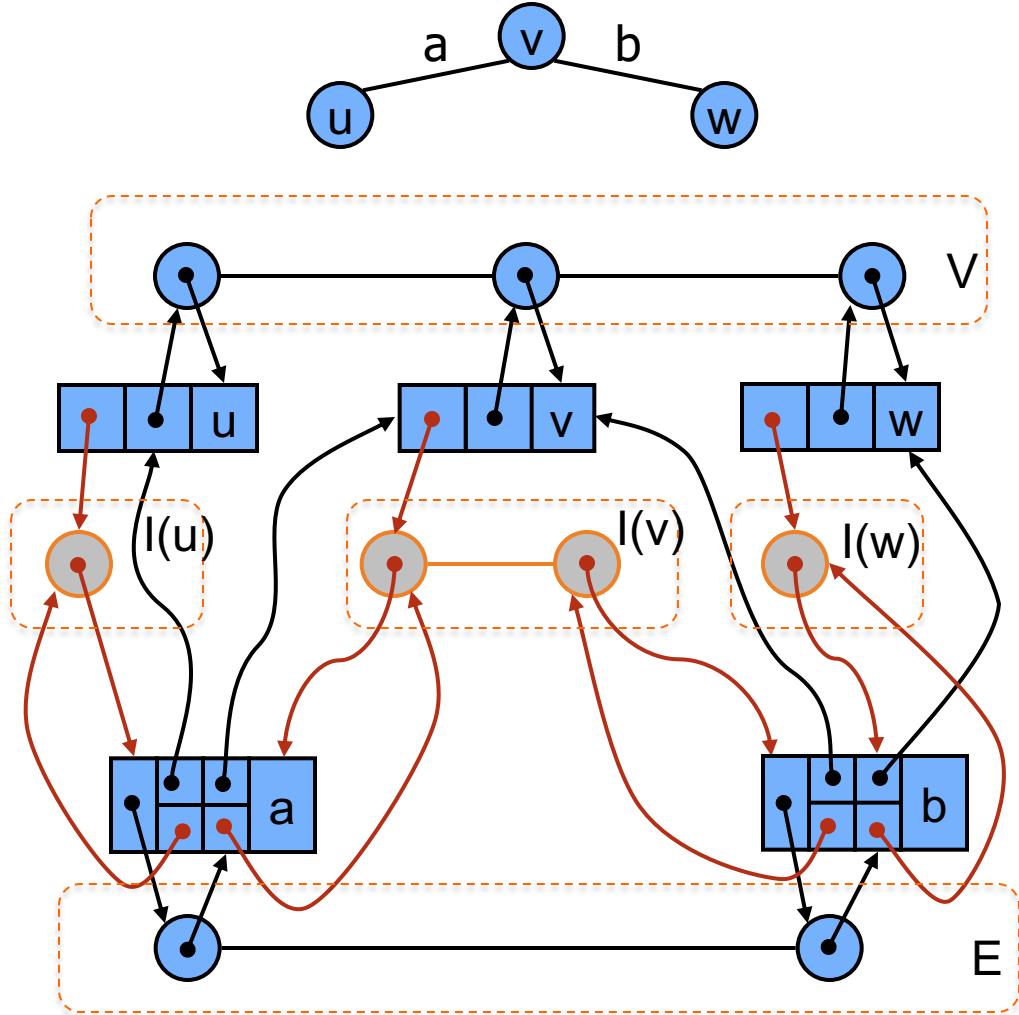
- sequence edges
- edge object keeps track of its position in the sequence
- Edge object points to the two vertices it connects



Adjacency List

Additionally each vertex keeps a sequence of edges incident on it

Edge objects keep reference to their position in the incidence sequence of its endpoints

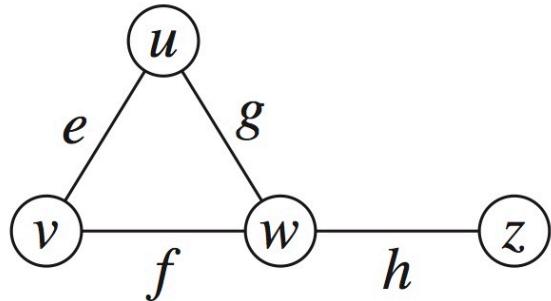


Adjacency Matrix Structure

Vertex array induces an index from 0 to n-1 for each vertex

2D-array adjacency matrix

- Reference to edge object for adjacent vertices
- Null for nonadjacent vertices



	0	1	2	3
$u \rightarrow$		e	g	
$v \rightarrow$	e		f	
$w \rightarrow$	g	f		h
$z \rightarrow$			h	

Asymptotic performance

<ul style="list-style-type: none"> ■ n vertices, m edges ■ no parallel edges ■ no self-loops 	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n + m)$	$O(n + m)$	$O(n^2)$
<code>incidentEdges(v)</code>	$O(m)$	$O(\deg(v))$	$O(n)$
<code>getEdge(u, v)</code>	$O(m)$	$O(\min(\deg(u), \deg(v)))$	$O(1)$
<code>insertVertex(x)</code>	$O(1)$	$O(1)$	$O(n^2)$
<code>insertEdge(u, v, x)</code>	$O(1)$	$O(1)$	$O(1)$
<code>removeVertex(v)</code>	$O(m)$	$O(\deg(v))$	$O(n^2)$
<code>removeEdge(e)</code>	$O(1)$	$O(1)$	$O(1)$

Graph traversals

A fundamental kind of algorithmic operation that we might wish to perform on a graph is **traversing the edges and the vertices** of that graph.

A **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges.

For example, a **web crawler**, which is the data collecting part of a search engine, must explore a graph of hypertext documents by examining its vertices, which are the documents, and its edges, which are the hyperlinks between documents.

A traversal is efficient if it visits all the vertices and edges in linear time: **$O(n+m)$** where **n**=number of vertices, **m**=number of edges.

Graph traversal techniques

A systematic and structured way of visiting all the vertices and all the edges of a graph

Two main strategies:

- Depth first search
- Breadth first search

Given adjacency list representation of the graph with n vertices and m edges both traversal run in $O(n + m)$ time

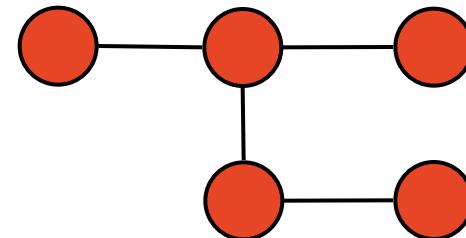
Reminder: Trees and Forests

An unrooted tree T is a graph such that

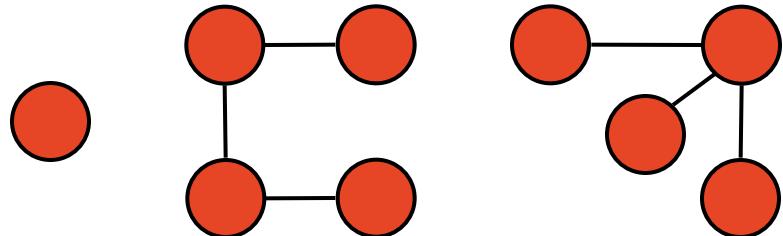
- T is connected
- T has no cycles

A forest is a graph without cycles. In other words, its connected components are trees

Fact: Every tree on n vertices has $n-1$ edges



Tree

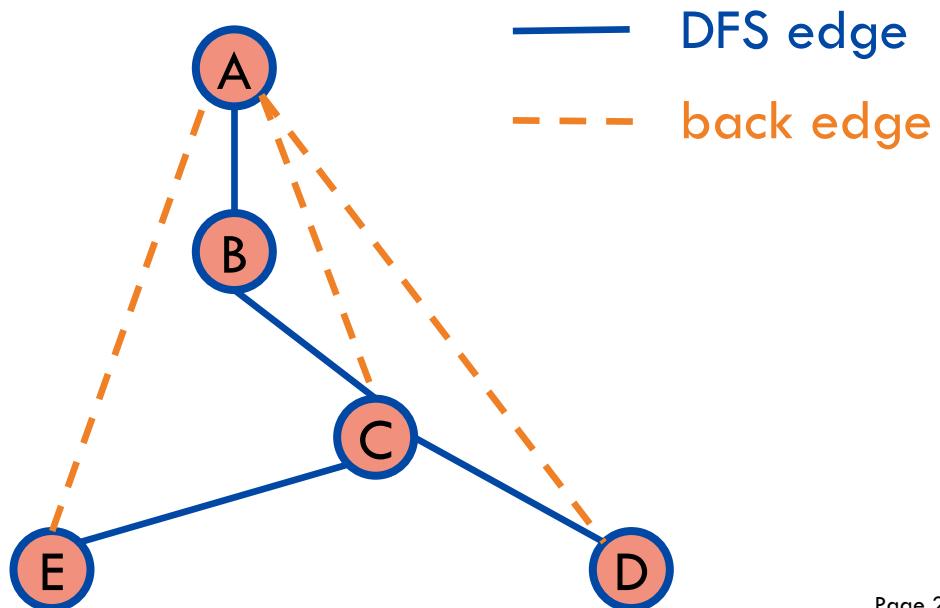
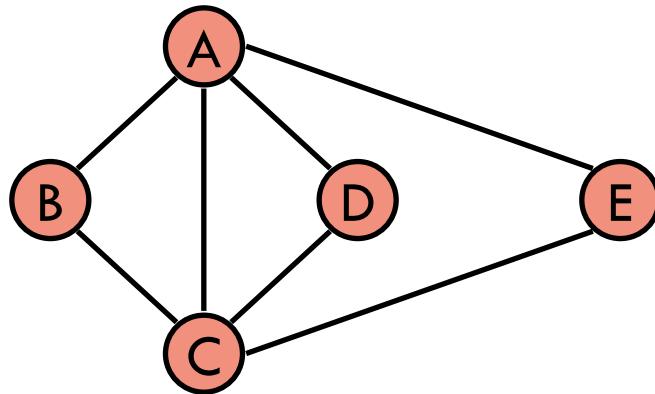


Forest

Depth-First Search (DFS)

This strategy tries to follow outgoing edges leading to yet unvisited vertices whenever possible, and backtrack if “stuck”

If an edge is used to discover a new vertex, we call it a DFS edge, otherwise we call it a back edge



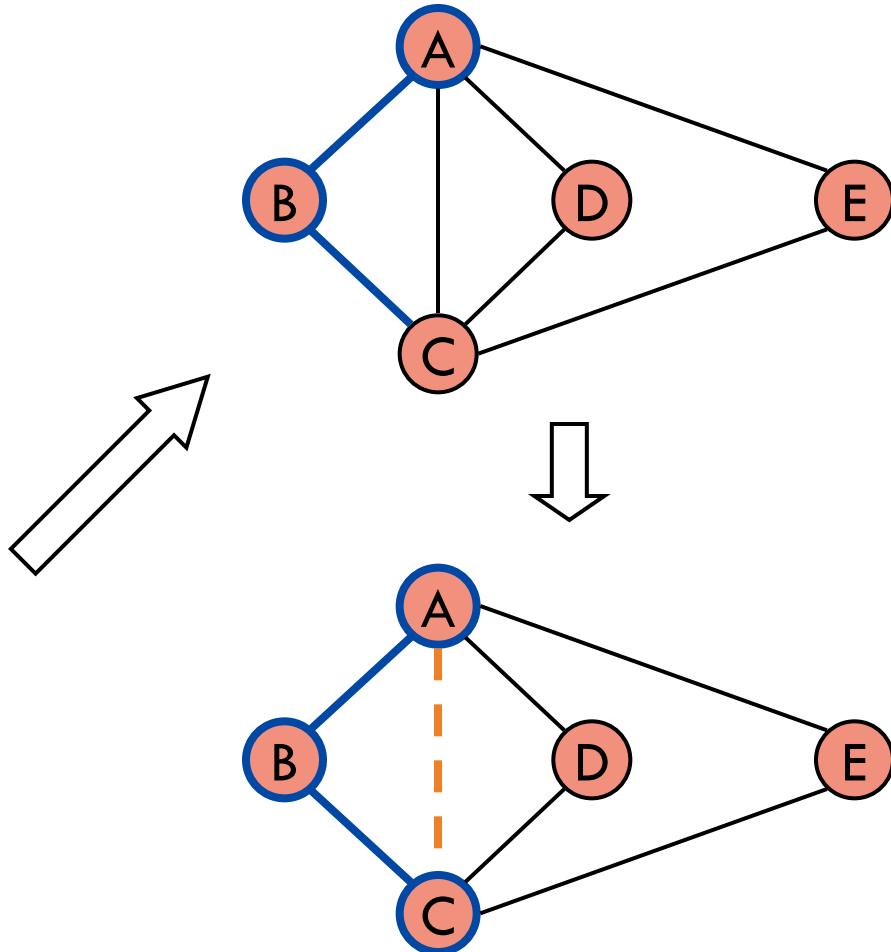
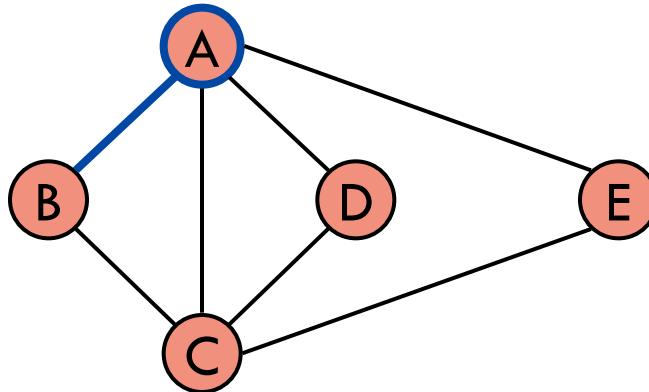
DFS pseudocode

```
def DFS(G):  
  
    # set things up for DFS  
    for u in G.vertices(): do  
        visited[u] ← False  
        parent[u] ← None  
  
    # visit vertices  
    for u in G.vertices(): do  
        if not visited[u] then  
            DFS_visit(u)  
  
    return parent
```

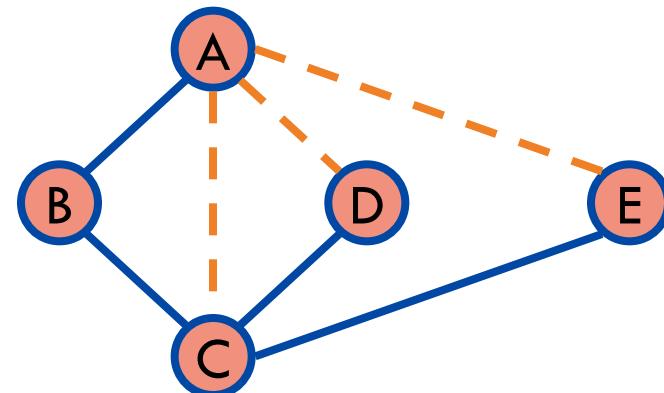
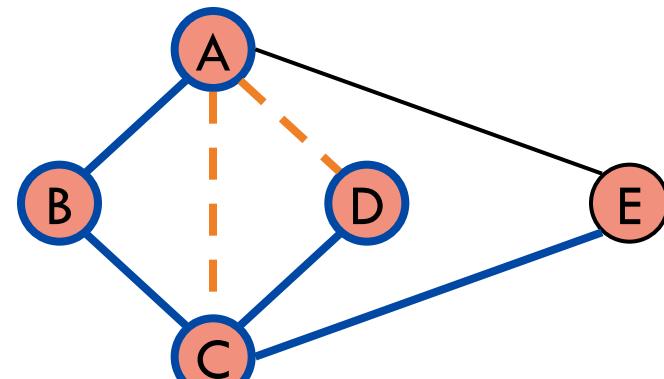
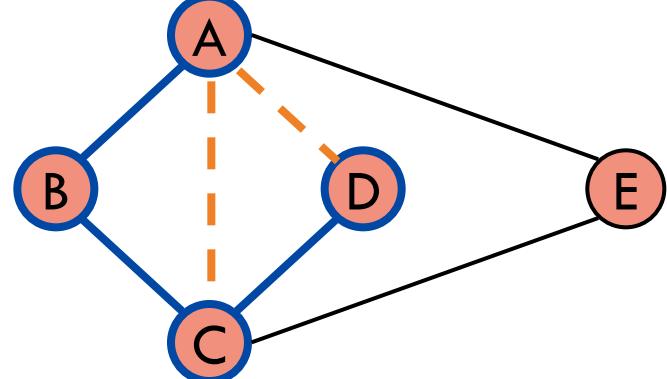
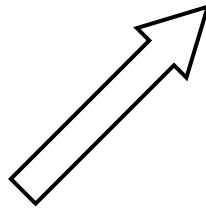
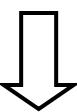
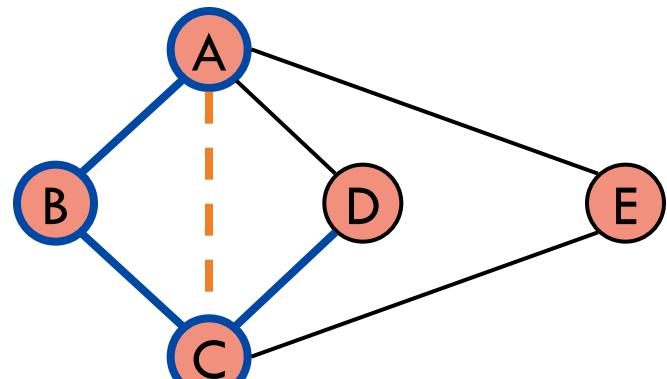
```
def DFS_visit(u):  
  
    visited[u] ← True  
  
    # visit neighbors of u  
    for v in G.incident(u): do  
        if not visited[v] then  
            parent[v] ← u  
            DFS_visit(v)
```

Example

- unexplored vertex
- visited vertex
- unexplored edge
- DFS edge
- - - back edge



Example (cont.)



DFS main function performance

```
def DFS(G):  
  
    # set things up for DFS  
    for u in G.vertices(): do  
        visited[u] ← False  
        parent[u] ← None  
  
    # visit vertices  
    for u in G.vertices(): do  
        if not visited[u]: then  
            DFS_visit(u)  
  
    return parent
```

Assuming adjacency list representation

$O(n)$ time

$O(n)$ time not counting work done in `DFS_visit`

DFS_visit performance

Assuming adjacency list representation

$O(\deg(u))$ time not counting work done in recursive calls to DFS_visit

Thus, overall time is

$$O(\sum_u \deg(u)) = O(m)$$

```
def DFS_visit(u):
    visited[u] ← True
    # visit neighbors of u
    for v in G.incident(u) do
        if not visited[v] then
            parent[v] ← u
            DFS_visit(v)
```

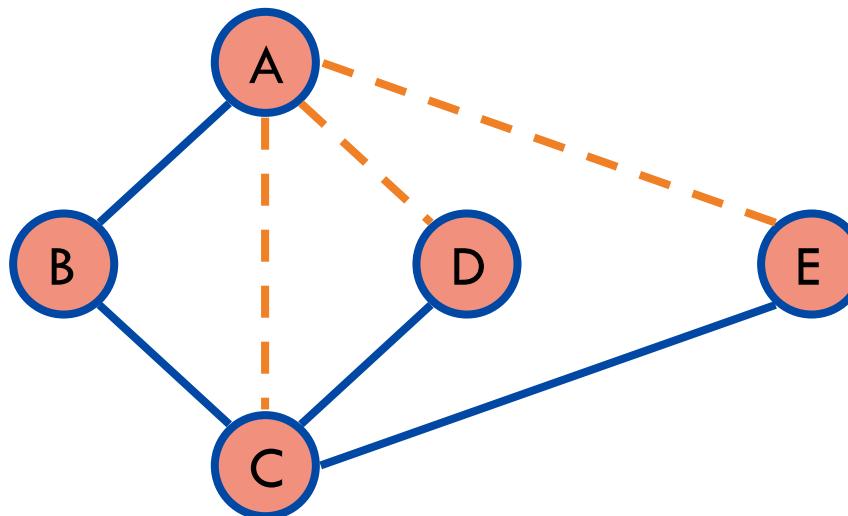
Properties of DFS

Let C_v be the connected component of v in our graph G

Fact: $\text{DFS_visit}(v)$ visits all vertices in C_v

Fact: Edges $\{ (u, \text{parent}[u]): u \in C_v \}$ form a spanning tree of C_v

Fact: Edges $\{ (u, \text{parent}[u]): u \in V \}$ form a spanning forest of G



DFS Applications

DFS can be used to solve other graph problems in $O(n + m)$ time:

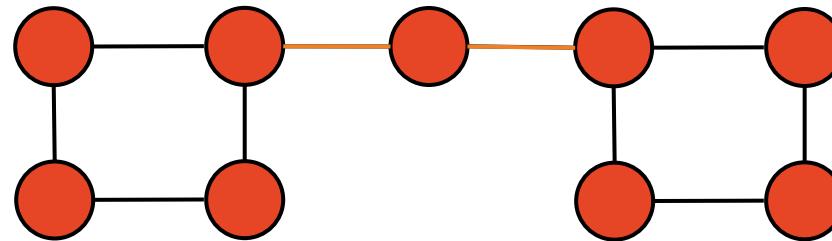
- Find a path between two given vertices, if any
- Find a cycle in the graph
- Test whether a graph is connected
- Compute connected components of a graph
- Compute spanning tree of a graph (if connected)

And is the building block of more sophisticated algorithms:

- testing bi-connectivity
- finding cut edges
- finding cut vertices

Identifying cut edges

In a connected graph $G=(V, E)$, we say that an edge (u, v) in E is a cut edge if $(V, E \setminus \{(u, v)\})$ is not connected



Identifying cut edges

In a connected graph $G=(V, E)$, we say that an edge (u, v) in E is a cut edge if $(V, E \setminus \{(u, v)\})$ is not connected

The cut edge problem is to identify all cut edges

Trivial $O(m^2)$ time algorithm: For each edge (u, v) in E , remove (u, v) and check using DFS if G is still connected, put back (u, v)

Better $O(nm)$ time algorithm: Only test edges in a DFS tree of G

Identifying cut edges in $O(n+m)$ time

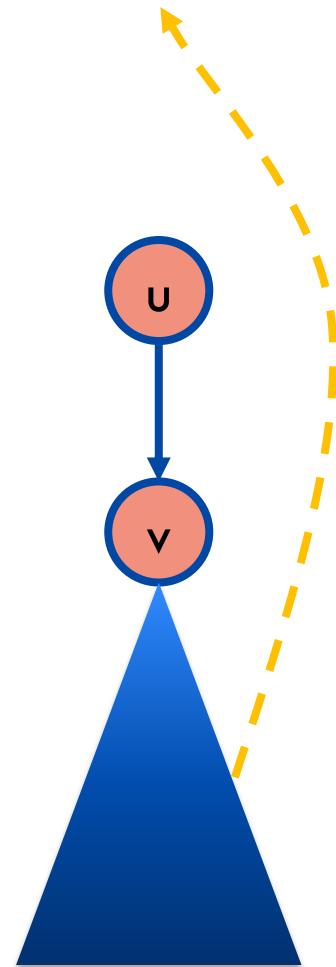
Compute a DFS tree of the input graph $G=(V, E)$

For every u in V , compute $\text{level}[u]$, its level in the DFS tree

For every vertex v compute the highest level that we can reach by taking DFS edges down the tree and then one back edge up. Call this $\text{down_and_up}[v]$

Fact: A DFS edge (u, v) where $u = \text{parent}[v]$ is not a cut edge if and only if $\text{down_and_up}[v] \leq \text{level}[u]$

Basis of an $O(n+m)$ time algorithm for finding cut edges

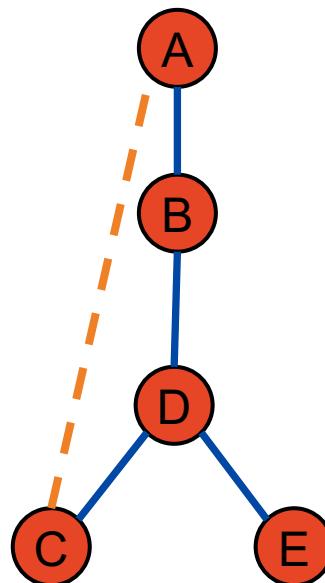
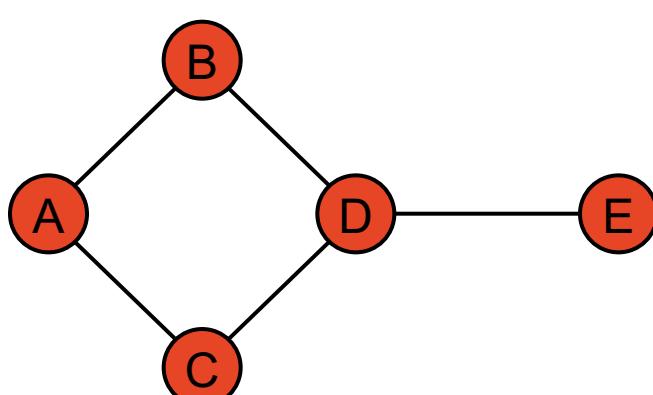


Identifying cut edges in $O(n+m)$ time

Compute a DFS tree of the input graph $G=(V, E)$

For every u in V , compute $\text{level}[u]$, its level in the DFS tree

For every vertex v compute the highest level that we can reach by taking DFS edges down the tree and then one back edge up. Call this $\text{down_and_up}[v]$

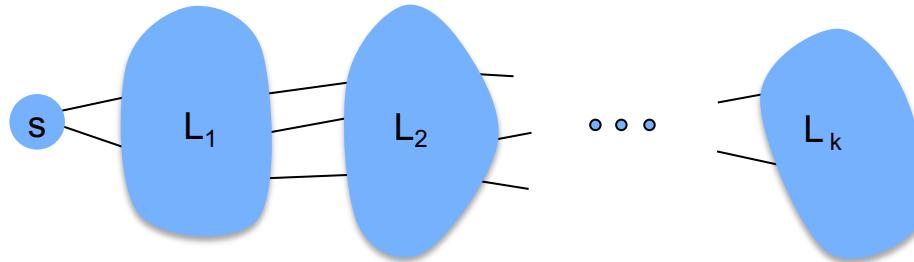


	level	d&u
A	0	0
B	1	0
C	2	0
D	2	0
E	3	3

Breadth-First Search (BFS)

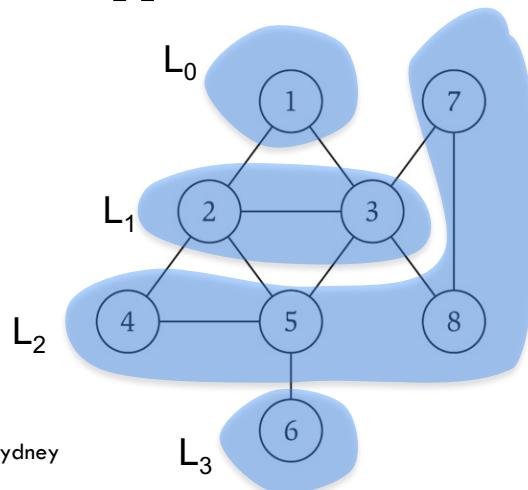
This strategy tries to visit all vertices at distance k from a start vertex s before visiting vertices at distance $k + 1$:

- $L_0 = \{s\}$
- $L_1 = \text{vertices one hop away from } s$
- $L_2 = \text{vertices two hops away from } s \text{ but no closer}$
- ⋮
- $L_k = \text{vertices } k \text{ hops away from } s \text{ but no closer}$

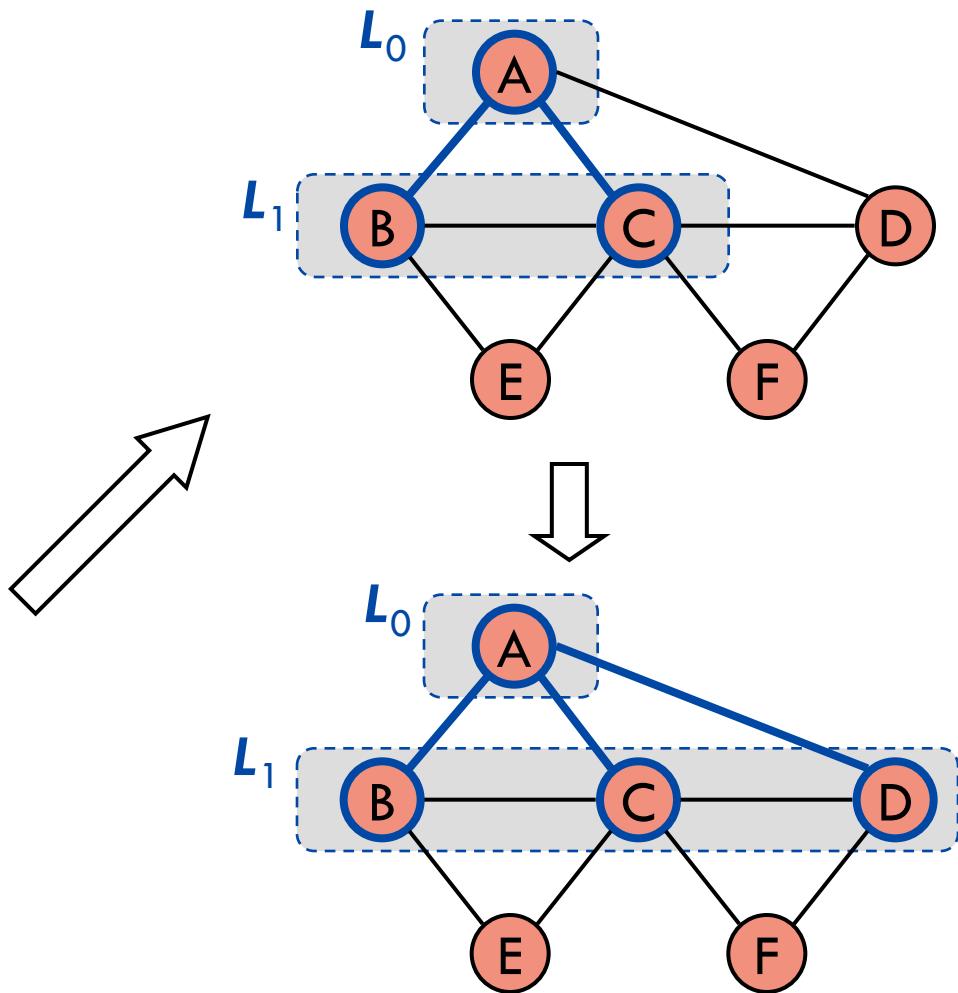
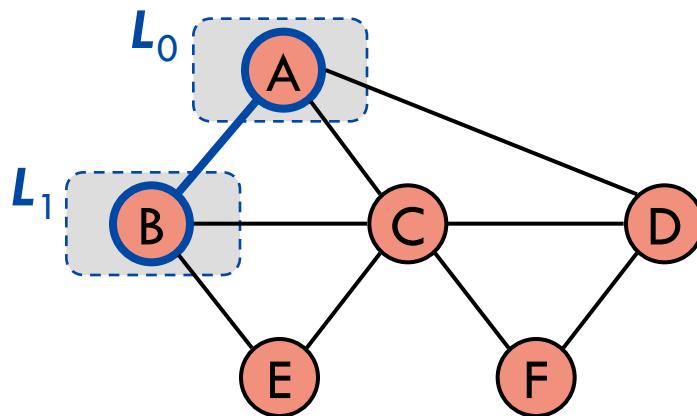
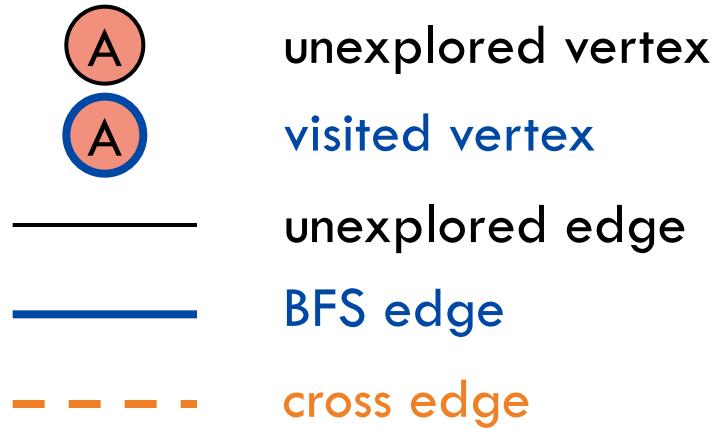


BFS

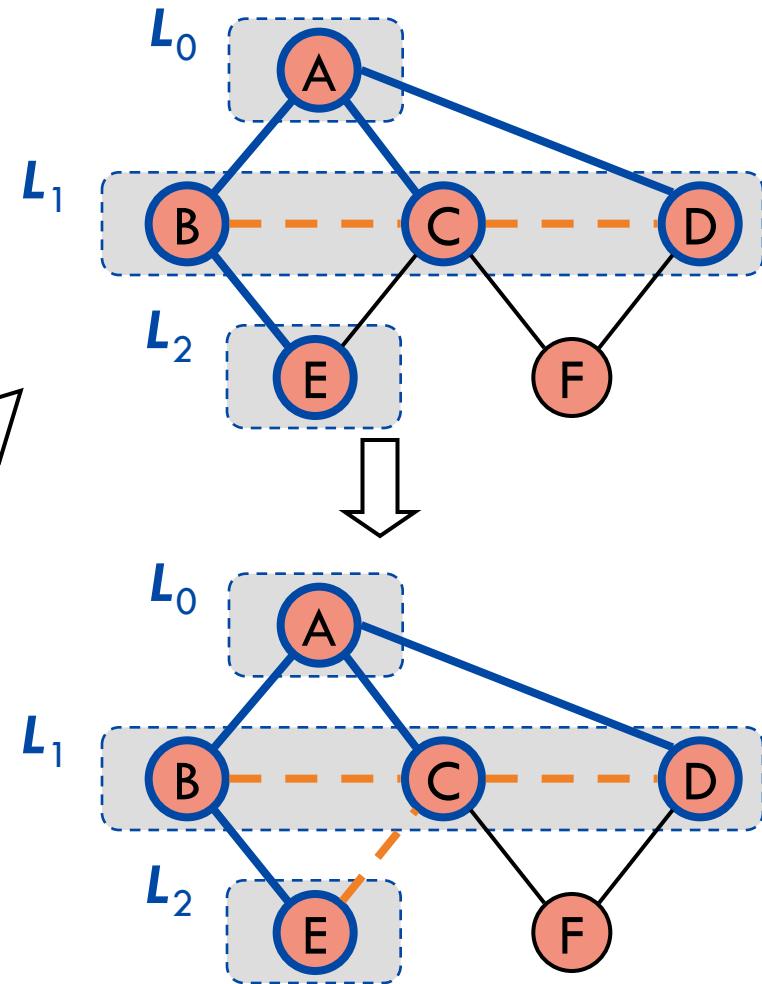
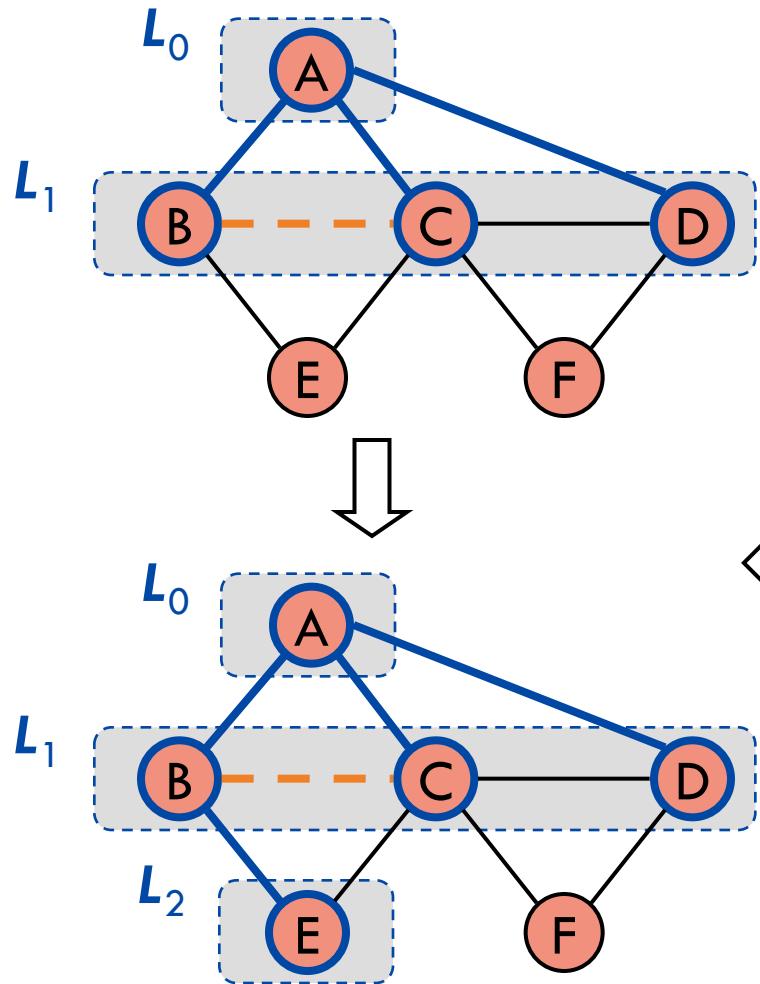
```
def BFS(G,s):  
  
    # set things up for BFS  
    for u in G.vertices():  
        seen[u] ← False  
        parent[u] ← None  
  
    seen[s] ← True  
    layers ← []  
    current ← [s]  
    next ← []  
  
    # process current layer  
    while not current.is_empty():  
        layers.append(current)  
        # iterate over current layer  
        for u in current:  
            for v in G.incident(u):  
                if not seen[v]:  
                    next.append(v)  
                    seen[v] ← True  
                    parent[v] ← u  
  
        # update current & next layers  
        current ← next  
        next ← []  
  
    return layers, parent
```



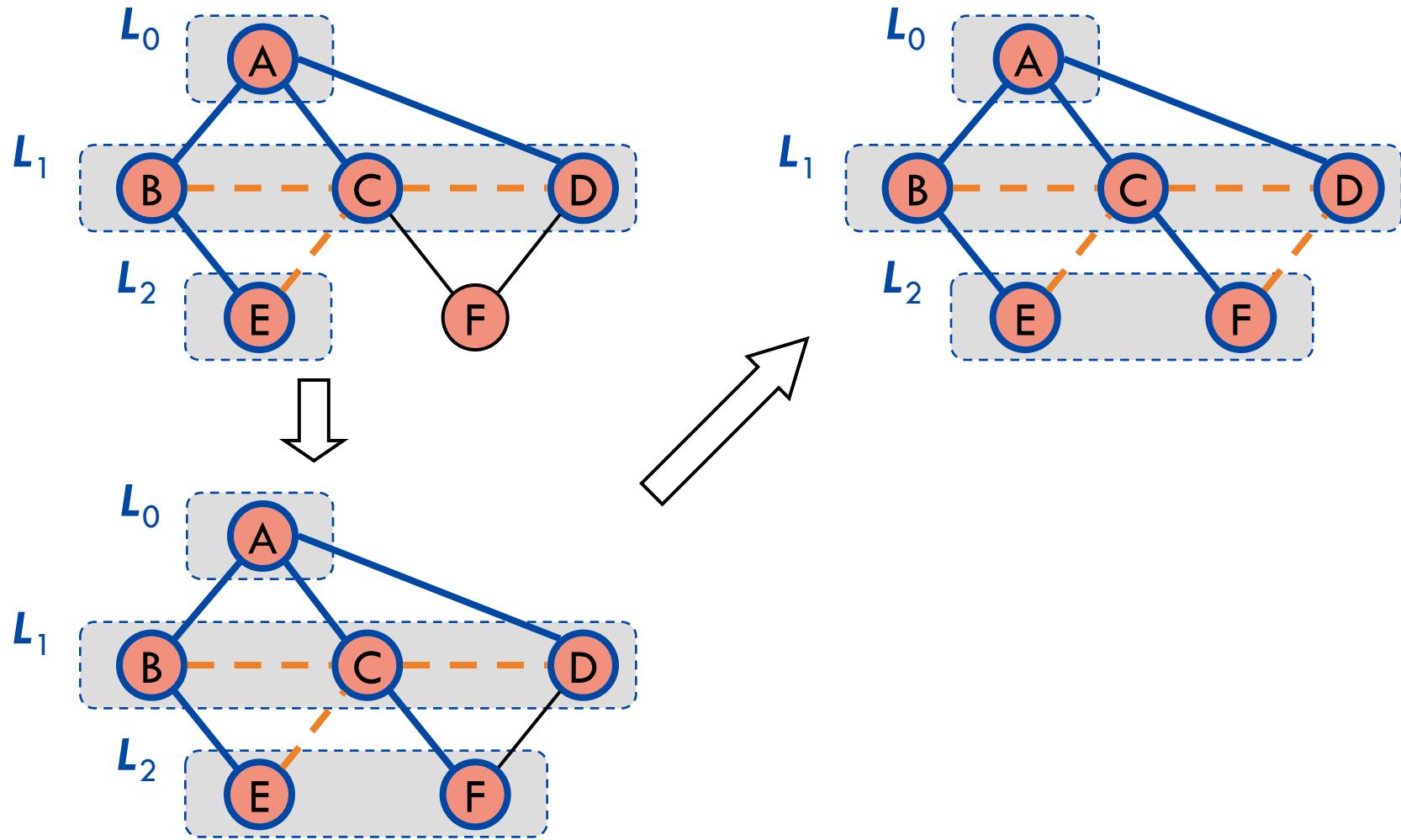
Example



Example (cont.)



Example (cont.)



Properties

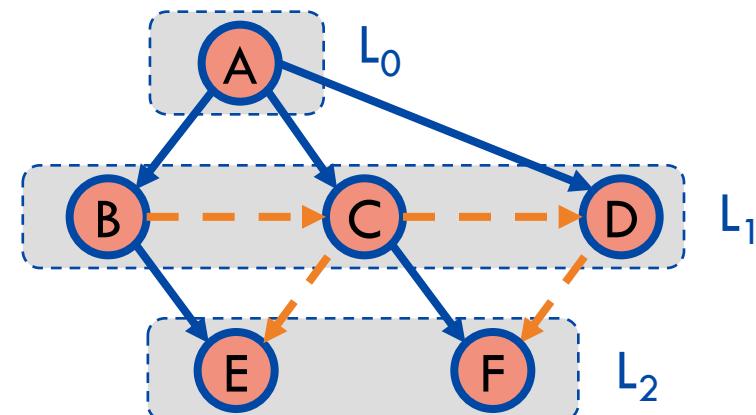
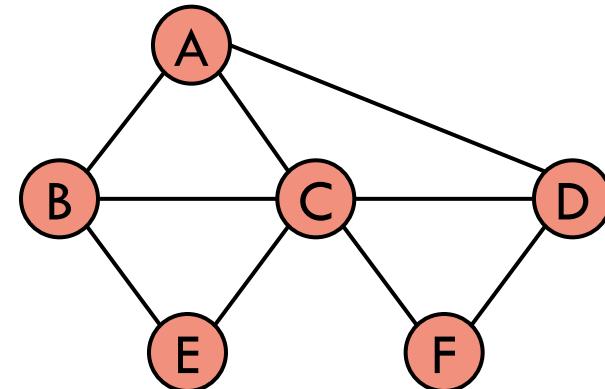
Let C_v be the connected component of v in our graph G

Fact: $\text{BFS}(G, s)$ visits all vertices in C_s

Fact: Edges $\{ (u, \text{parent}[u]): u \in C_s \}$ form a spanning tree T_s of C_s

Fact: For each v in L_i there is a path in T_s from s to v with i edges

Fact: For each v in L_i any path in G from s to v has at least i edges



BFS performance

```
def BFS(G, s):  
  
    # set things up for BFS  
    for u in G.vertices() do  
        seen[u] ← False  
        parent[u] ← None  
  
    seen[s] ← True  
    layers ← []  
    current ← [s]  
    next ← []  
  
    # process current layer  
    while not current.is_empty() do  
        layers.append(current)  
        # iterate over current layer  
        for u in current do  
            for v in G.incident(u) do  
                if not seen[v] then  
                    next.append(v)  
                    seen[v] ← True  
                    parent[v] ← u  
  
        # update curr and next layers  
        current ← next  
        next ← []  
  
    return layers
```

$\mathcal{O}(n)$ time

$\mathcal{O}(\sum_u \deg(u)) = \mathcal{O}(m)$ time

BFS performance

Fact: Assuming adjacency list representation we can perform a BFS traversal of a graph with n vertices and m edges in $O(n+m)$ time

Fact: Assuming adjacency matrix representation we can perform a BFS traversal of a graph with n vertices and m edges in $O(n^2)$ time

The additional attributes about the vertices (seen and parent) can be associated directly via Vertex class or we can use an external map data structure

BFS Applications

BFS can be used to solve other graph problems in $O(n + m)$ time:

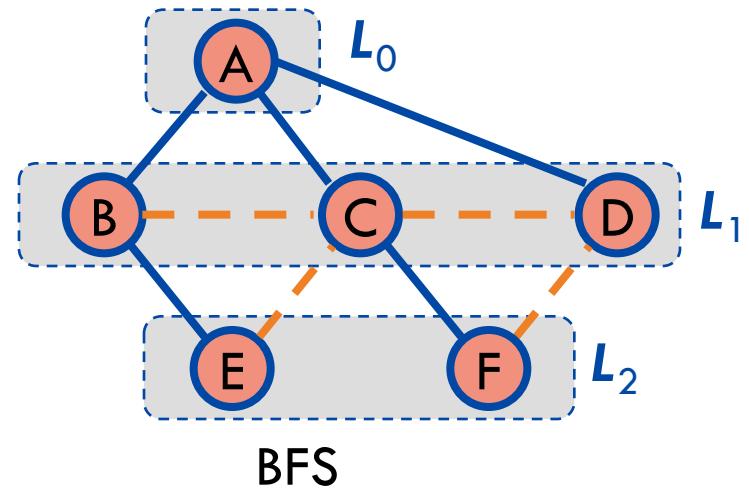
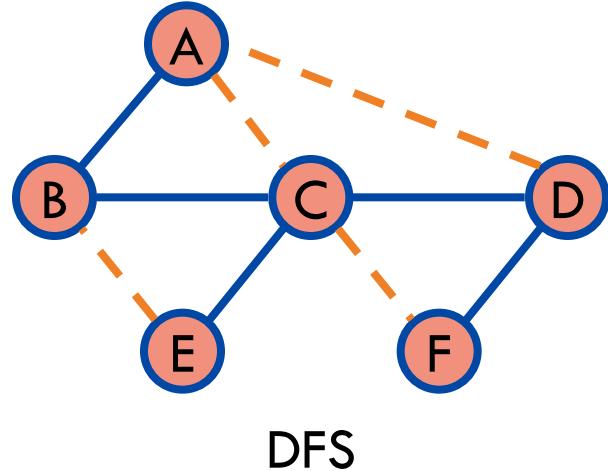
- Find a shortest path between two given vertices
- Find a cycle in the graph
- Test whether a graph is connected
- Compute a spanning tree of a graph (if connected)

And is the building block of more sophisticated algorithms:

- Testing if graph is bipartite

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	✓	✓
Shortest paths		✓
Biconnected components	✓	

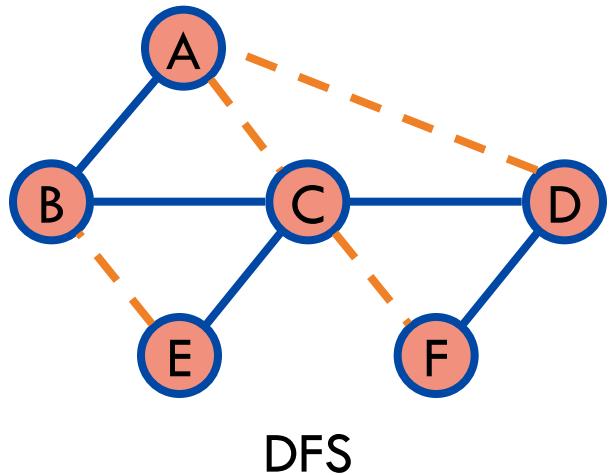


DFS vs. BFS (cont.)

Non-tree DFS edge (v, w)

w is an ancestor of v
in the DFS tree

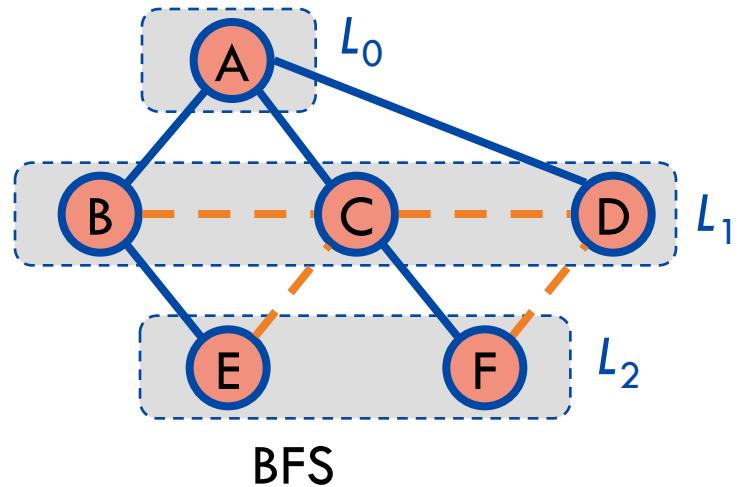
Called back edges



Non-tree BFS edge (v, w)

w is in the same level as v or
in the next level

Called cross edges



COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Data structures and Algorithms

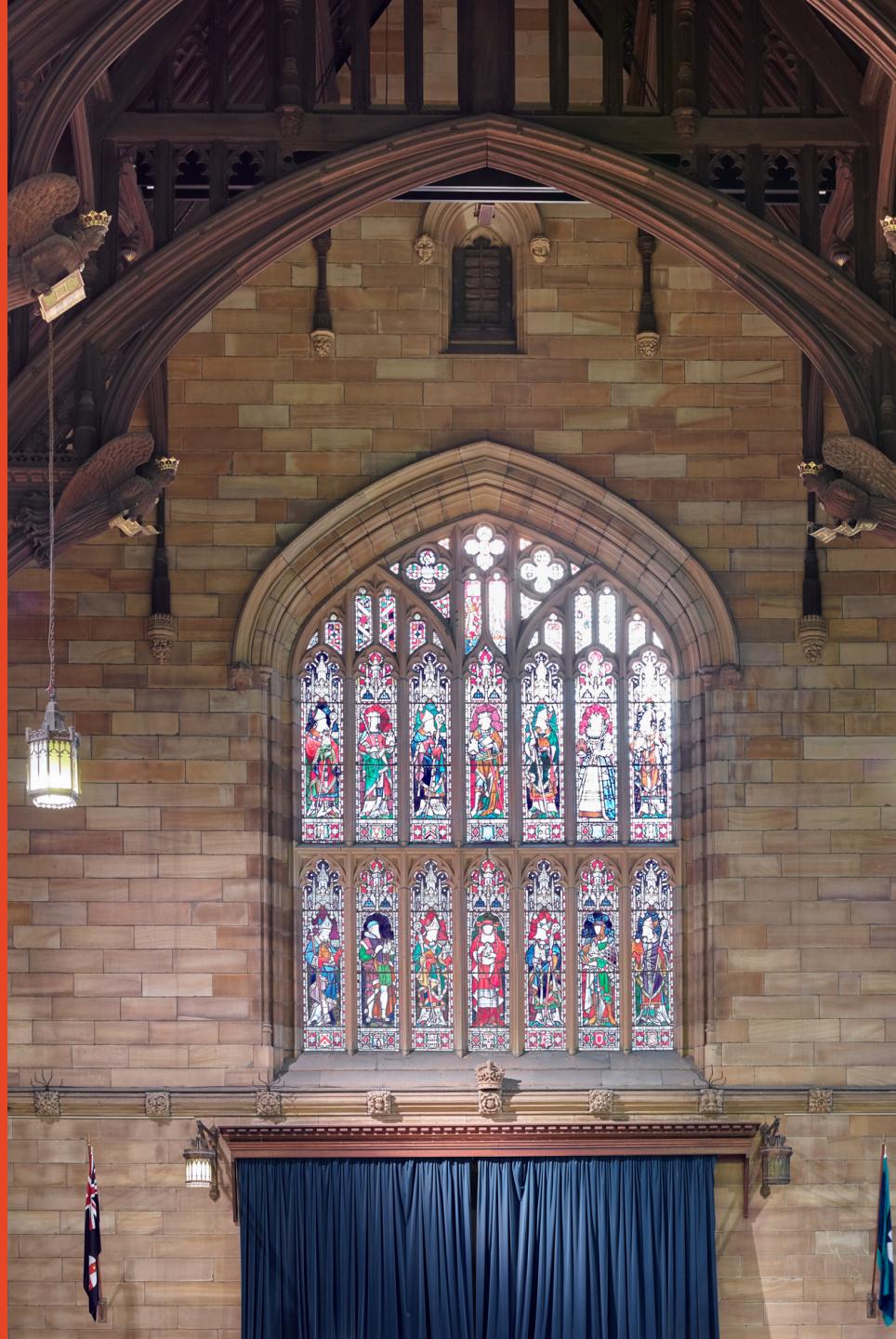
Lecture 8: Shortest Paths and Minimum Spanning Trees [GT 14.1-2, 15.1-3]

Dr. André van Renssen
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



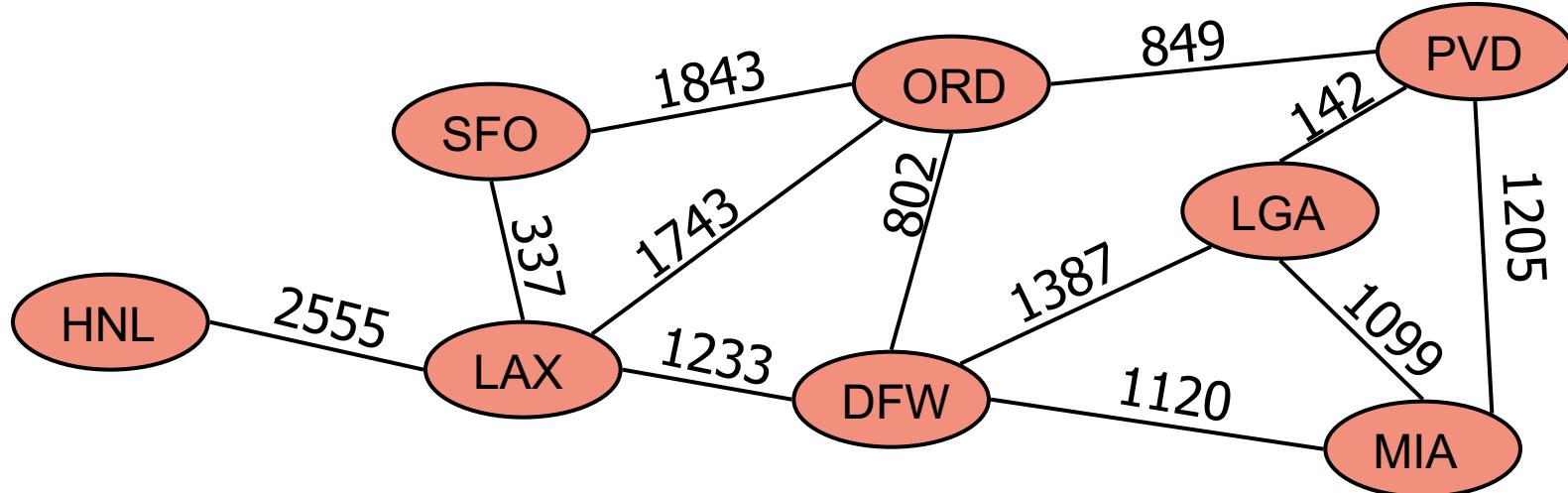
THE UNIVERSITY OF
SYDNEY



Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.

Example: In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports

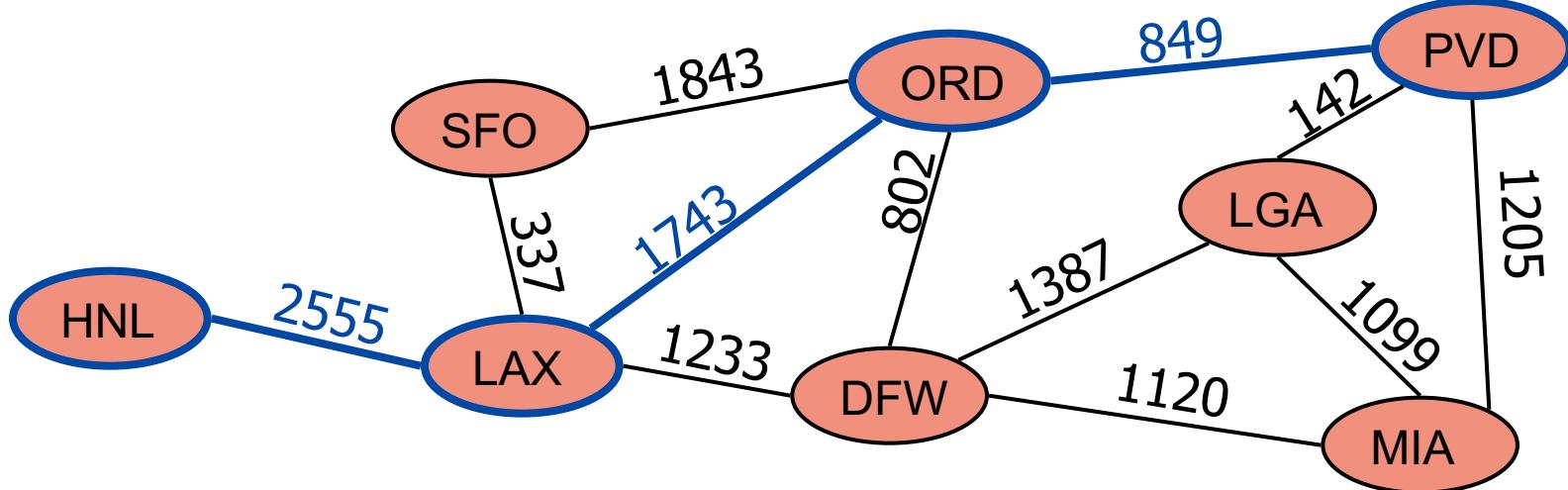


Shortest Paths

Given an edge weighted graph and two vertices u and v , we want to find a path of minimum total weight between u and v , where the weight of a path is the sum of the weights of its edges.

Applications: Internet packet routing, flight reservations and driving directions.

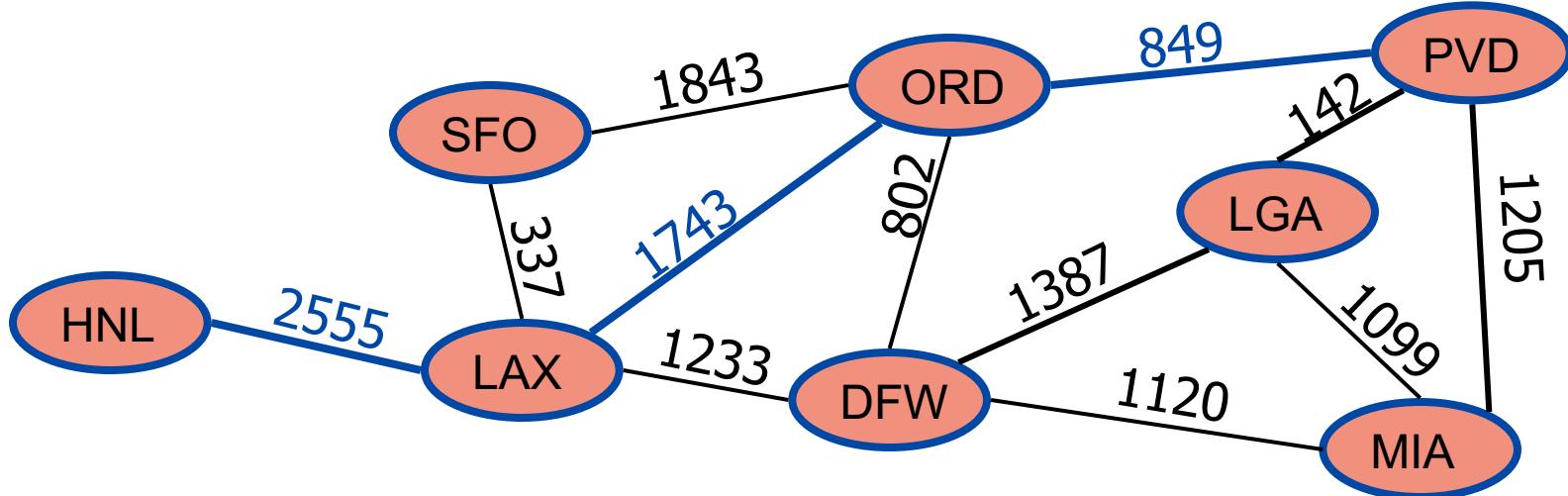
Example: Shortest path between Providence (PVD) and Honolulu (HNL)



Shortest Path Properties

Property: A subpath of a shortest path is itself a shortest path

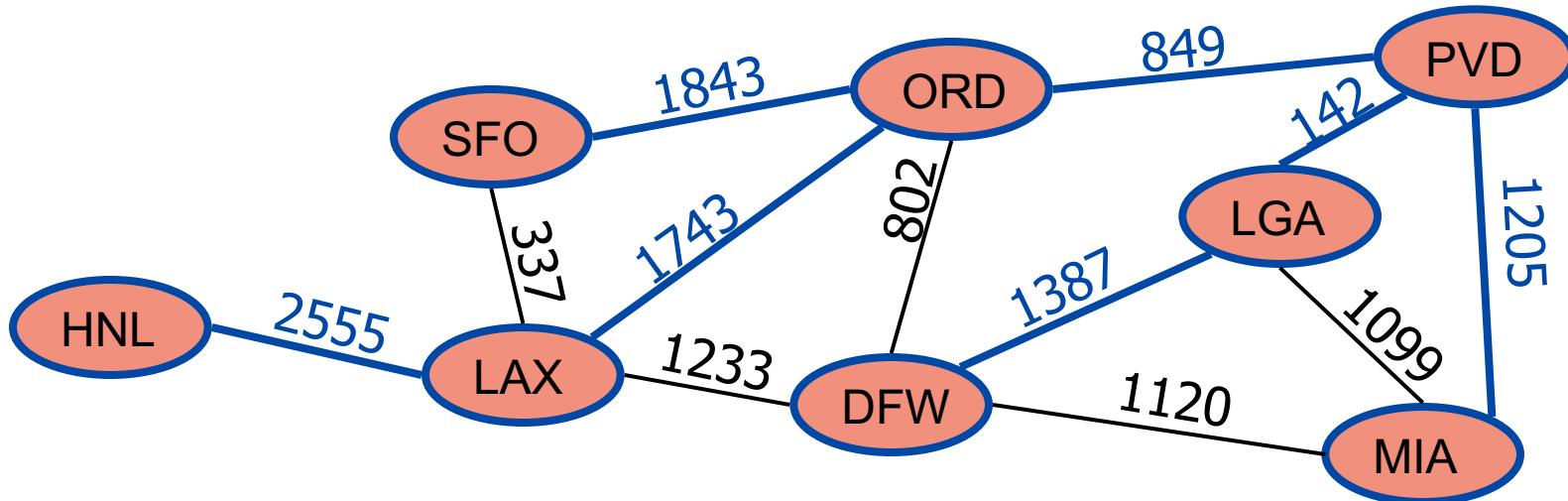
Example: Shortest path from Providence (PVD) to Honolulu (HNL)
also contains a shortest path from Providence (PVD) to
Los Angeles (LAX)



Shortest Path Properties

Property: There is a tree of shortest paths from a start vertex to all the other vertices (shortest path tree).

Example: Tree of shortest paths from Providence (PVD)



Dijkstra's Algorithm

Input:

- Graph $G = (V, E)$
- Edges weights $w : E \rightarrow \mathbb{R}_+$
- Start vertex s

Output:

- Distance from s to all v in V
- Shortest path tree rooted at s

Assumptions:

- G is connected and undirected
- edge weights are nonnegative

High level idea:

- Maintain a distance estimate
 $D[v] \geq \text{dist}_w(s, v)$ for all v in V
- Keep track of a subset S of V s.t.
 $D[v] = \text{dist}_w(s, v)$ for all v in S

Initially:

- $D[s] = 0$
- $D[v] = \infty$ for all v in $V - s$

In each iteration we:

- add to S vertex u in $V \setminus S$ with smallest $D[u]$
- update D -values for vertices adjacent to u

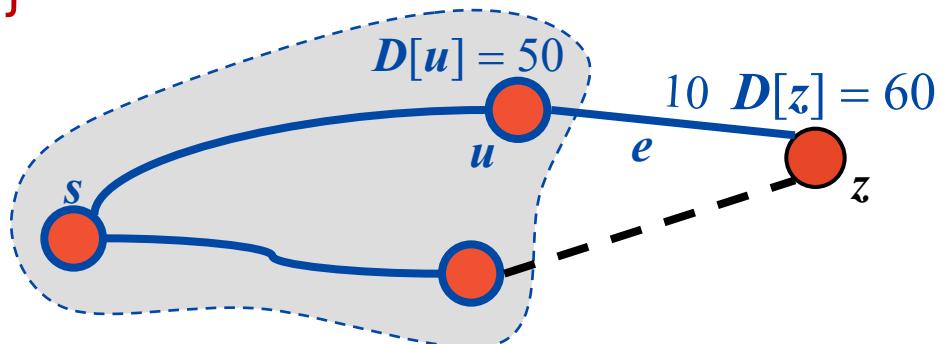
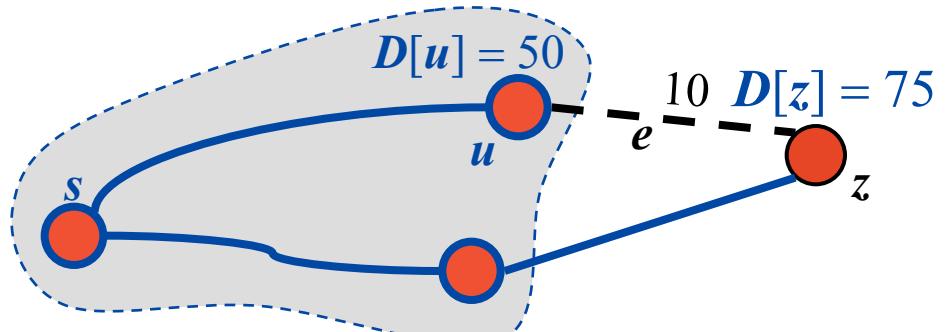
Edge Relaxation

Consider edge $e = (u, z)$ such that:

- u is the last vertex added to S
- z is not in S

The relaxation of edge (u, z) updates $D[z]$ as follows:

$$D[z] \leftarrow \min\{D[z], D[u] + w(u, z)\}$$



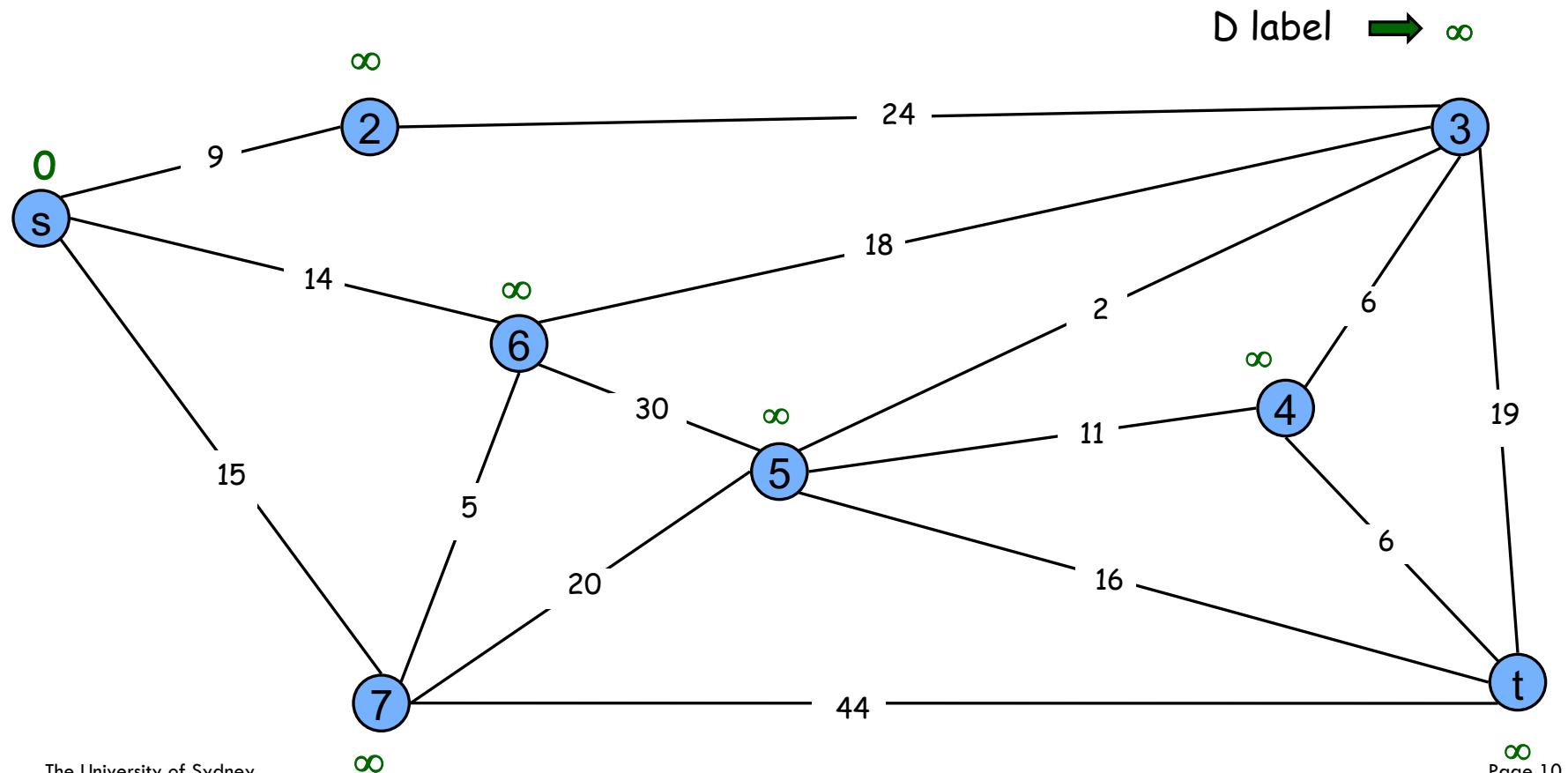
Dijkstra's Algorithm pseudocode

```
def Dijkstra(G, w, s):  
  
    # initialize algorithm  
    for v in V do  
        D[v] ← ∞  
        parent[v] ← Ø  
    D[s] ← 0  
    Q ← new priority queue for { (v, D[v]) : v in V }  
  
    # iteratively add vertices to S  
    while Q is not empty do  
        u ← Q.remove_min()  
        for z in G.neighbors(u) do  
            if D[u] + w[u, z] < D[z] then  
                D[z] ← D[u] + w[u, z]  
                Q.update_priority(z, D[z])  
                parent[z] ← u  
    return D, parent
```

Dijkstra's Shortest Path Algorithm

$S = \{ \}$

$PQ = \{ s, 2, 3, 4, 5, 6, 7, t \}$

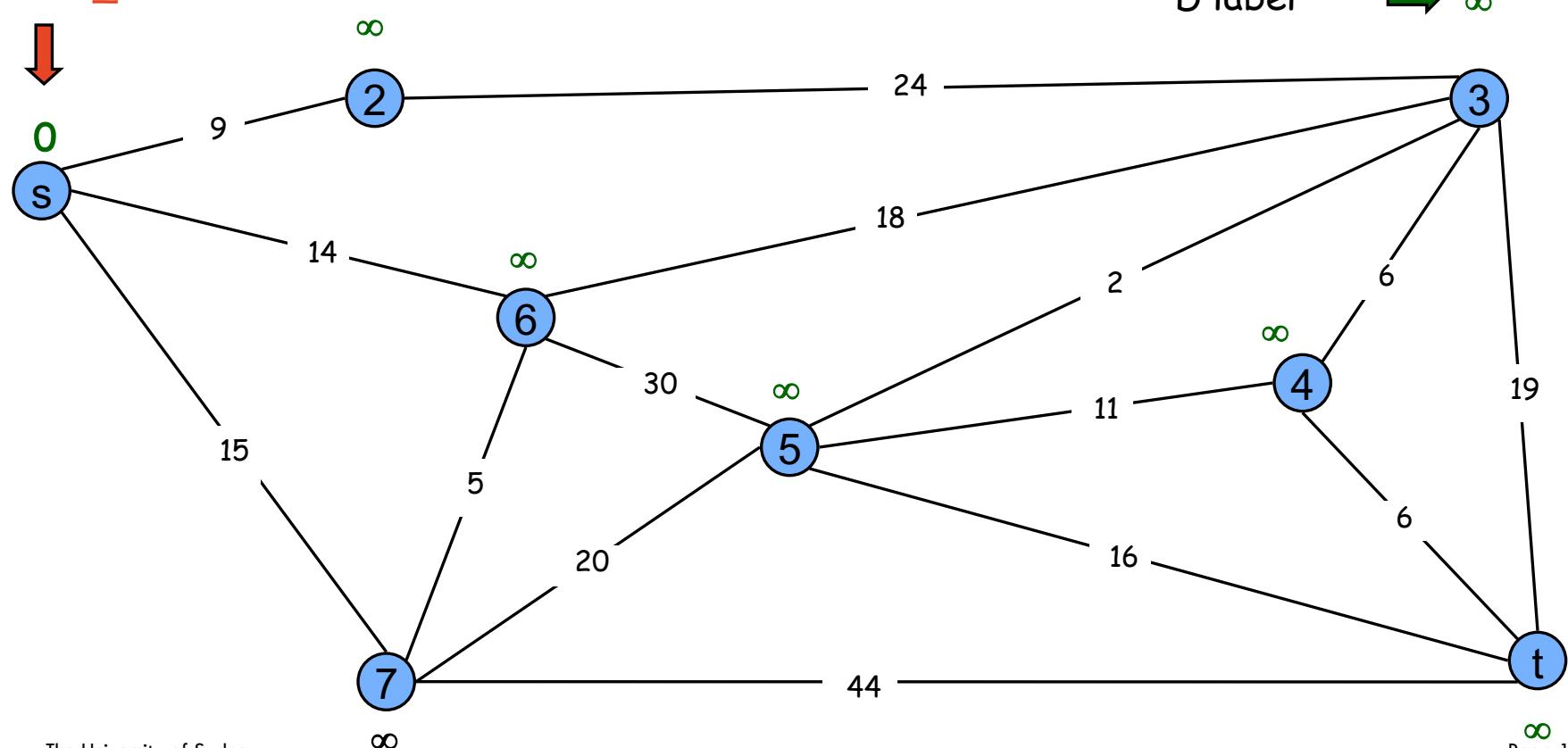


Dijkstra's Shortest Path Algorithm

$S = \{ \}$

$PQ = \{ s, 2, 3, 4, 5, 6, 7, t \}$

remove_min

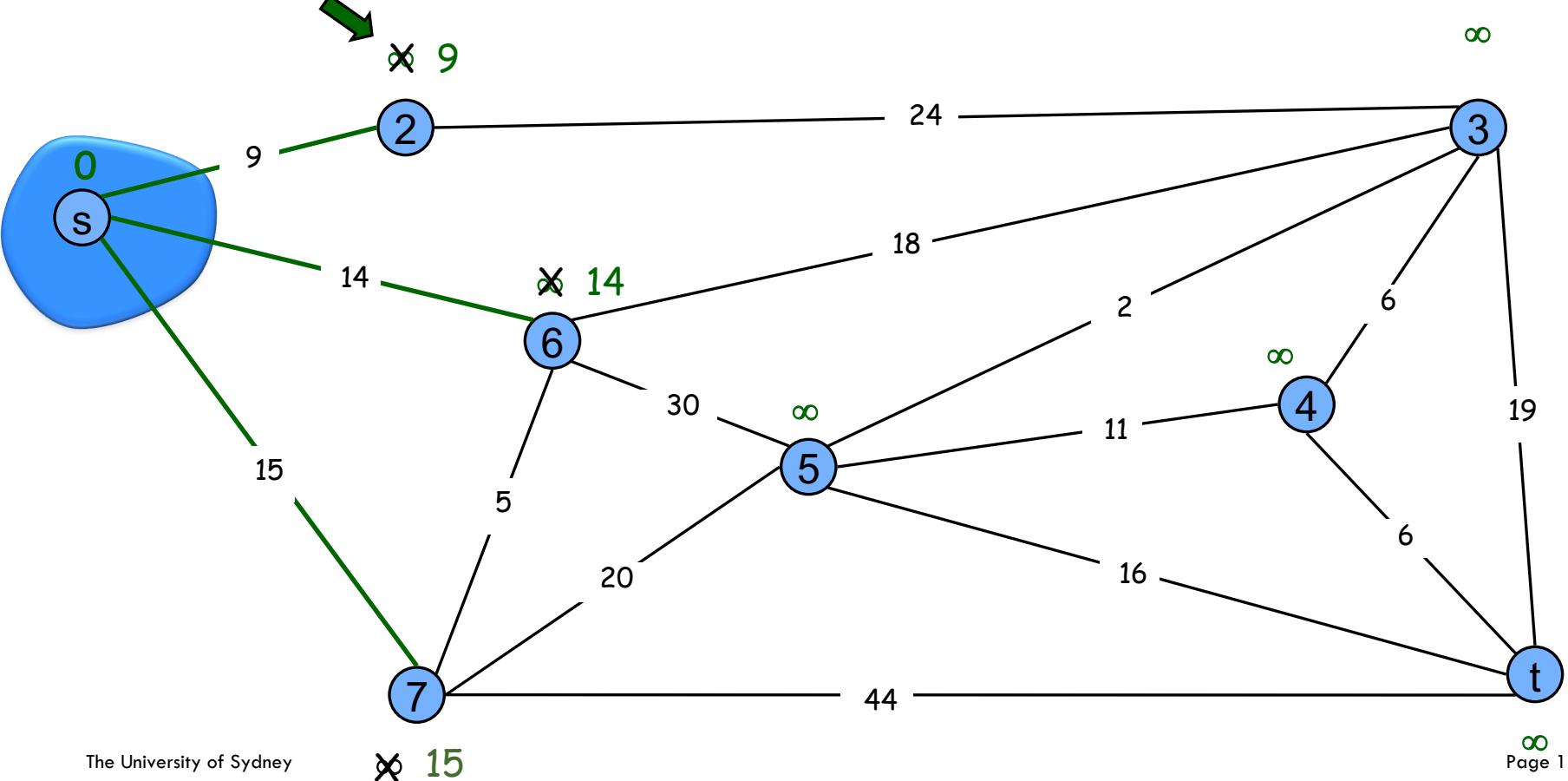


Dijkstra's Shortest Path Algorithm

$S = \{ s \}$

$PQ = \{ 2, 3, 4, 5, 6, 7, t \}$

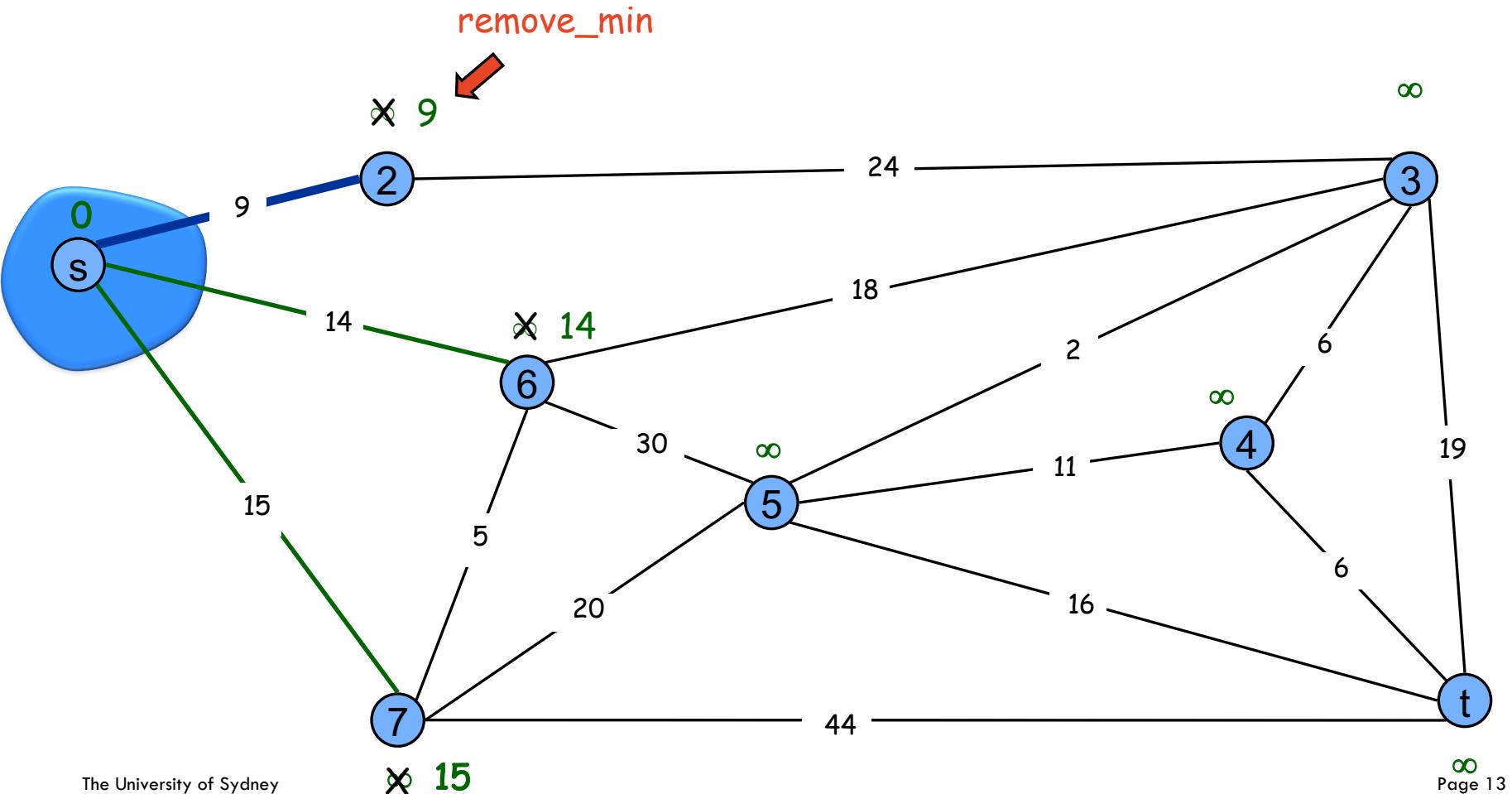
decrease key



Dijkstra's Shortest Path Algorithm

$S = \{ s \}$

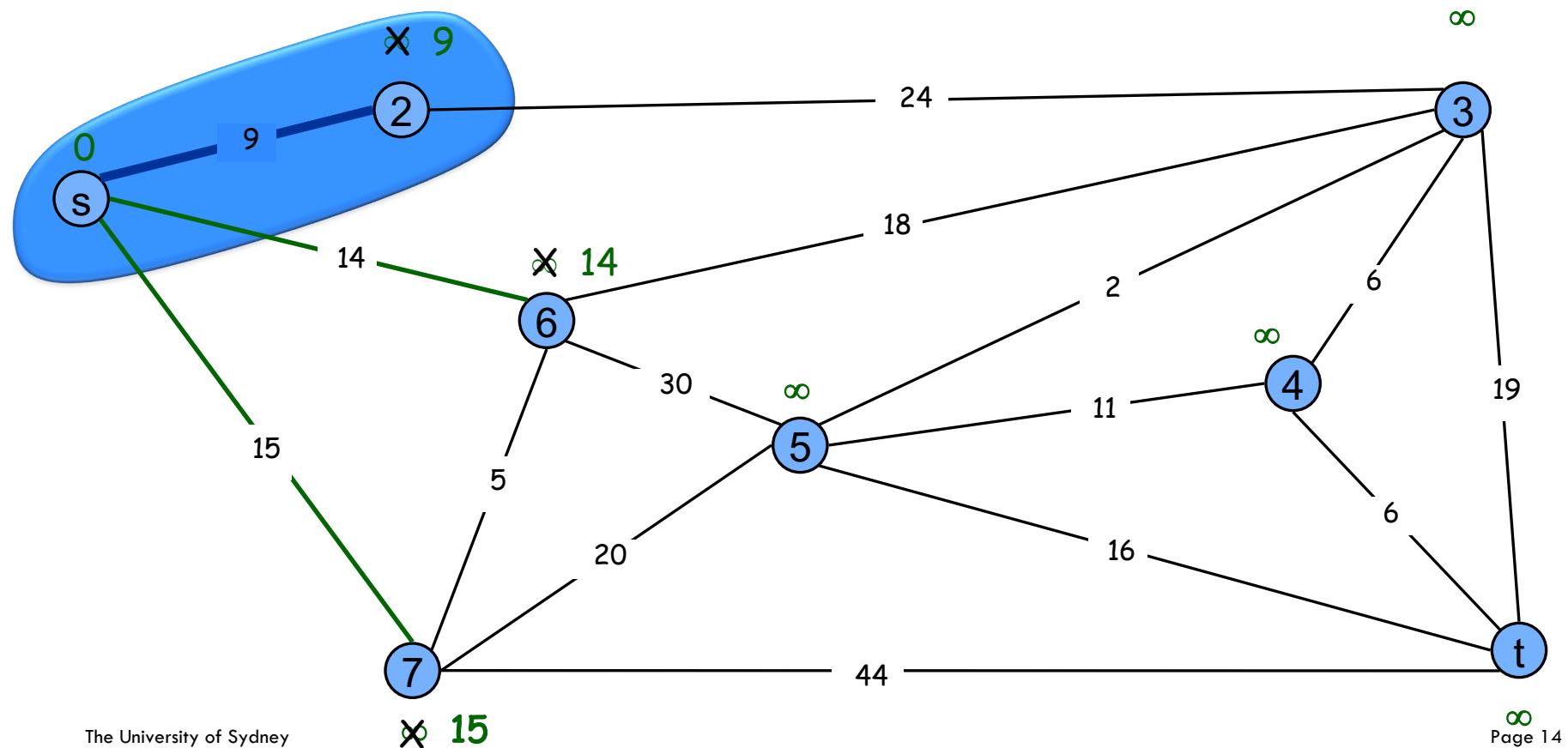
$PQ = \{ 2, 3, 4, 5, 6, 7, t \}$



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2 \}$

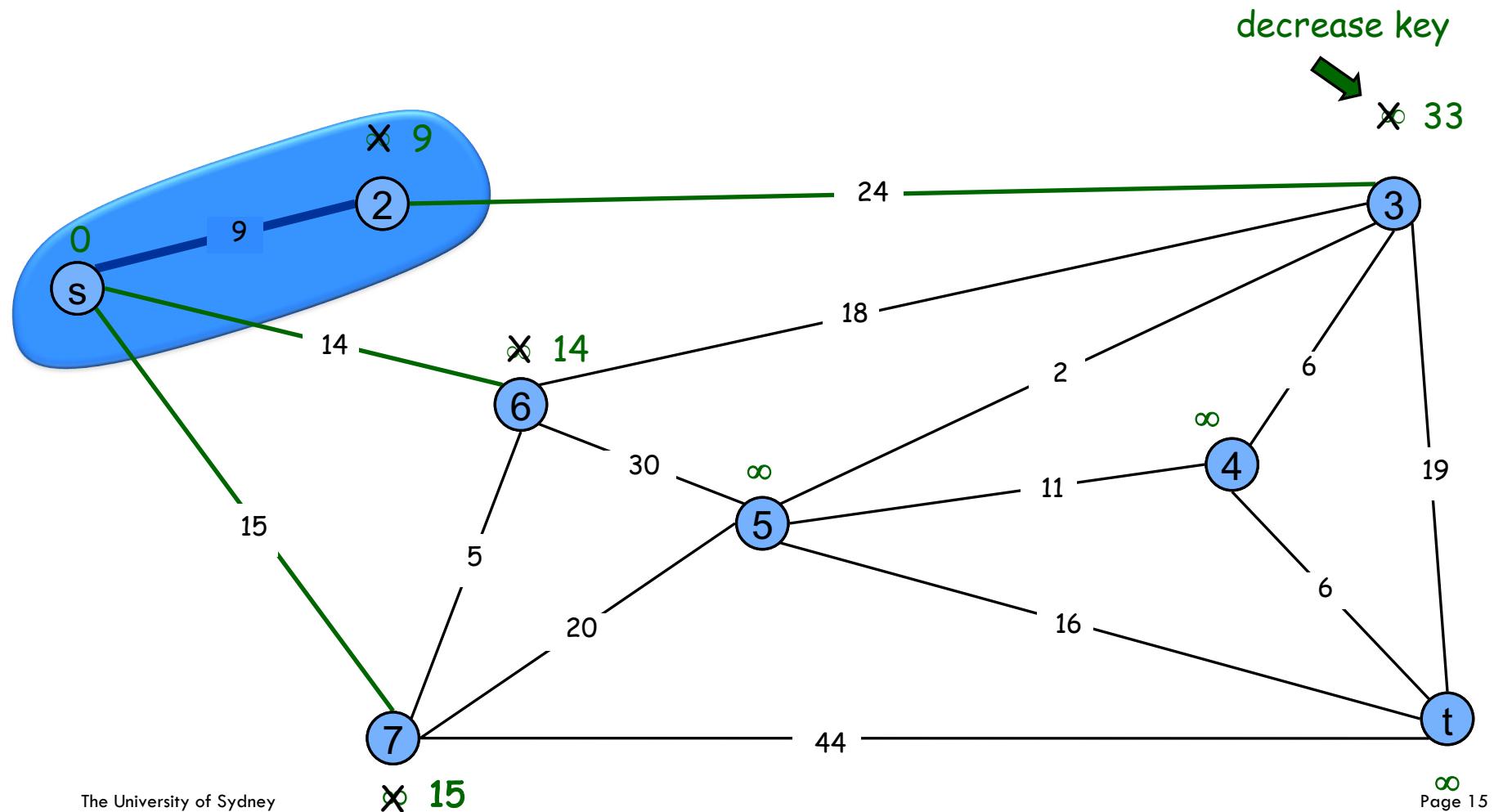
$PQ = \{ 3, 4, 5, 6, 7, t \}$



Dijkstra's Shortest Path Algorithm

$$S = \{ s, 2 \}$$

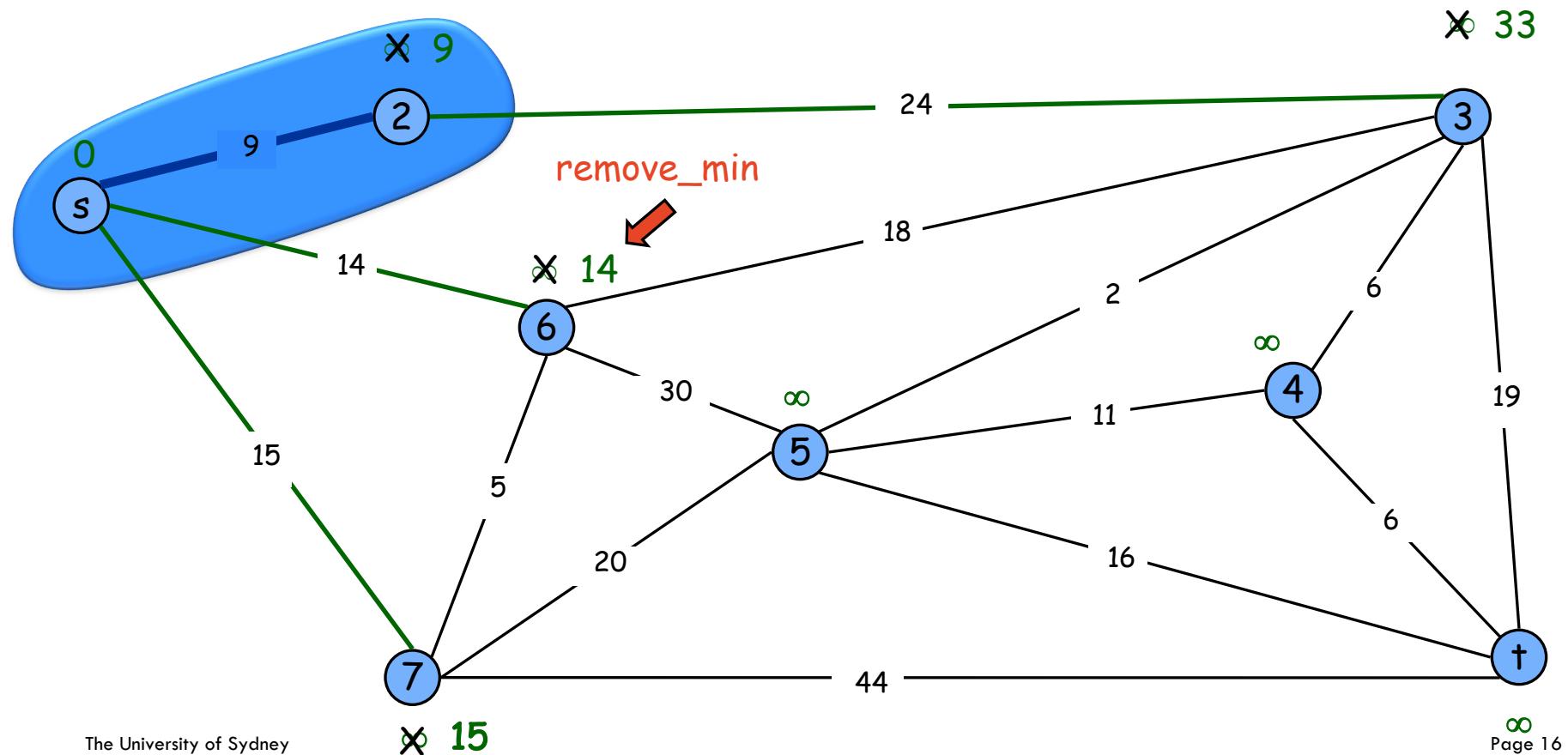
$$PQ = \{ 3, 4, 5, 6, 7, t \}$$



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2 \}$

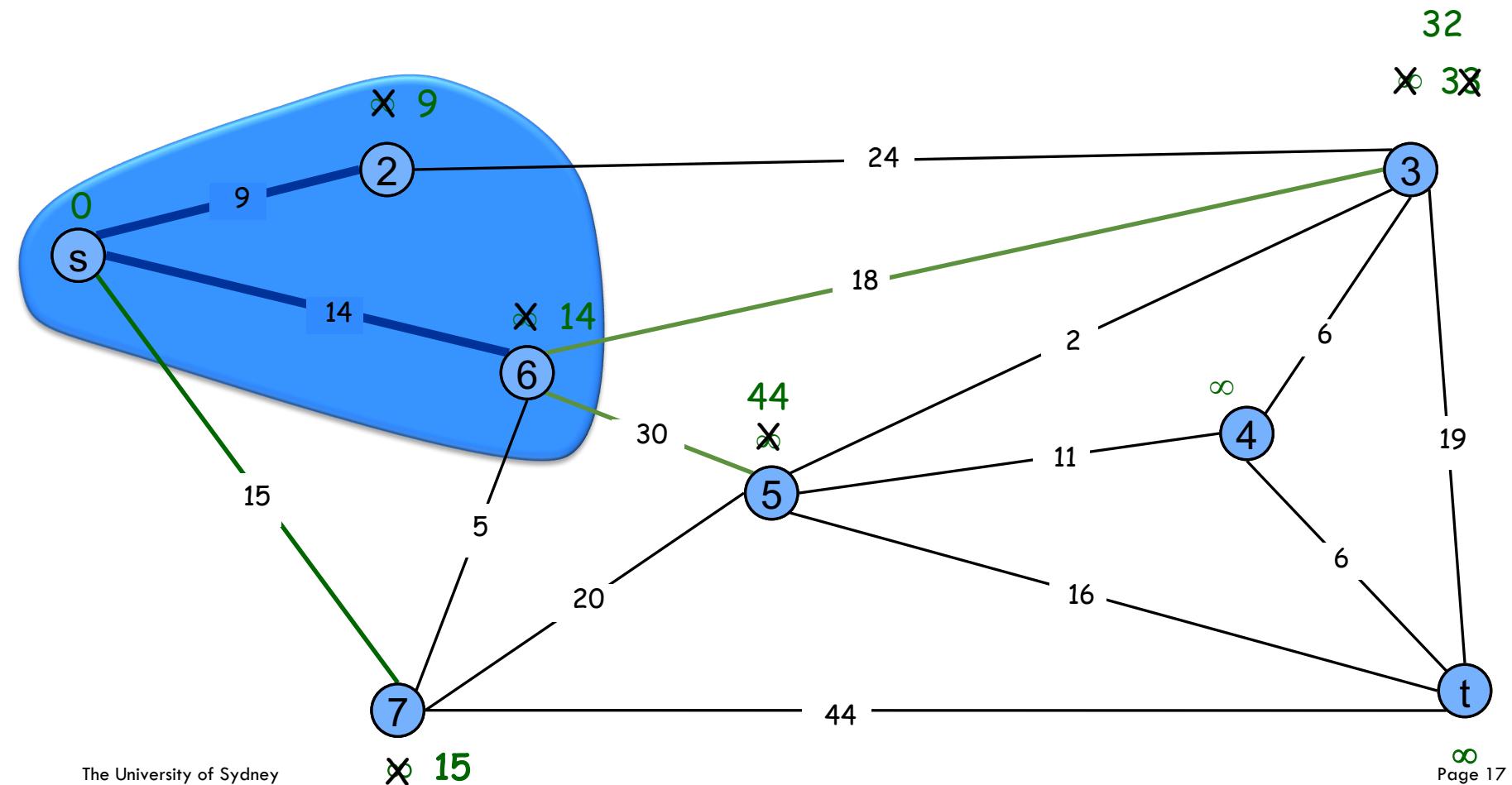
$PQ = \{ 3, 4, 5, 6, 7, t \}$



Dijkstra's Shortest Path Algorithm

$$S = \{ s, 2, 6 \}$$

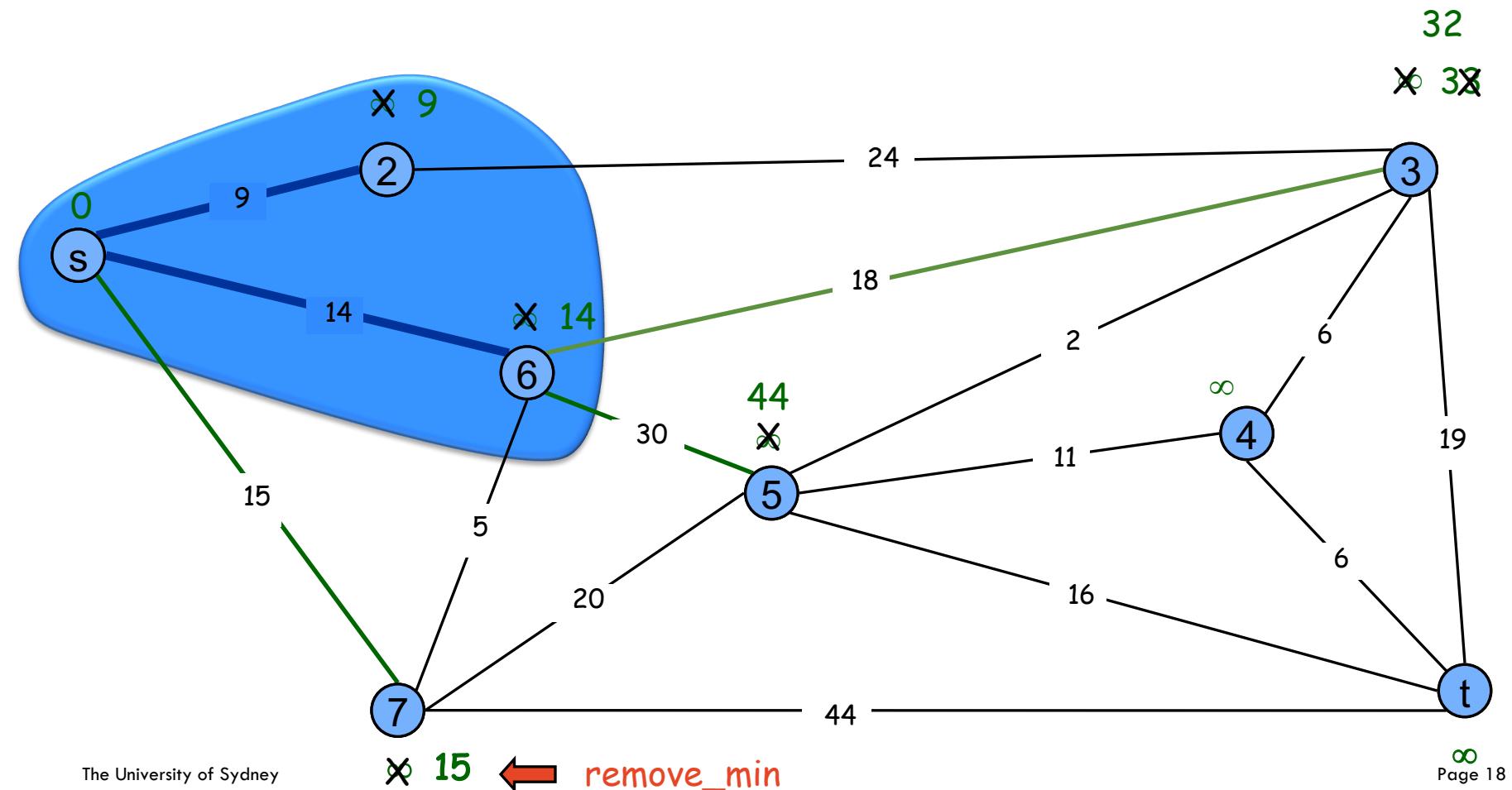
$$PQ = \{ 3, 4, 5, 7, t \}$$



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 6 \}$

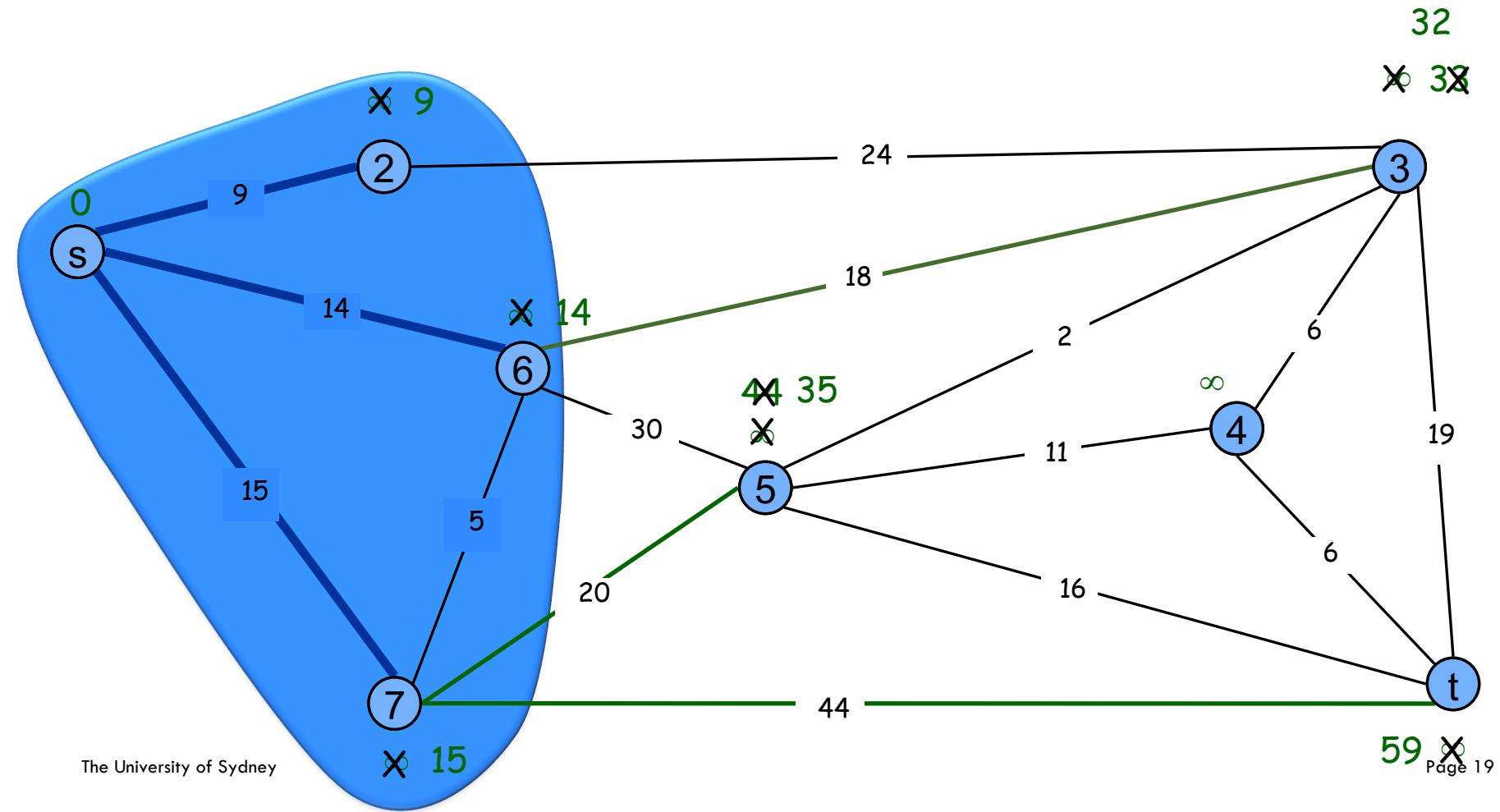
$PQ = \{ 3, 4, 5, 7, t \}$



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 6, 7 \}$

$PQ = \{ 3, 4, 5, t \}$



Dijkstra's Shortest Path Algorithm

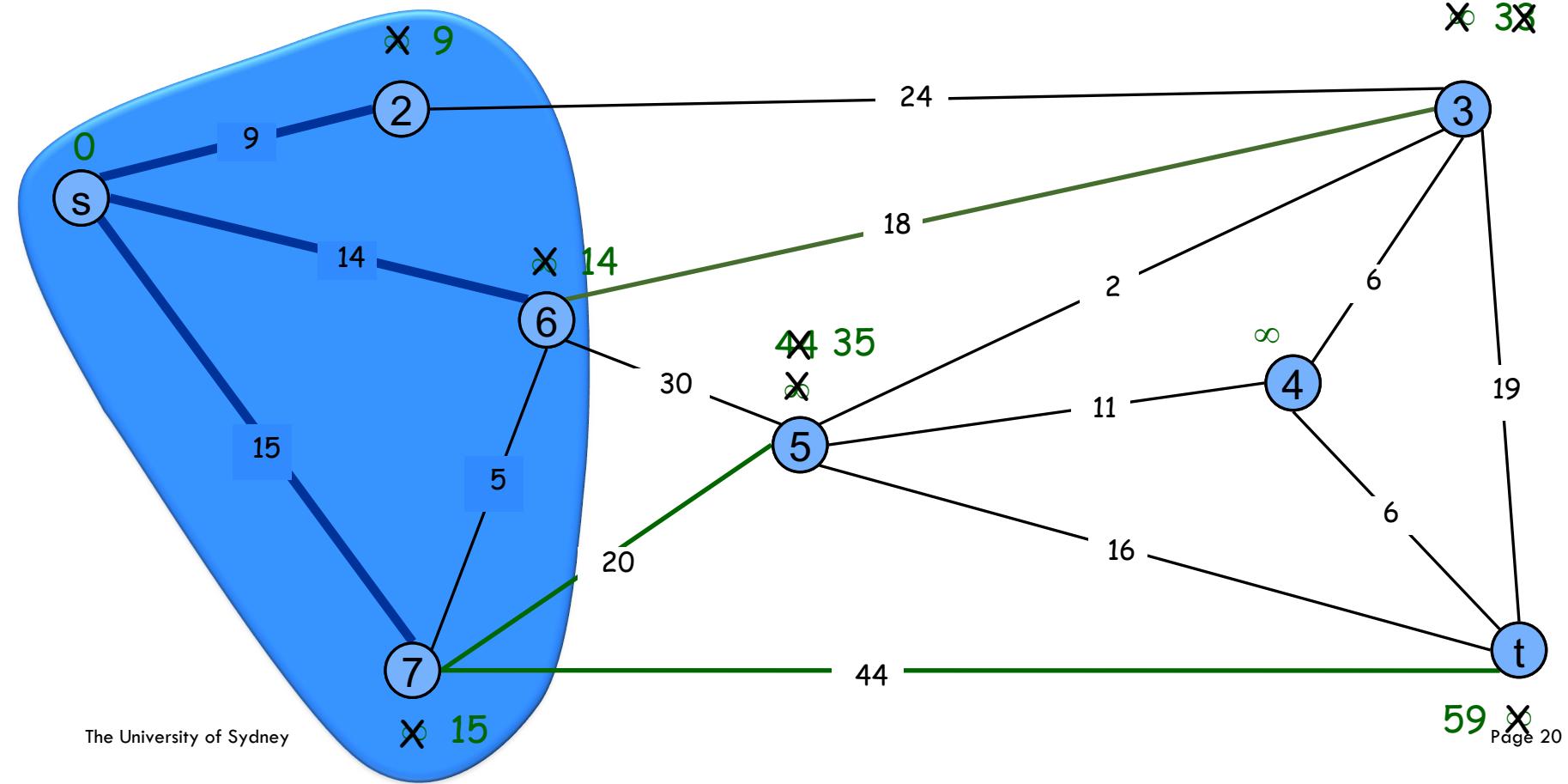
$S = \{ s, 2, 6, 7 \}$

$PQ = \{ 3, 4, 5, t \}$

remove_min

32

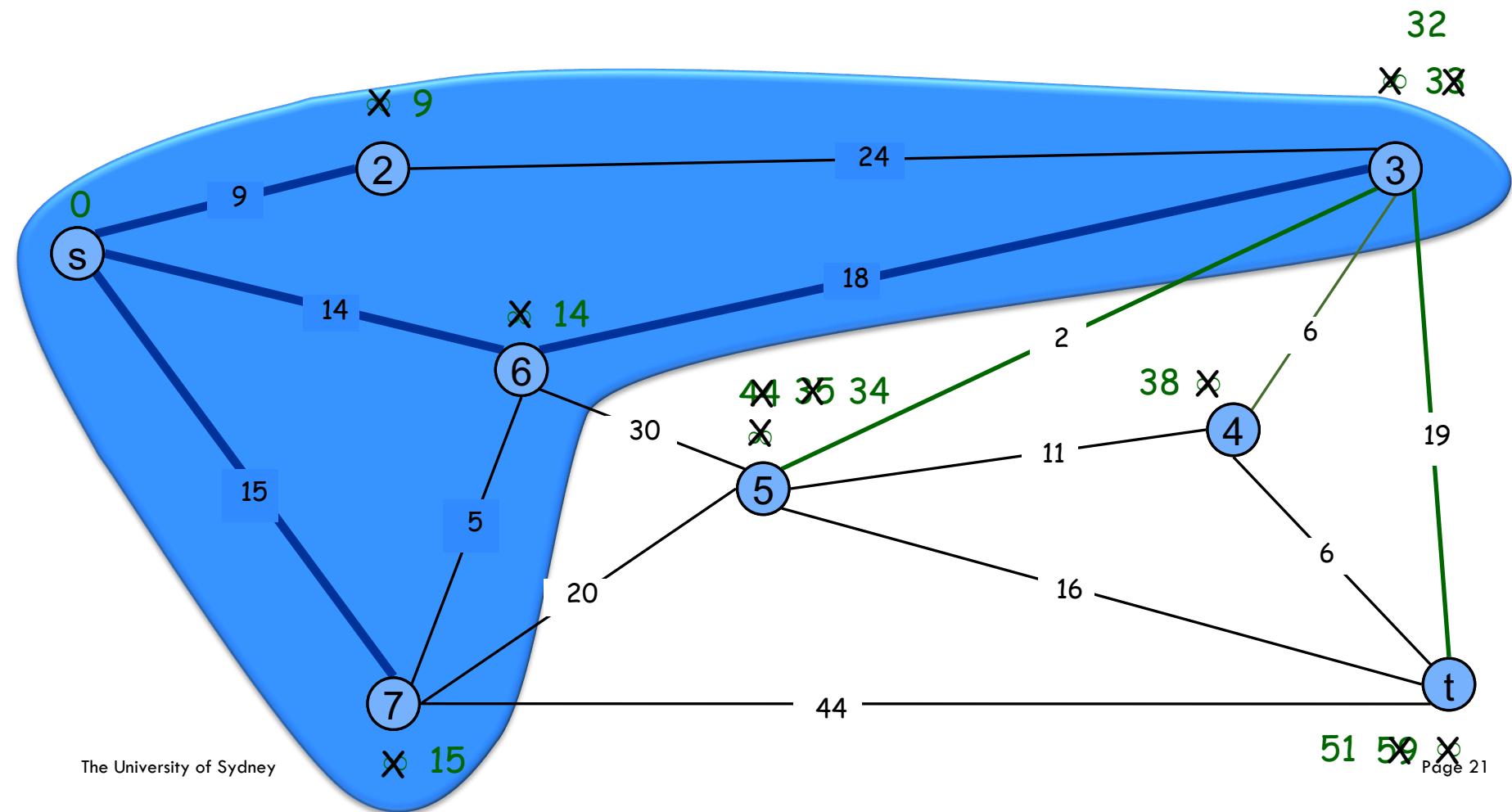
38



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 6, 7 \}$

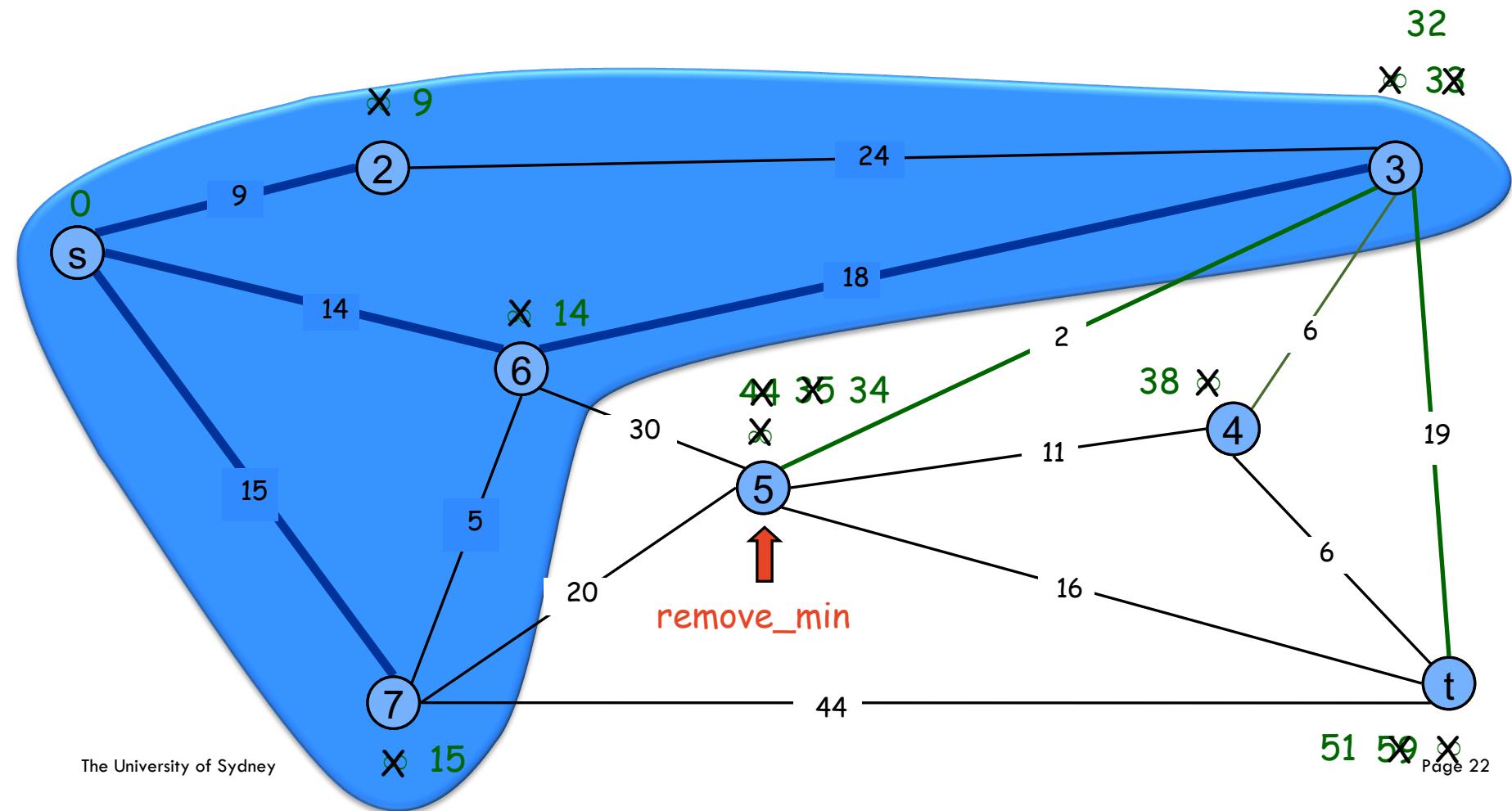
$PQ = \{ 4, 5, t \}$



Dijkstra's Shortest Path Algorithm

$$S = \{ s, 2, 3, 6, 7 \}$$

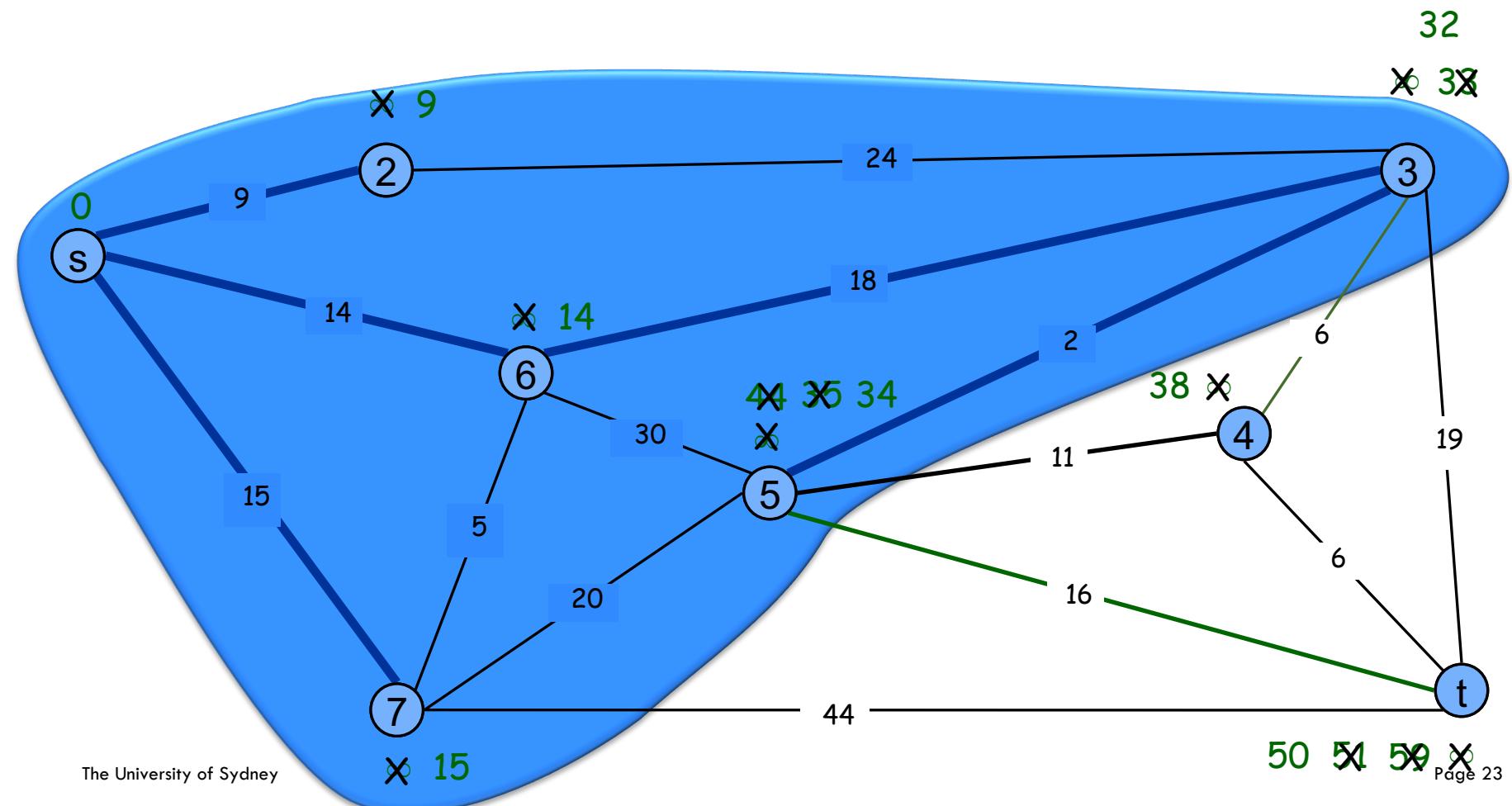
$$PQ = \{ 4, 5, t \}$$



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 5, 6, 7 \}$

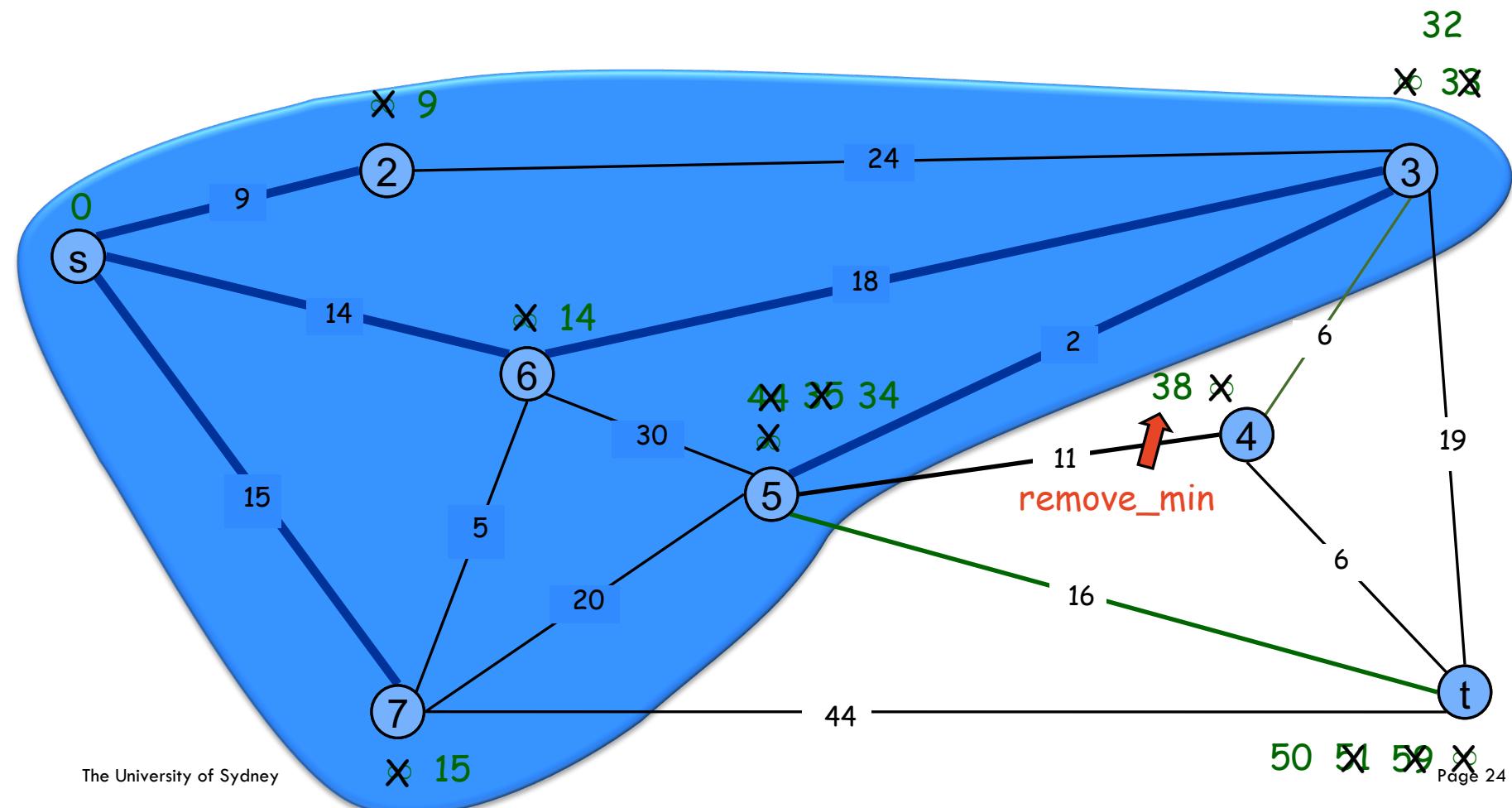
$PQ = \{ 4, t \}$



Dijkstra's Shortest Path Algorithm

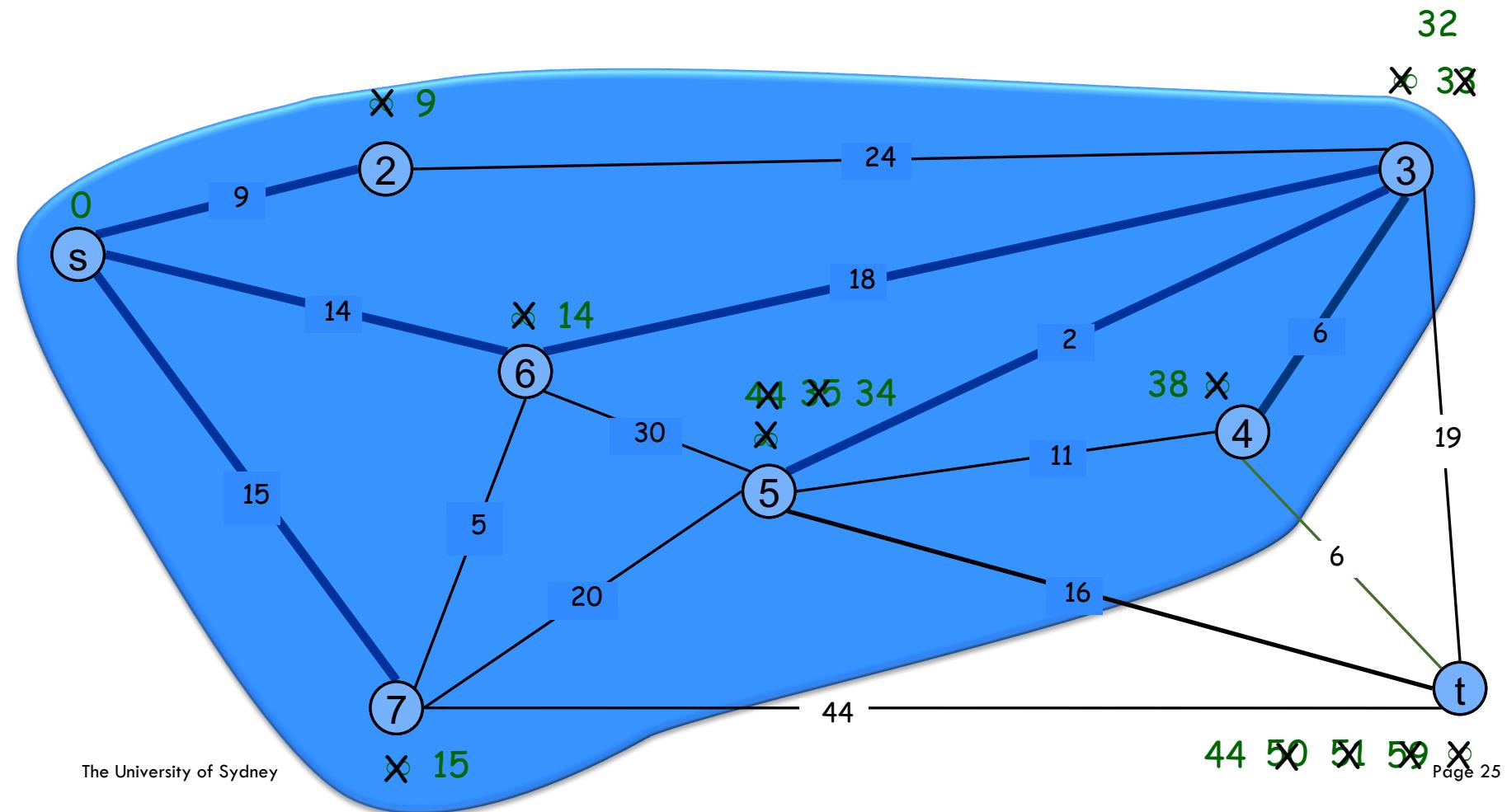
$$S = \{ s, 2, 3, 5, 6, 7 \}$$

$$PQ = \{ 4, t \}$$



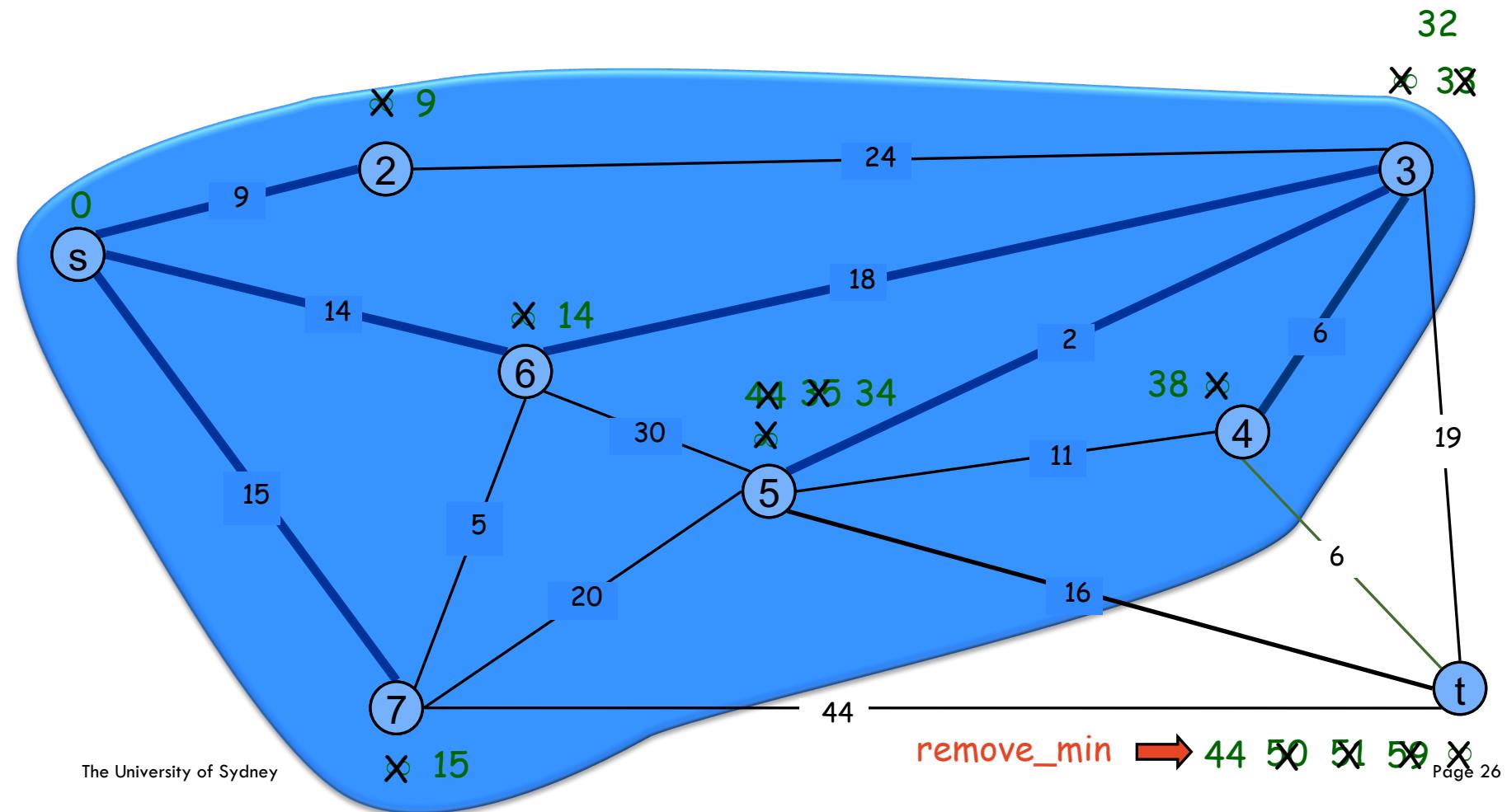
Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 4, 5, 6, 7 \}$
 $PQ = \{ t \}$



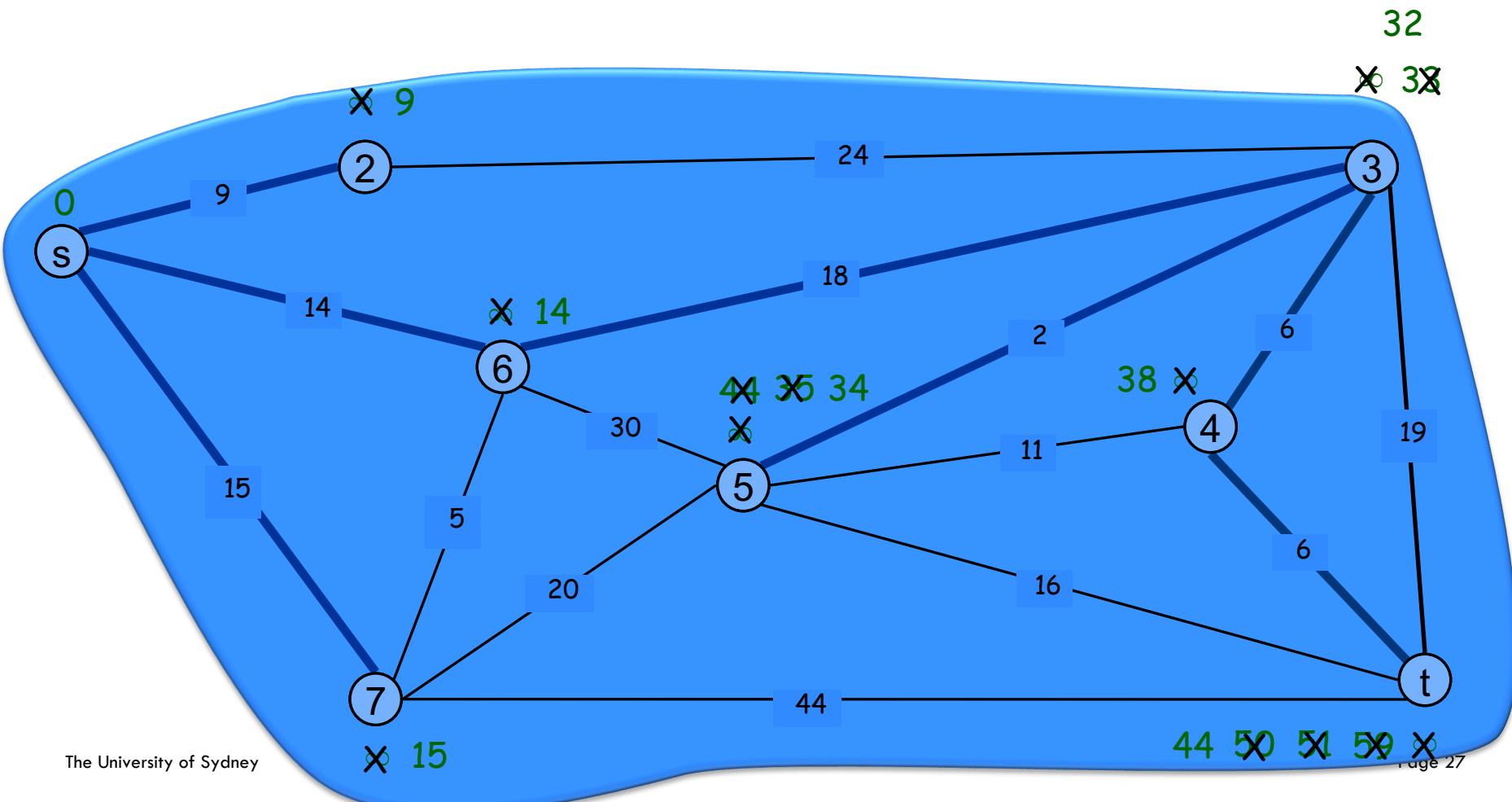
Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 4, 5, 6, 7 \}$
 $PQ = \{ t \}$



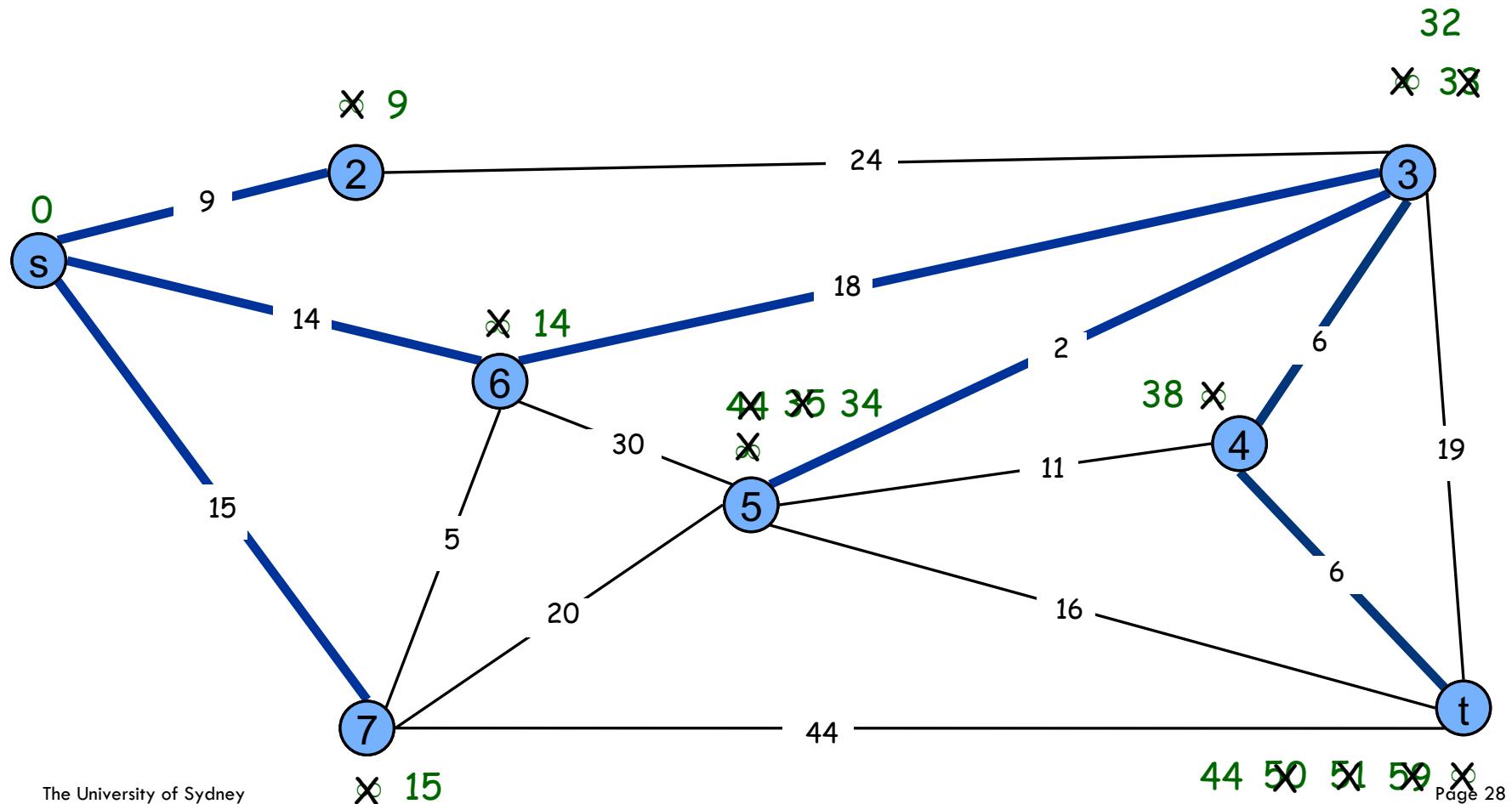
Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 4, 5, 6, 7, t \}$
 $PQ = \{ \}$



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 4, 5, 6, 7, t \}$
 $PQ = \{ \}$



Dijkstra complexity analysis except PQ ops

```
def Dijkstra(G, w, s):
```

```
# initialize algorithm
```

```
for v in V do
    D[v] ← ∞
    parent[v] ← ∅
D[s] ← 0
```

}

$O(n)$

```
Q ← new priority queue for { (v, D[v]) : v in V }
```

```
# iteratively add vertices to S
```

```
while Q is not empty do
```

```
    u ← Q.remove_min()
```

```
    for z in G.neighbors(u) do
```

```
        if D[u] + w[u, z] < D[z] then
```

```
            D[z] ← D[u] + w[u, z]
```

```
            Q.update_priority(z, D[z])
```

```
            parent[z] ← u
```

```
return D, parent
```

}

$O(\deg(u))$ for each u in V
plus update_priority work

Dijkstra's Algorithm complexity analysis

Assuming the graph is connected (so $m \geq n-1$), the algorithm spends $O(m)$ time on everything except PQ operations

Priority queue operation counts:

- insert: n
- decrease_key: m
- remove_min: n

Fact: Using a heap for PQ, Dijkstra runs in $O(m \log n)$ time

Fibonacci heap is a PQ that can carry out decrease key in $O(1)$ amortized time. Using that instead we get $O(m + n \log n)$ time.

Dijkstra's Algorithm Correctness

Invariant: For each $u \in S = V \setminus Q$, we have $D[u] = \text{dist}_w(s, u)$

Proof: (by induction on $|S|$)

Base case: $|S| = 1$ is trivial since $D[s] = 0$

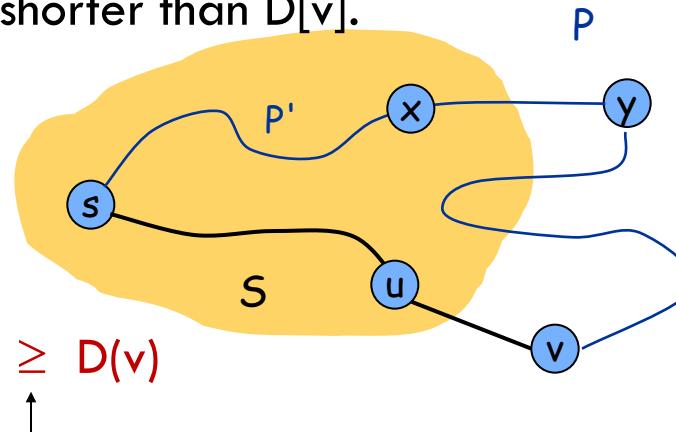
Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S and $u = \text{parent}[v]$
- The shortest $s-u$ path plus (u, v) is an $s-v$ path of length $D[v]$
- Consider any $s-v$ path P . We'll see that it's no shorter than $D[v]$.
- Let $x-y$ be the first edge in P that leaves S , and let P' be the subpath to x .
- P is already too long as soon as it leaves S :

$$w(P) \geq w(P') + w(x, y) \geq D(x) + w(x, y) \geq D(y) \geq D(v)$$

↑
inductive
hypothesis

↑
Def of
 $D(y)$
Dijkstra chose v
instead of y



Dijkstra's Algorithm Correctness

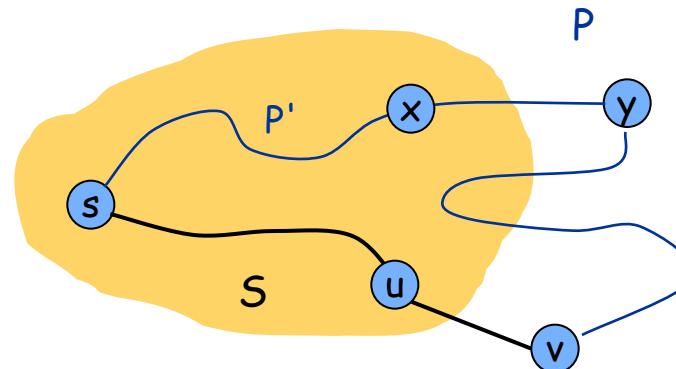
Invariant: The set $\{ (u, \text{parent}[u]) : u \text{ in } S \setminus \{ s \} \}$ forms a shortest path tree from s to every node in S

Proof: (by induction on $|S|$)

Base case: $|S| = 1$ is trivial since tree is empty

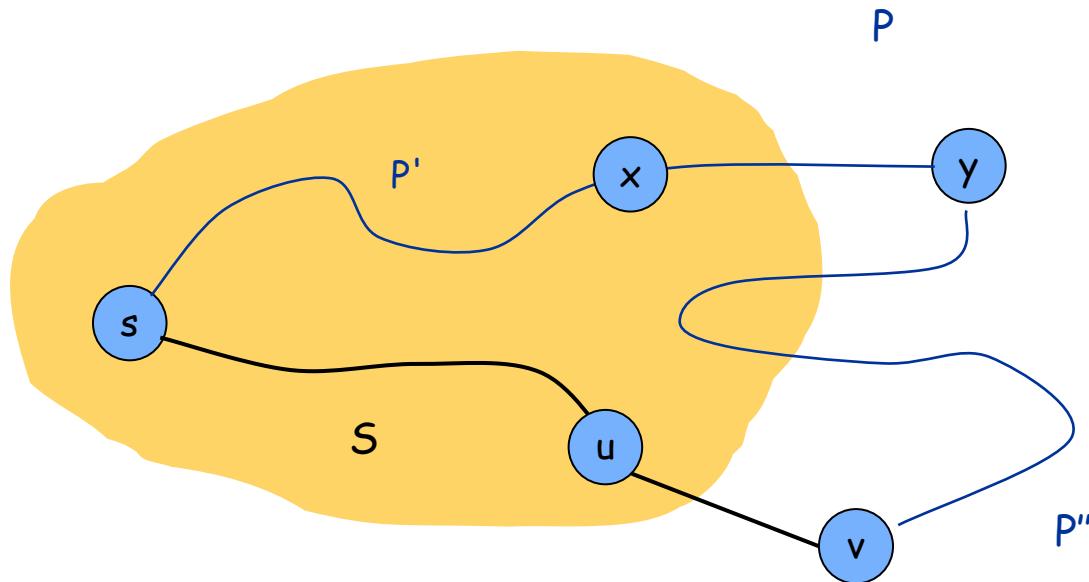
Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S and $u = \text{parent}[v]$
- $\text{dist}_w(s, v) = \text{dist}_w(s, u) + w[u, v]$
- Adding v to S we still have the invariants



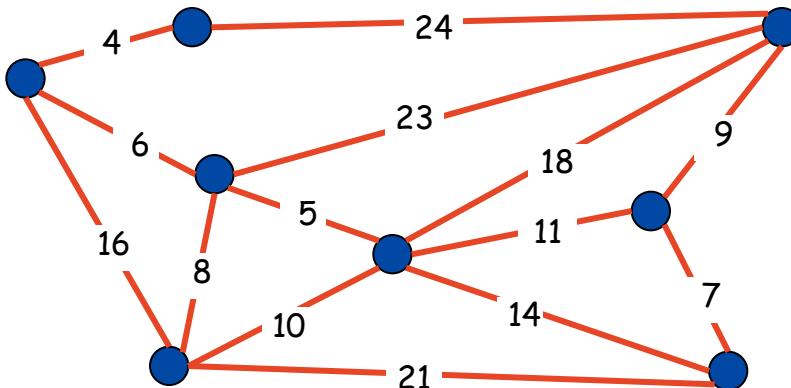
Warning: Dijkstra may not work for negative-weight edges

In the proof of correctness, even if $D[v]$ is the smallest label, it may be that $\text{dist}_w(s, v) < D[v]$ if $w(P'') < 0$

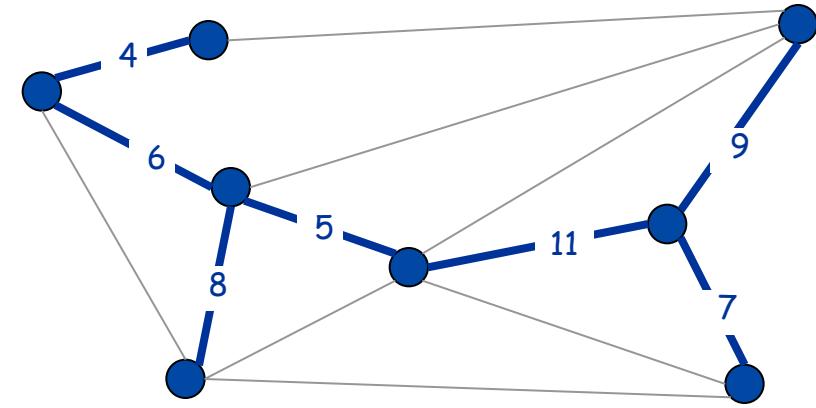


Minimum Spanning Tree

Given a connected graph $G = (V, E)$ with real-valued edge weights c_e , an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized.



$G = (V, E)$



T with $\sum_{e \in T} c_e = 50$

Applications

MST is fundamental problem with diverse applications.

Network design: Telephone, electrical, hydraulic, TV cable, computer, road

Approximation algorithms for NP-hard problems: traveling salesperson problem, Steiner tree

Indirect applications.

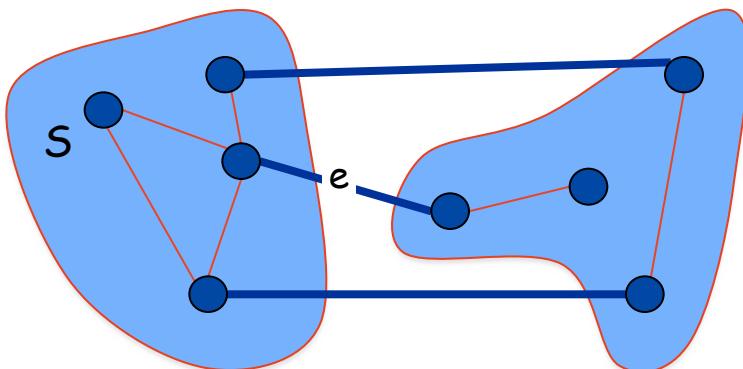
- max bottleneck paths
- LDPC codes for error correction
- image registration with Renyi entropy
- learning salient features for real-time face verification
- reducing data storage in sequencing amino acids in a protein
- ...

MST properties

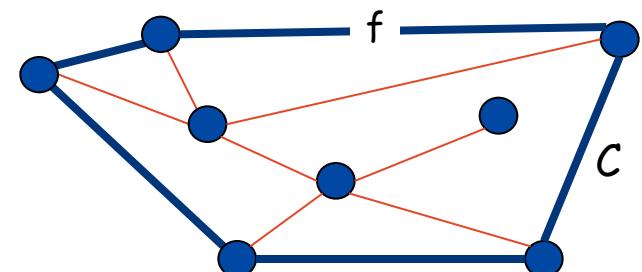
Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S . Then the MST contains e .

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C . Then the MST does not contain f .



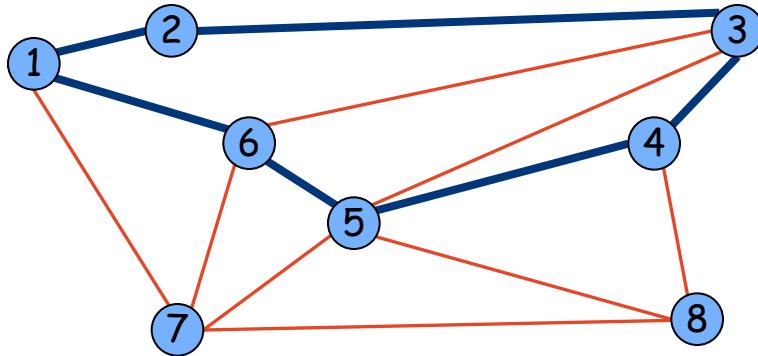
e is in the MST



f is not in the MST

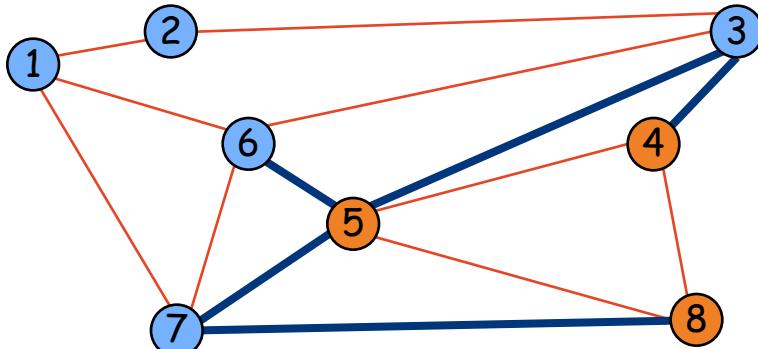
Cycles and Cuts

Cycle. Set of edges of the form $a-b, b-c, c-d, \dots, y-z, z-a$.



$$\text{Cycle } C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$$

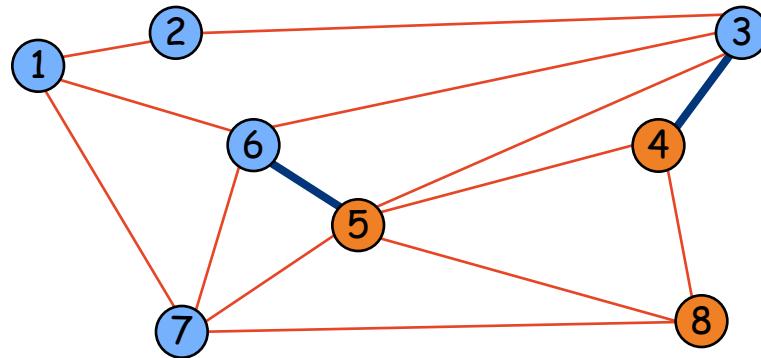
Cutset. A cut is a subset of nodes S . The corresponding cutset D is the subset of edges with exactly one endpoint in S .



$$\begin{aligned} \text{Cut } S &= \{4, 5, 8\} \\ \text{Cutset } D &= 5-6, 5-7, 3-4, 3-5, 7-8 \end{aligned}$$

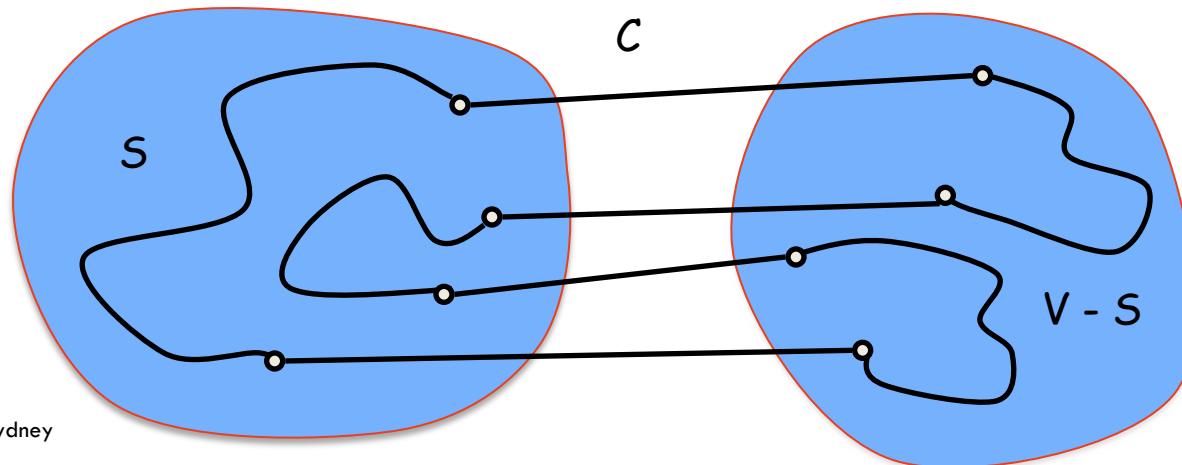
Cycle-Cut Intersection

Claim. A cycle and a cutset intersect in an even number of edges.



Cycle $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$
Cutset $D = 3-4, 3-5, 5-6, 5-7, 7-8$
Intersection = 3-4, 5-6

Proof:



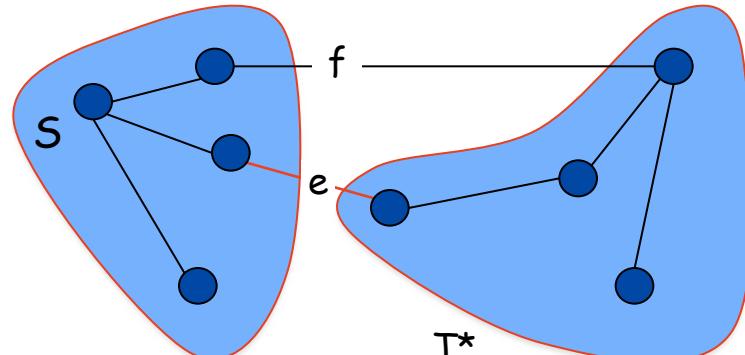
Proving the Cut Property

Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S . Then the MST contains e .

Proof: (exchange argument)

- Let T^* be MST and suppose e does not belong to T^*
- Adding e to T^* creates a cycle C in T^*
- Edge e is both in the cycle C and in the cutset D corresponding to $S \Rightarrow$ there exists another edge, say f , that is in both C and D .
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- A contradiction, so e must belong in T^*



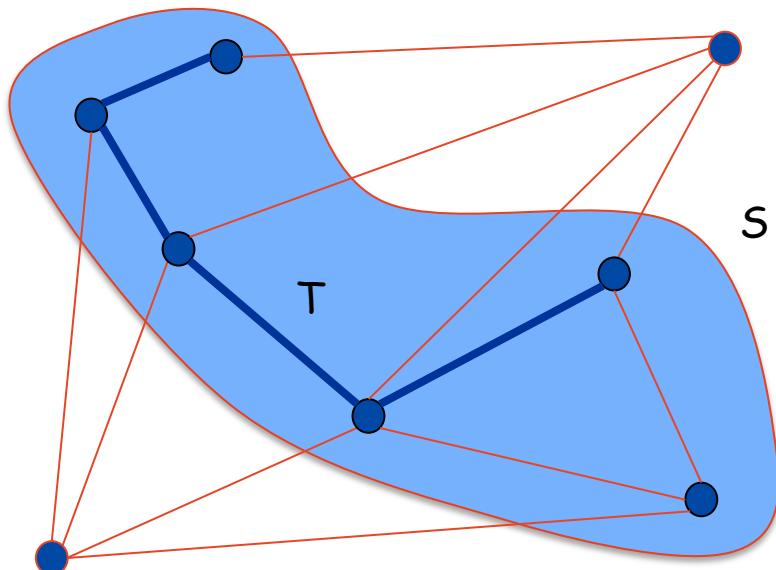
Prim's Algorithm

```
def prim(G, c):
    u ← arbitrary vertex in V
    S ← { u }
    T ← ∅
    while |S| < |V| do
        (u, v) ← min cost edge s.t. u in S and v not in S
        add (u, v) to T
        add v to S
    return T
```

Prim's Algorithm: Correctness

```
def prim(G, c):
    u ← arbitrary vertex in V
    S ← { u }
    T ← ∅
    while |S| < |V| do
        (u, v) ← min cost edge s.t. u in S and v not in S
        add (u, v) to T
        add v to S
    return T
```

Every time we add
an edge we follow
cut property!



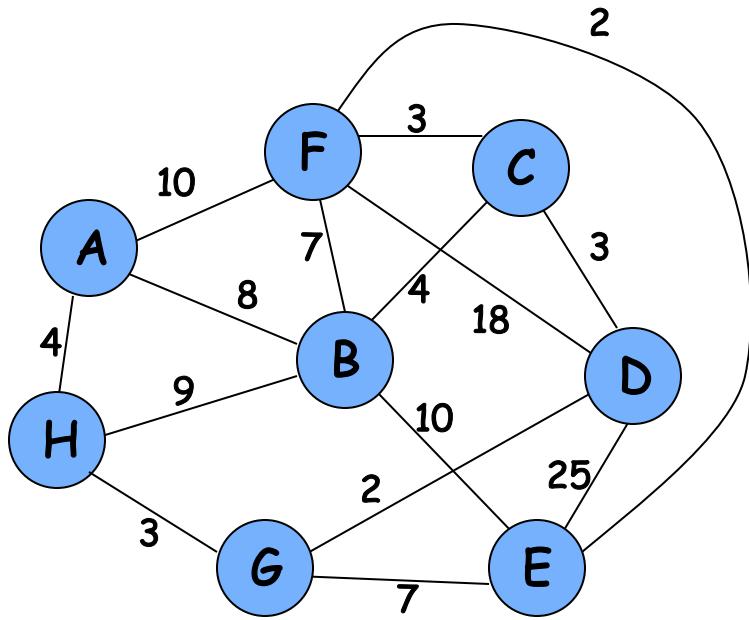
Implementation: Prim's Algorithm

```
def prim(G, c) {  
    for v in V do  
        d[v] ← ∞  
        parent[v] ← ∅  
    u ← arbitrary vertex in V  
    d[u] ← 0  
    Q ← new PQ with items { (v, d[v]) for v in V }  
    S ← ∅  
  
    while Q is not empty do  
        u ← delete min element from Q  
        add u to S  
        for (u, v) incident to u do  
            if v ∈ S and  $c_e < d[v]$  then  
                parent[v] ← u  
                decrease priority  $d[v]$  to  $c_e$   
return parent
```

Main idea: for every $v \in V \setminus S$ we keep

- $d[v] = \text{distance to closest neighbor in } S$
- $\text{parent}[v] = \text{closest neighbor in } S$

Walk-Through



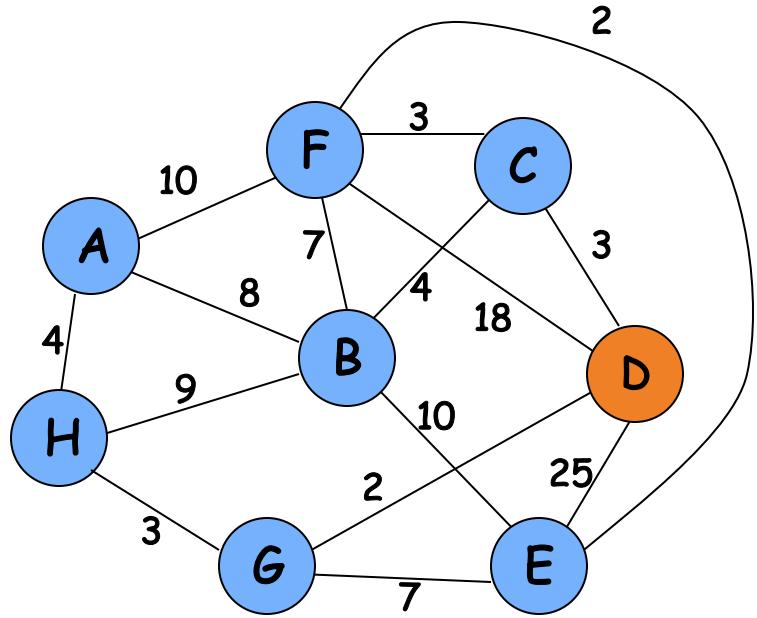
Initialize array

	S	d_v	p_v
A	F	∞	-
B	F	∞	-
C	F	∞	-
D	F	∞	-
E	F	∞	-
F	F	∞	-
G	F	∞	-
H	F	∞	-

Set
 S

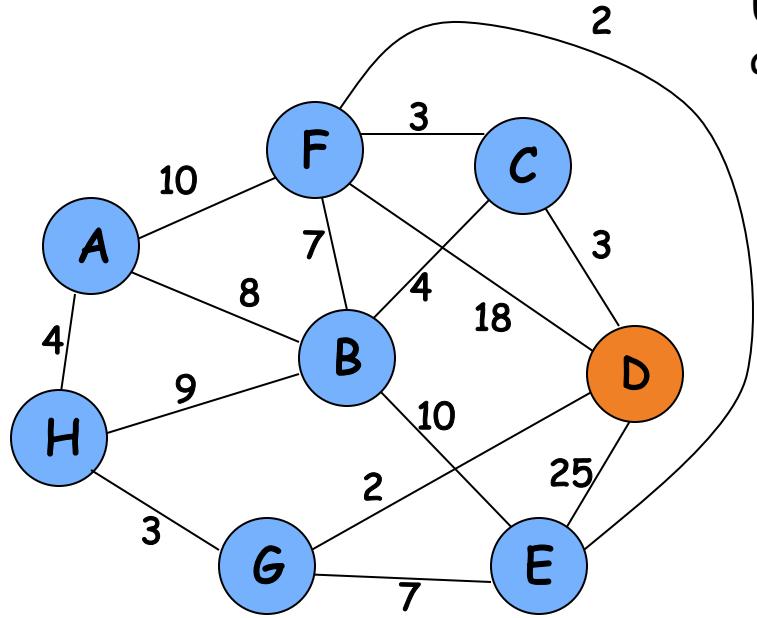
Min distance
to S

Closest
vertex in S



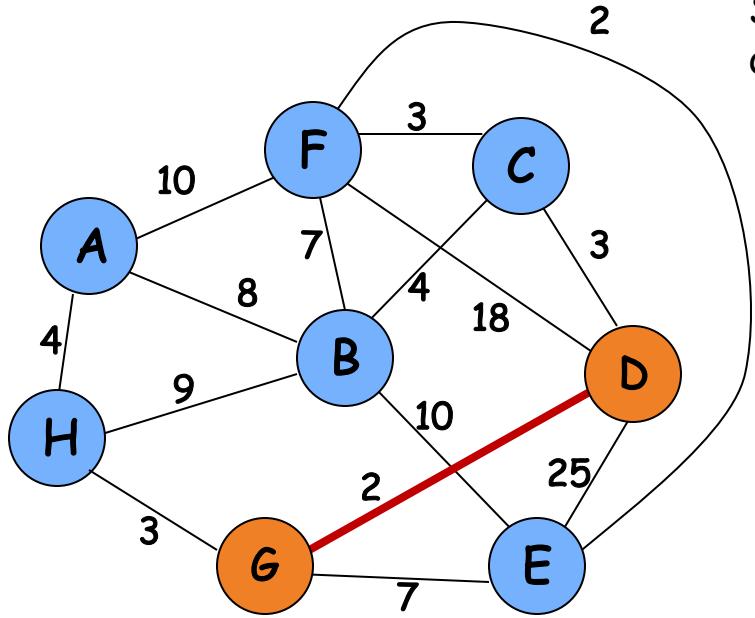
Start with any node, say D

	S	d_v	p_v
A			
B			
C			
D	T	0	-
E			
F			
G			
H			



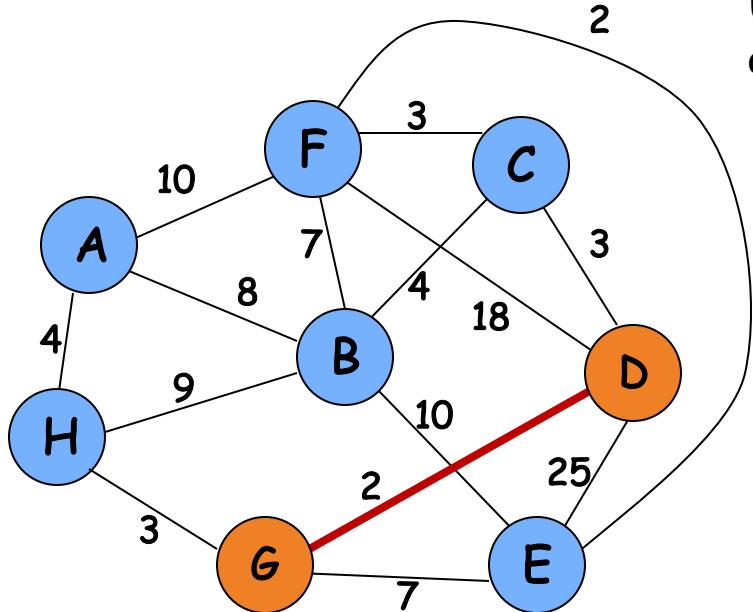
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	-
E		25	D
F		18	D
G		2	D
H			



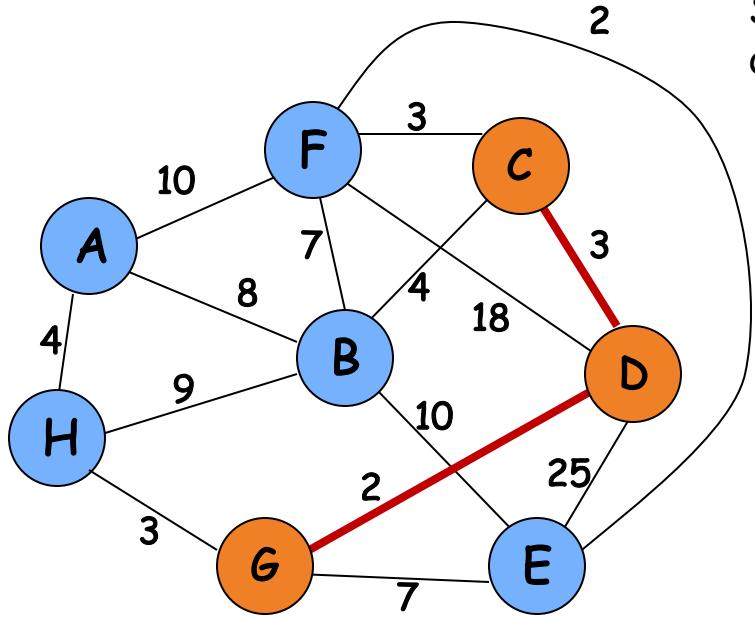
Select node with minimum distance

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	-
E		25	D
F		18	D
G	T	2	D
H			



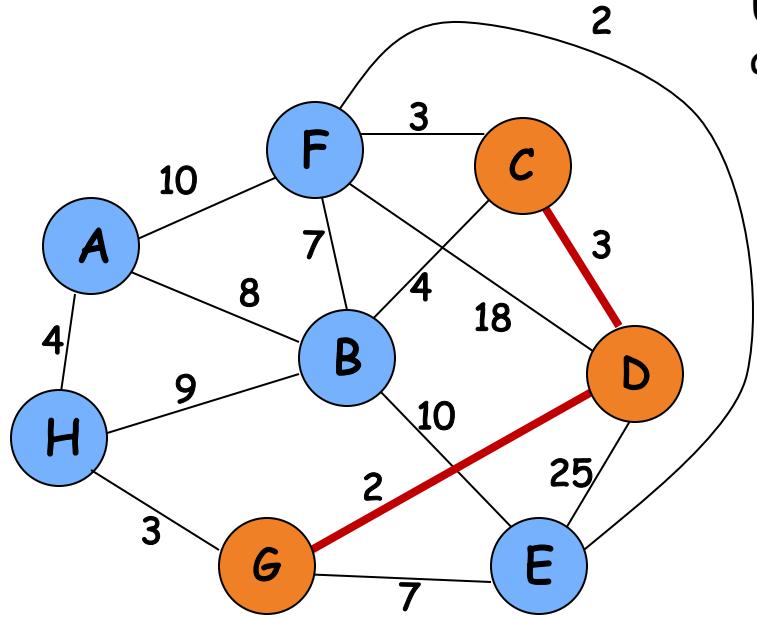
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	-
E		7	G
F		18	D
G	T	2	D
H		3	G



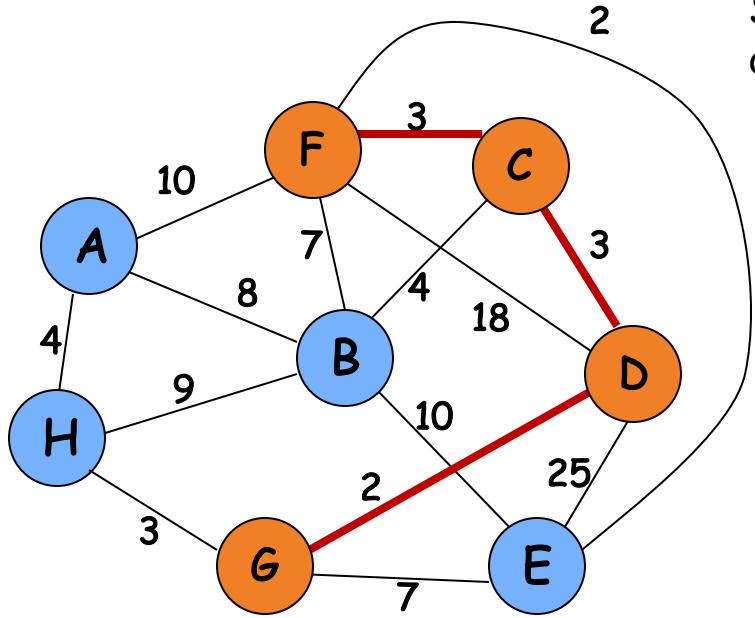
Select node with minimum distance

	S	d_v	p_v
A			
B			
C	T	3	D
D	T	0	-
E		7	G
F		18	D
G	T	2	D
H		3	G



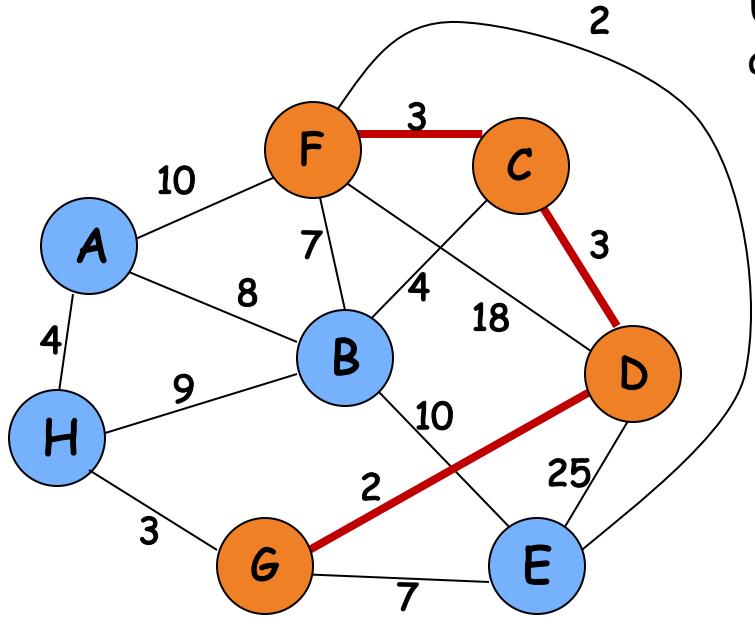
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	-
E		7	G
F		3	C
G	T	2	D
H		3	G



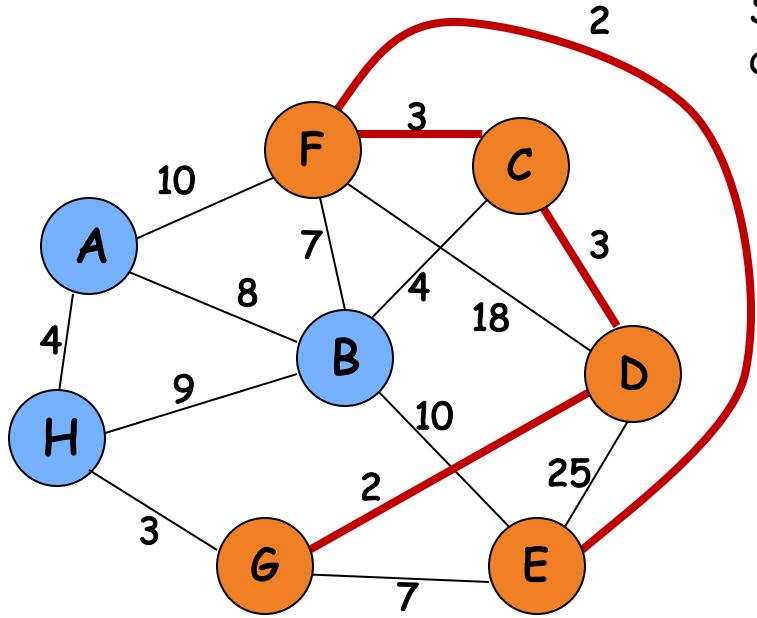
Select node with minimum distance

	S	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	-
E		7	G
F	T	3	C
G	T	2	D
H		3	G



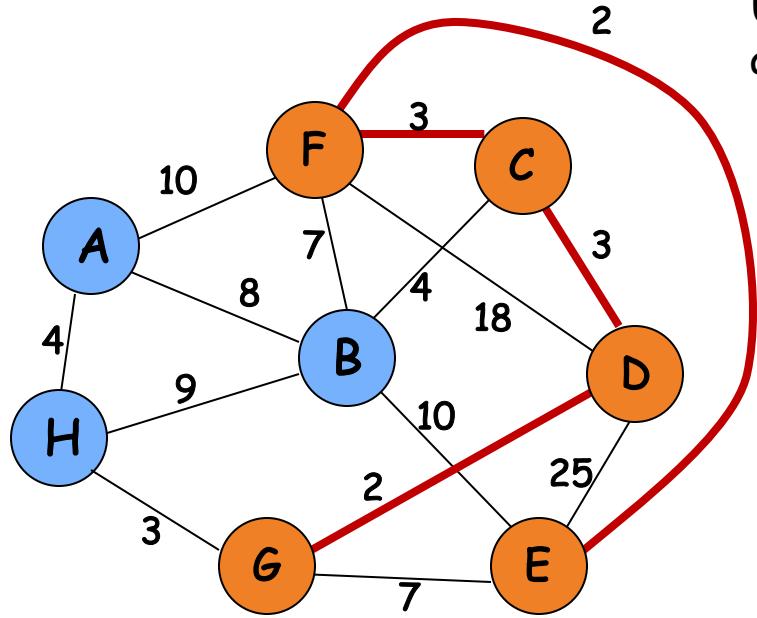
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E		2	F
F	T	3	C
G	T	2	D
H		3	G



Select node with minimum distance

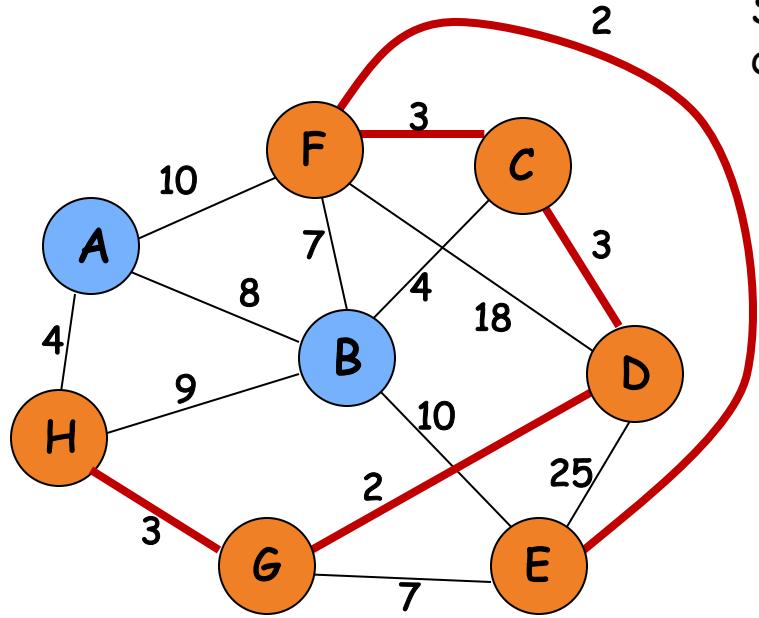
	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G



Update distances of adjacent, unselected nodes

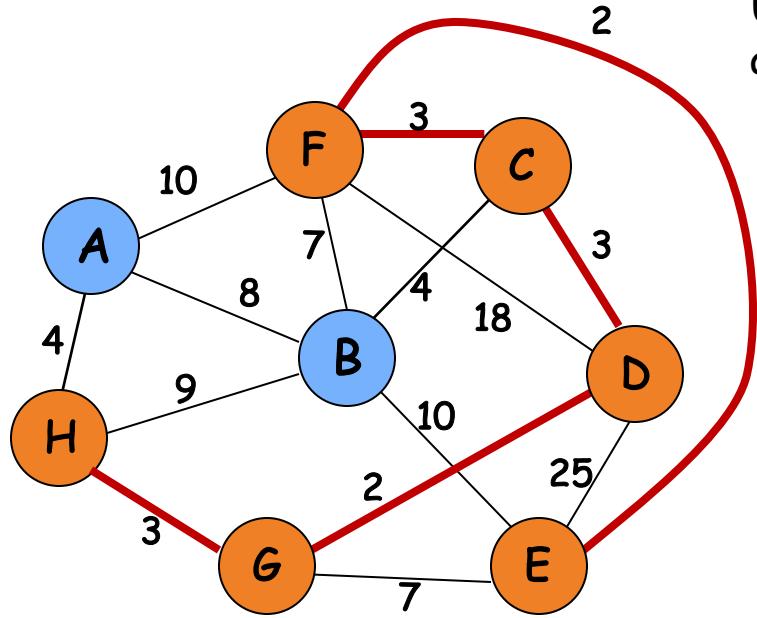
	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

Table entries unchanged



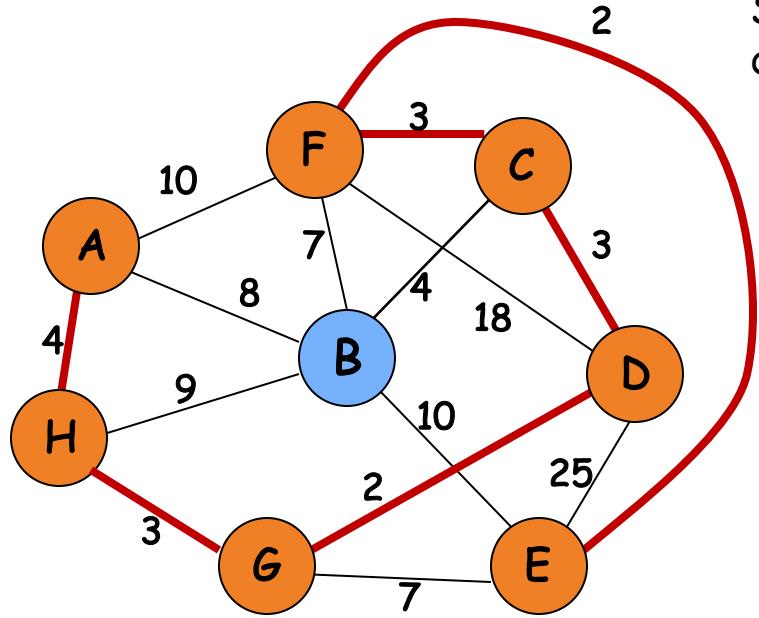
Select node with minimum distance

	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



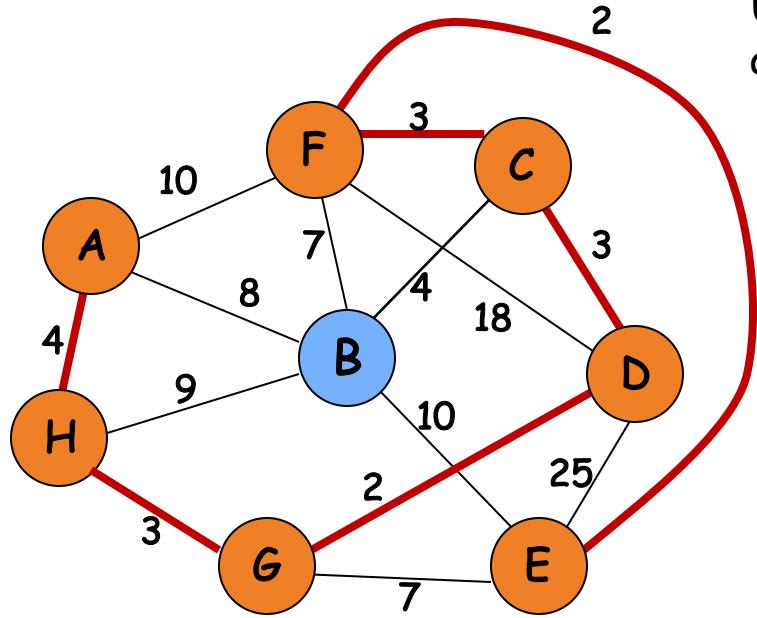
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A		4	H
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Select node with minimum distance

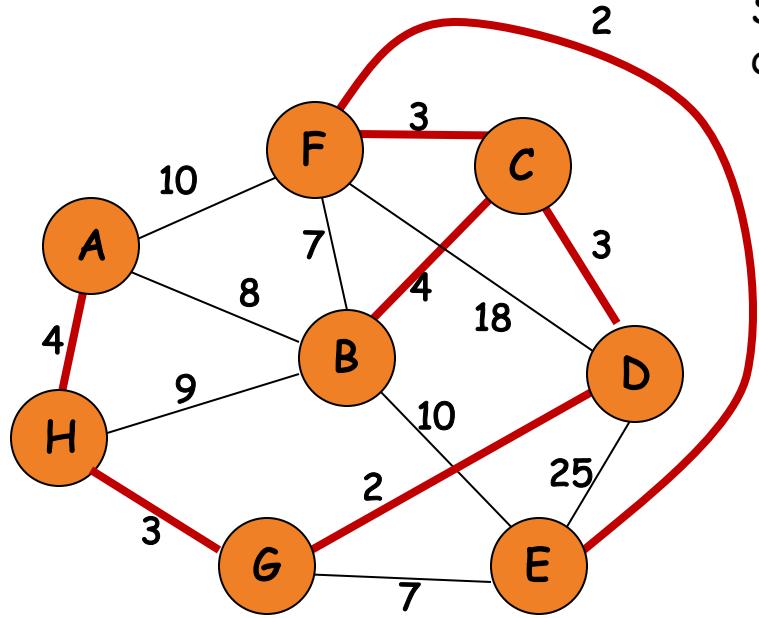
	S	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Update distances of adjacent, unselected nodes

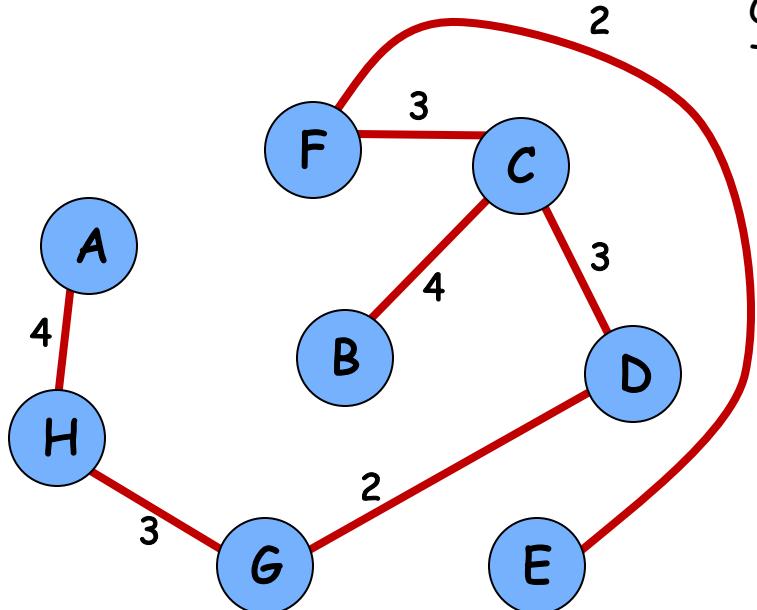
	S	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Table entries unchanged



Select node with minimum distance

	S	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Cost of Minimum Spanning Tree = $\sum d_v = 21$

	S	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Done!

Prim's Algorithm complexity

```
def prim(G, c) {  
    for v in V do  
        d[v] ← ∞  
        parent[v] ← ∅  
    u ← arbitrary vertex in V  
    d[u] ← 0  
    Q ← new PQ with items { (v, d[v]) for v in V }  
    S ← ∅  
  
    while Q is not empty do  
        u ← delete min element from Q  
        S ← S ∪ { u }  
        for (u, v) incident to u do  
            if v ∈ S and c_e < d[v] then  
                parent[v] ← u  
                decrease priority d[v] to c_e  
return parent
```

Similar analysis to Dijkstra's algorithm:

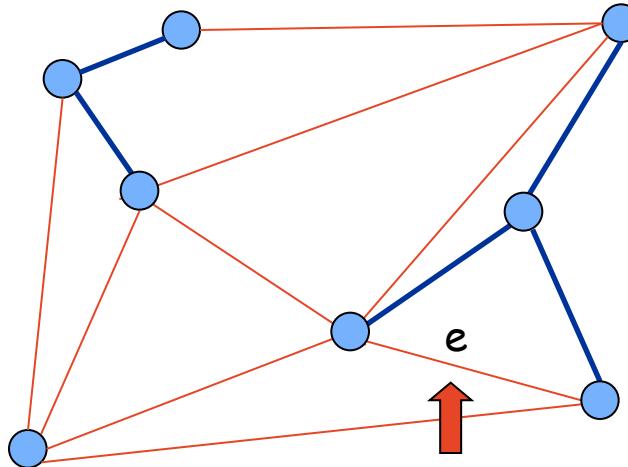
- $O(m \log n)$ using a heap
- $O(m + n \log n)$ using Fibonacci heap

Kruskal's Algorithm

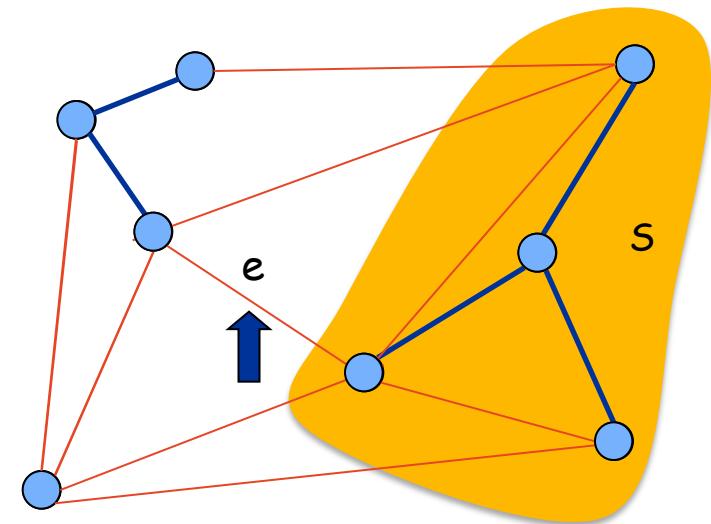
Consider edges in ascending order of weight.

Case 1: If adding e to T creates a cycle, discard e according to cycle property.

Case 2: Otherwise, insert $e = (u, v)$ into T according to cut property where $S = \text{set of nodes in } u\text{'s connected component}$.

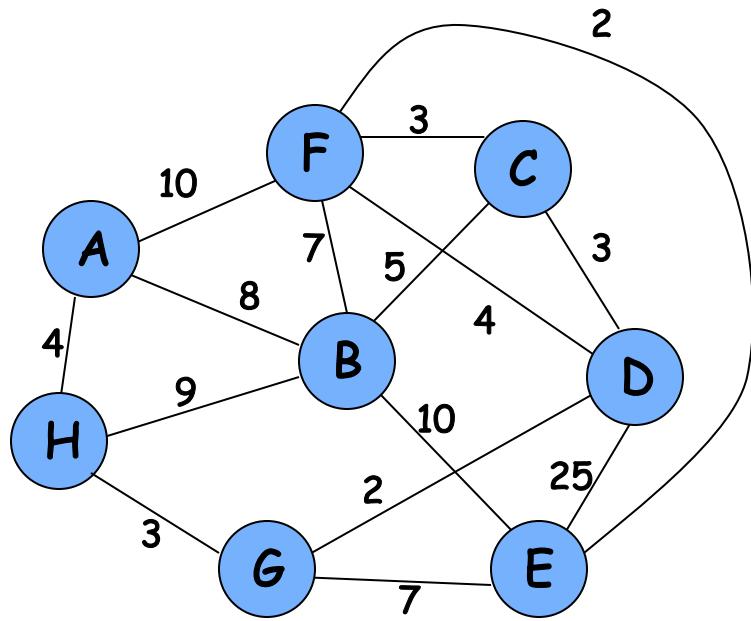


Case 1

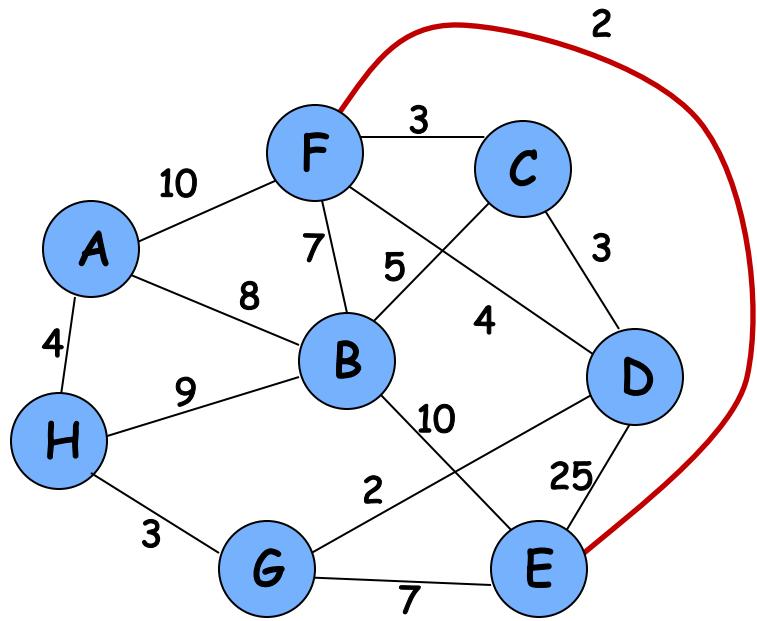


Case 2

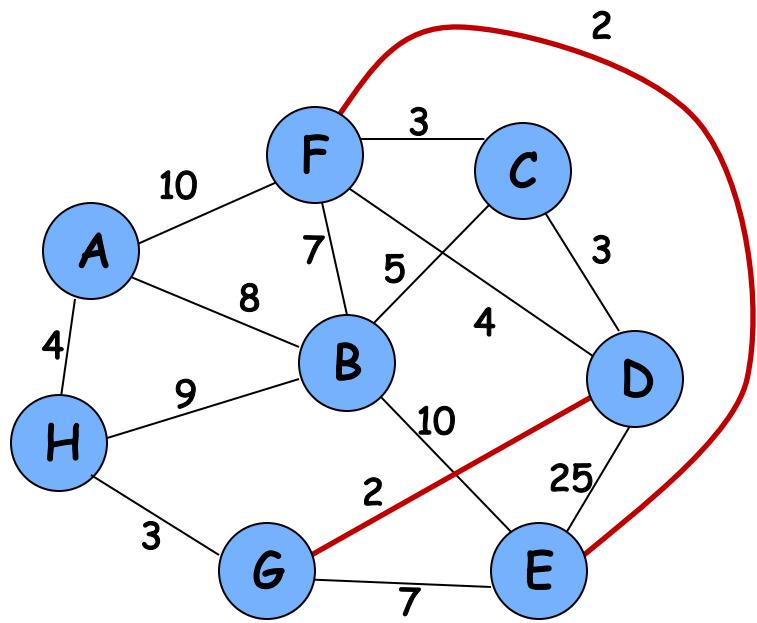
Walk-Through



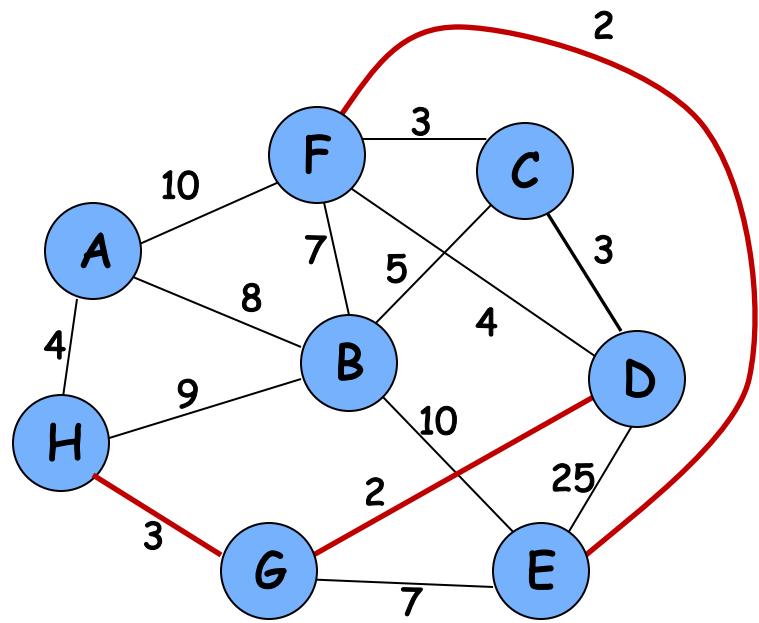
Walk-Through



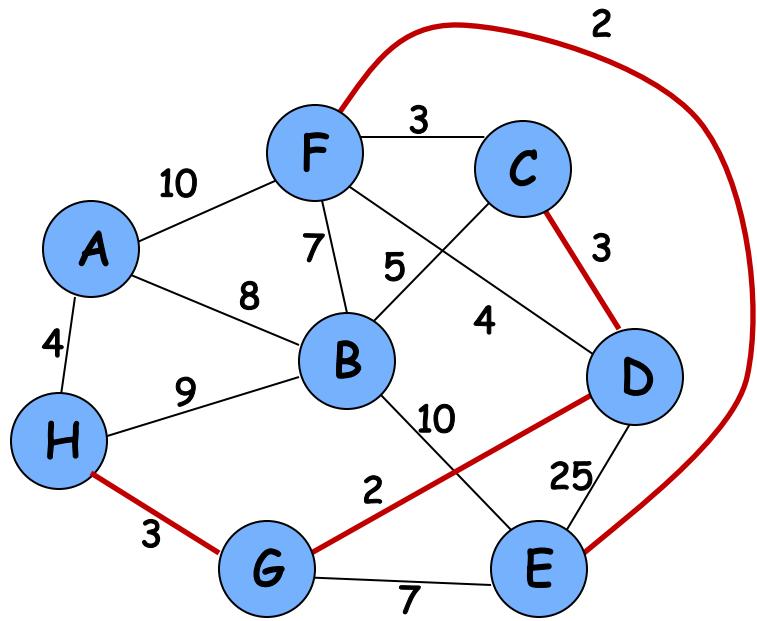
Walk-Through



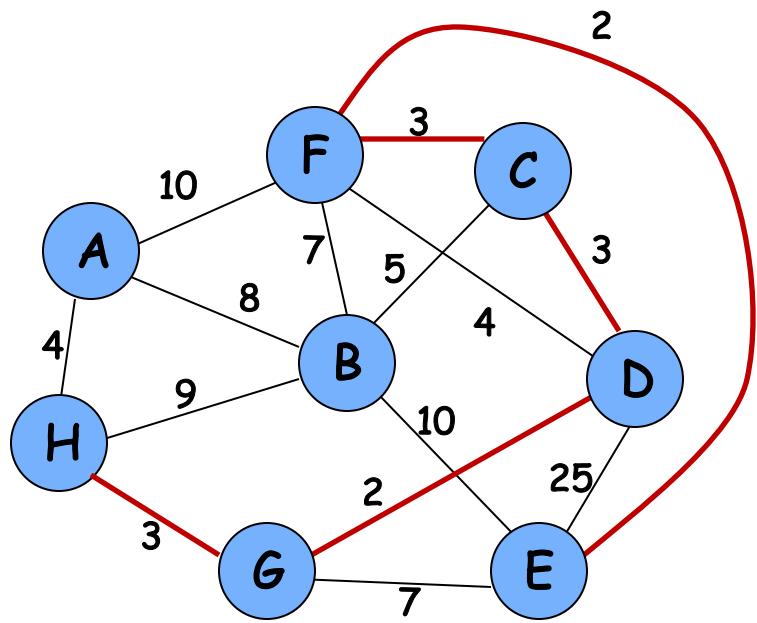
Walk-Through



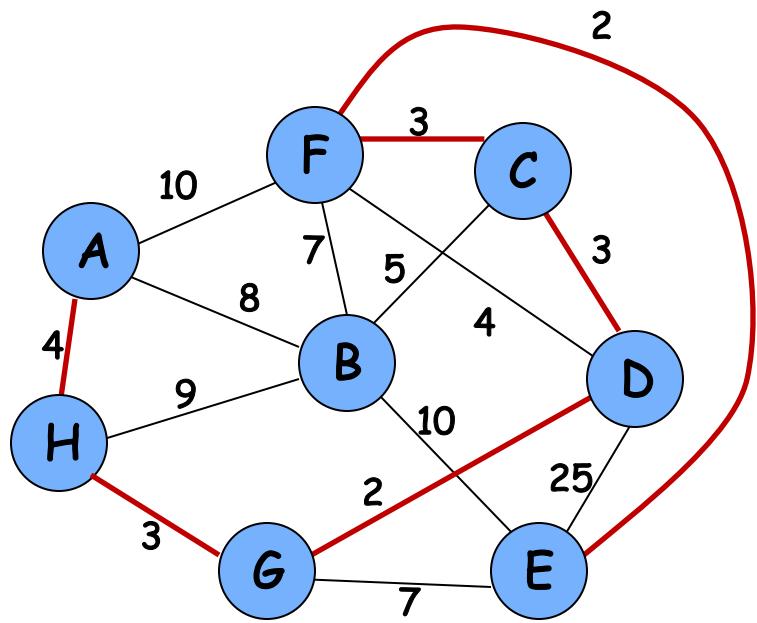
Walk-Through



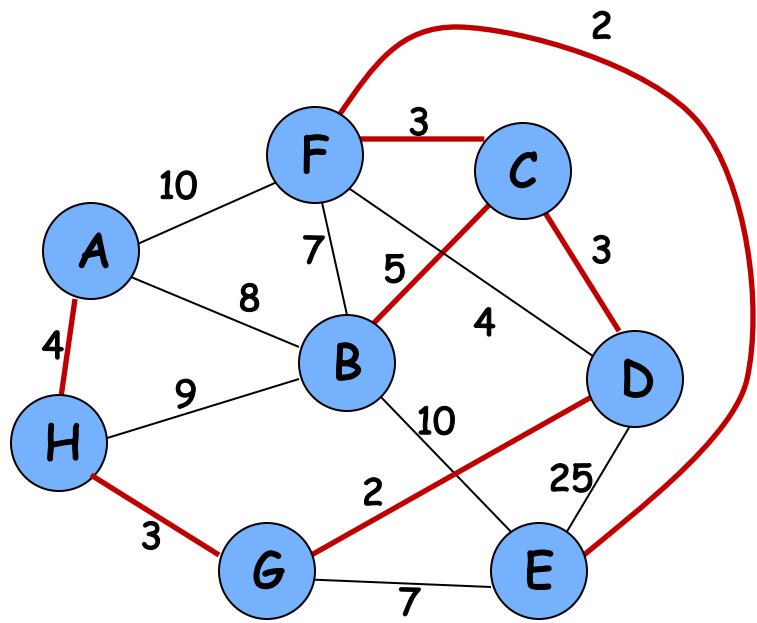
Walk-Through



Walk-Through



Walk-Through



Kruskal's Algorithm: Time complexity

Sorting edges takes $O(m \log m)$ time

We need to be able to test if adding a new edge creates a cycle, in which case we skip the edge

One option is to run DFS in each iteration to see if the number of connected components stays the same. This leads to $O(m n)$ time for the main loop

Can we do better?

Yes, keep track of the connected components with a data structure

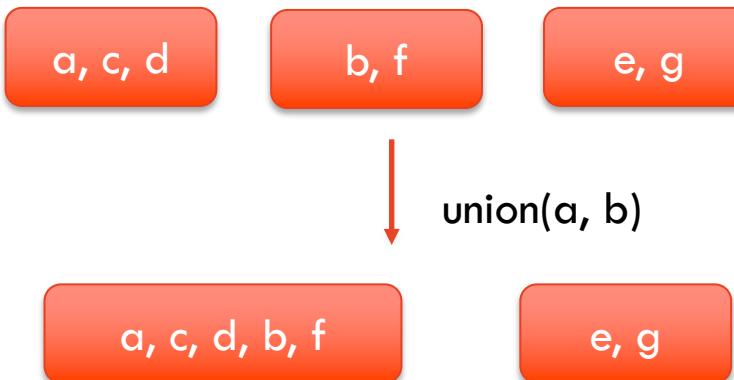
Union Find ADT

Data structure defined on a ground set of elements A

Used to keep track of an evolving partition of A

Supported operations:

- `make_sets(A)` : makes $|A|$ singleton sets with elements in A
- `find(a)` : returns an id for the set element a belongs to
- `union(a,b)` : union the sets elements a and b belong to



Kruskal's algorithm implementation

```
def Kruskal(G,c):  
  
    sort E in increasing c-value  
    answer ← [ ]  
    comp ← make_sets(V)  
    for (u,v) in E do  
        if comp.find(u) ≠ comp.find(v) then  
            answer.append( (u,v) )  
            comp.union(u, v)  
    return answer
```

Union find operations:

- `make_sets(A)` : one call with $|A| = |V|$
- `find(a)` : $2m$ calls
- `union(a,b)` : $n-1$ calls

Simple union-find implementation

Sets are represented with lists. An array points to the set each element belongs to

- `make_sets(A)` creates and initialized the array
- `find(u)` is a simple lookup in the array
- `union(u,v)` adds elements in u's set to v's set

Time complexity:

- `make_sets(A)` takes $O(n)$ time, where $n = |A|$
- `find(u)` takes $O(1)$ time
- `union(u,v)` take $O(n)$ time

Kruskal's algorithm would run in $O(n^2)$ time after the edge weights are sorted.

Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

For example, assuming all costs are integral, if we add i/n^2 to each edge e_i , then any MST under the perturbed weights is still an MST under the original weights.

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2123

Data structures and Algorithms

Lecture 9: The greedy method
[GT 10]

Dr. André van Renssen
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



Greedy algorithms

A class of algorithms where we build a solution one step at a time making locally optimal choices at each stage in the hope of finding a global optimum solution

Some of the most elegant algorithms and the simplest to implement, but often among the hardest to design and analyze

Even when they are not optimal in theory, greedy algorithms can be the basis of a very good heuristic.

Generic form

```
def generic_greedy(input):  
  
    # initialization  
    initialize result  
  
    determine order in which to consider input  
  
    # iteratively make greedy choice  
    for each element i of the input (in above order) do  
        if element i improves result then  
            update result with element i  
  
    return result
```

The Fractional Knapsack Problem



Given: A set S of n items, with each item i having

- b_i : a positive benefit
- w_i : a positive weight

Goal: Choose items with maximum total benefit of weight at most W .

Let x_i denote the amount we take of item i

Objective: maximize $\sum_{i \in S} b_i(x_i / w_i)$ [maximize benefit]

Constraint: $\sum_{i \in S} x_i \leq W$ [total weight is bounded]

$0 \leq x_i \leq w_i$ [individual weight is bounded]

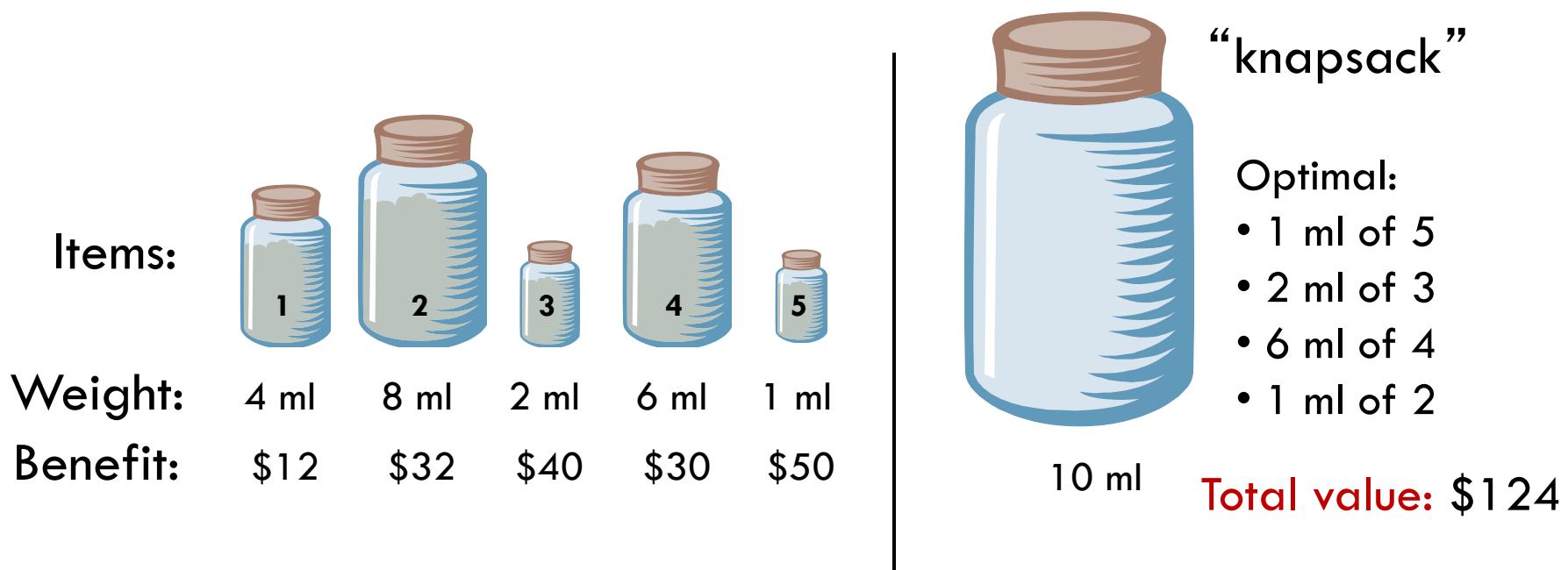
Example



Given: A set S of n items, with each item i having

- b_i - a positive benefit
- w_i - a positive weight

Goal: Choose items with maximum total benefit of weight at most W .



The Fractional Knapsack Algorithm



Initial configuration: no
items chosen

Each step: identify the
“best” item available and
add as much as possible
(all of it if you can) to the
knapsack

What defines “**best**” choice
of item to add next?

```
def fractional_knapsack(b, w, W):  
  
    # initialization  
    x ← array of size |b| of zeros  
    curr ← 0  
  
    # iteratively make greedy choice  
    while curr < W do  
        i ← “best” item not yet chosen  
        x[i] ← min(w[i], W - curr)  
        curr ← curr + x[i]  
    return x
```

Different strategies.



A greedy choice: Keep taking as much as possible of the “**best**” item, where best means:

[highest benefit]: Select items with highest benefit.

[smallest weight]: Select items with smallest weight.

[benefit/weight]: Select items with highest benefit to weight ratio.

Each of these defines a different greedy strategy for this problem.

What's “best”?



Greedy choice: Keep taking the “best” item.

[highest benefit]: Select items with highest benefit.

1 ml of 5 → \$50

2 ml of 3 → \$40

7 ml of 2 → \$28

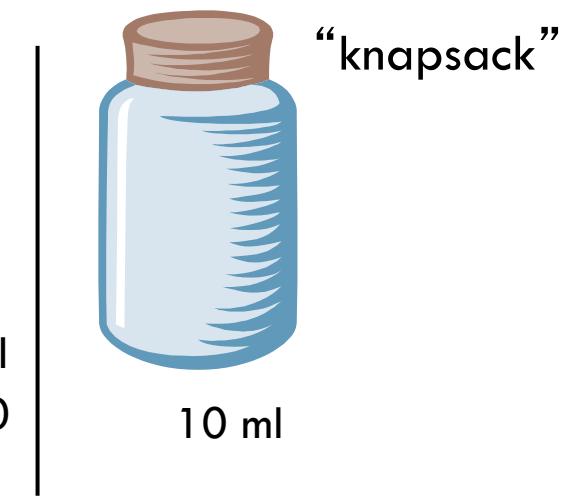
Total value: \$118

Items:



Weight: 4 ml 8 ml 2 ml 6 ml 1 ml

Benefit: \$12 \$32 \$40 \$30 \$50



What's “best”?



Greedy choice: Keep taking the “best” item.

[smallest weight]: Select items with smallest weight.

1 ml of 5 → \$50

2 ml of 3 → \$40

4 ml of 1 → \$12

3 ml of 4 → \$15

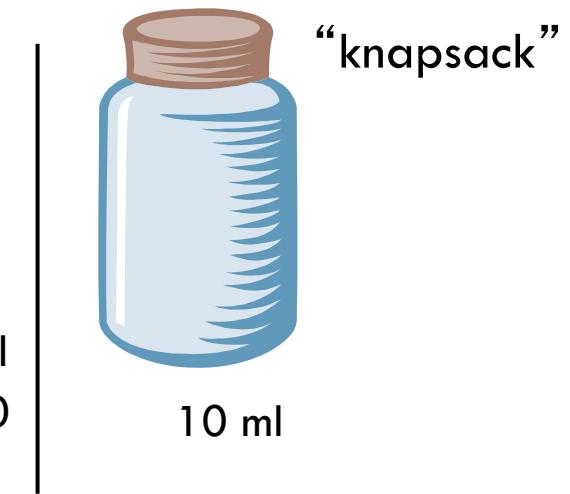
Total value: \$117

Items:



Weight: 4 ml 8 ml 2 ml 6 ml 1 ml

Benefit: \$12 \$32 \$40 \$30 \$50



What's “best”?



Greedy choice: Keep taking the “best” item.

[benefit/weight]: Select items with highest benefit to weight ratio.

1 ml of 5 → \$50

2 ml of 3 → \$40

6 ml of 4 → \$30

1 ml of 2 → \$4

Total value: \$124

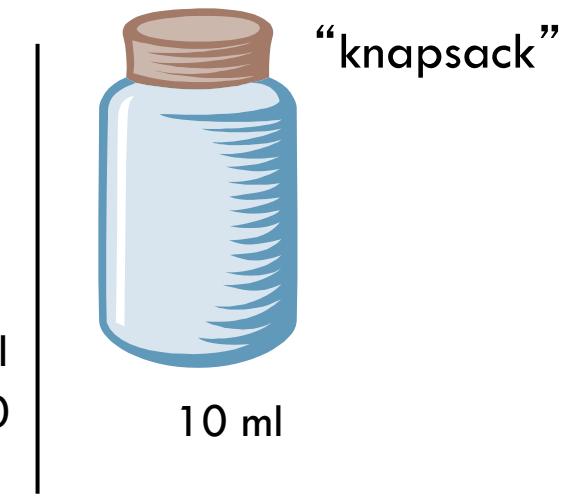
Items:



Weight: 4 ml 8 ml 2 ml 6 ml 1 ml

Benefit: \$12 \$32 \$40 \$30 \$50

Benefit/ml: 3 4 20 5 50



The Fractional Knapsack Algorithm: Correctness

Theorem: The greedy strategy of picking item with highest benefit to weight ratio computes an optimal solution.

Proof (sketch):

- Use an exchange argument
- Assume for simplicity that all ratios are different $b_i/w_i \neq b_k/w_k$
- Consider some feasible solution x different than the greedy one
- There must be items i and k s.t. $x_i < w_i$, $x_k > 0$ and $b_i/w_i > b_k/w_k$
- If we replace some k with some of i , we get a better solution
- How much? $\min\{w_i - x_i, x_k\}$
- Thus, there is no better solution than the greedy one

The Fractional Knapsack Algorithm: Complexity

Sort items by their benefit-to-weight values, and then process them in this order.

Require $O(n \log n)$ time to sort the items and then $O(n)$ time to process them in the for-loop.

```
def fractional_knapsack(b, w, W):
    # initialization
    x ← array of size |b| of zeros
    curr ← 0

    # iteratively do greedy choice
    for i in descending b[i]/w[i] order do
        x[i] ← min(w[i], W - curr)
        curr ← curr + x[i]
    return x
```

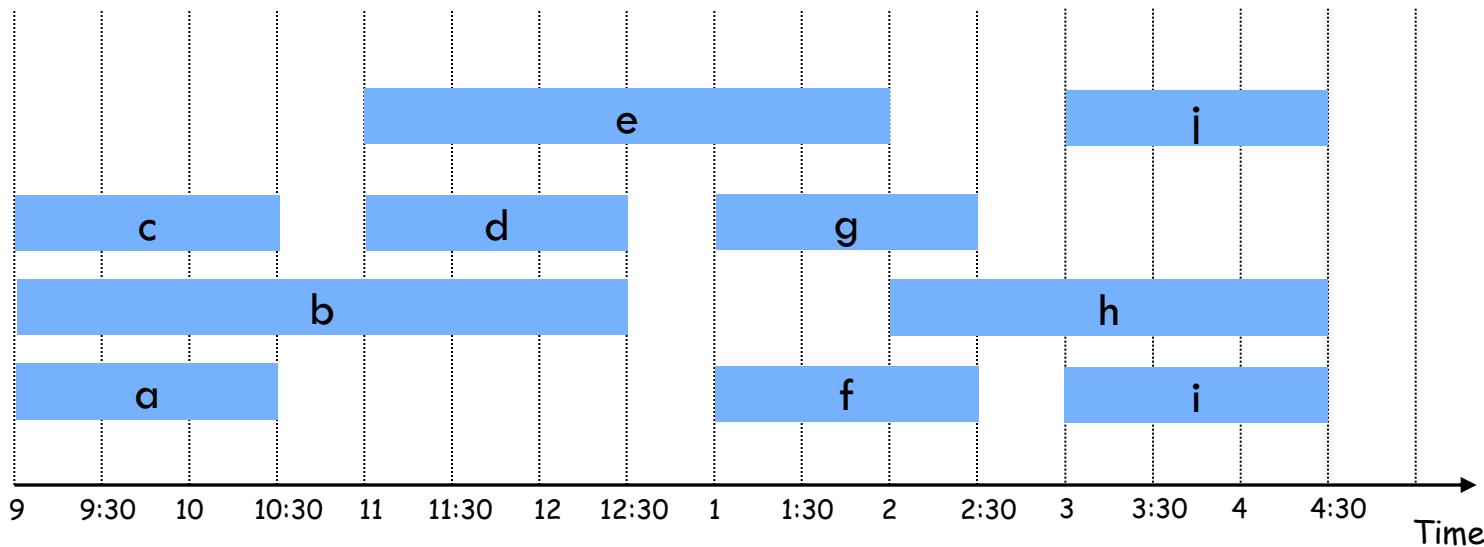
Task scheduling

Given: A set S of n lectures

Lecture i starts at s_i and finishes at f_i .

Goal: Find the minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Example: This schedule uses 4 classrooms to schedule 10 lectures.



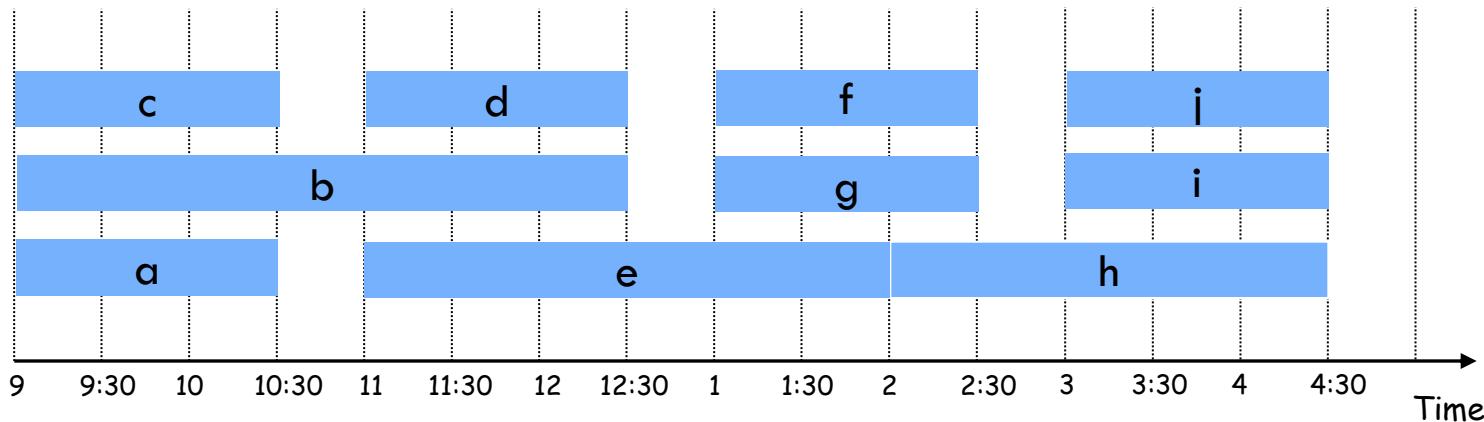
Task scheduling

Given: A set S of n lectures

Lecture i starts at s_i and finishes at f_i .

Goal: Find the minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Example: This schedule uses only 3!



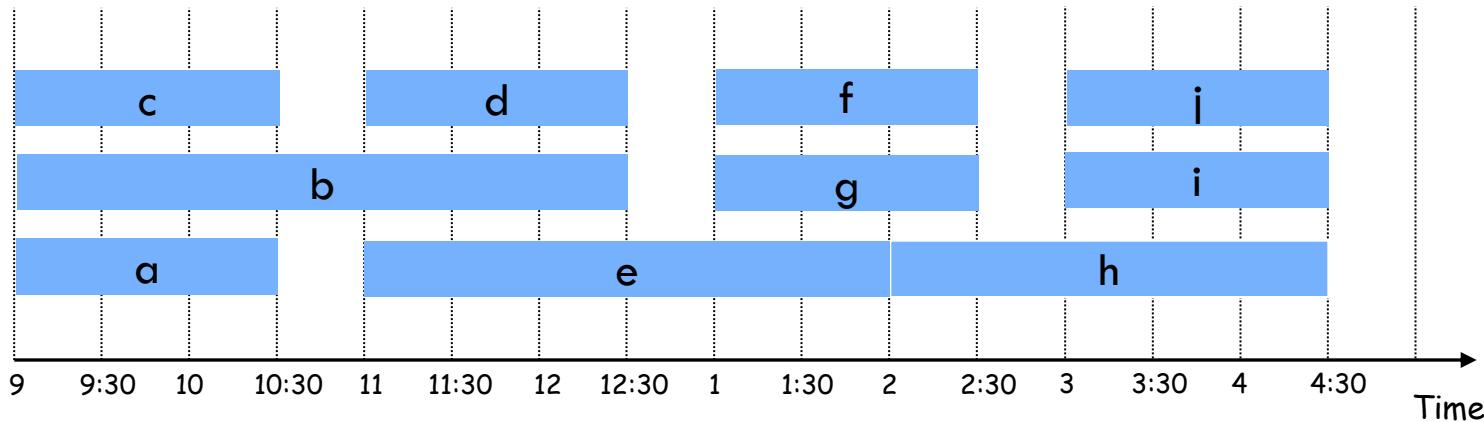
Interval Partitioning: Lower bound

Definition: The **depth** of a set of open intervals is the maximum number that contain any given time.

Observation: Number of classrooms needed \geq depth. Why?

Example: Depth of schedule below is 3 [a, b, c all contain 9:30]
 \Rightarrow schedule below is optimal.

Question: Does there always exist a schedule equal to depth of intervals?



Interval Partitioning: Greedy Algorithm

Greedy algorithm: Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
def interval_partition(S):
    # initialization
    sort intervals in increasing starting time order
    d ← 0      # number of allocated classrooms

    # iteratively do greedy choice
    for i in increasing starting time order do
        if lecture i is compatible with some classroom k then
            schedule lecture i in classroom 1 ≤ k ≤ d
        else
            allocate a new classroom d+1
            schedule lecture i in classroom d+1
            d ← d+1
    return d
```

Interval Partitioning: Greedy Algorithm

Greedy algorithm: Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
def interval_partition(S):
    # initialization
    sort intervals in increasing starting time order
    d ← 0      # number of allocated classrooms

    # iteratively do greedy choice
    for i in increasing starting time order do
        if lecture i is compatible with some classroom k then
            schedule lecture i in classroom  $1 \leq k \leq d$ 
        else
            allocate a new classroom  $d+1$ 
            schedule lecture i in classroom  $d+1$ 
            d ← d+1
    return d
```

Implementation: $O(n \log n)$.

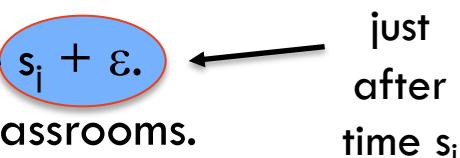
- For each classroom k , maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

Interval Partitioning: Greedy Analysis

Observation: Greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem: Greedy algorithm is optimal.

Proof:

- $d = \text{number of classrooms that the greedy algorithm allocates.}$
- Classroom d is opened because we needed to schedule a job, say i , that is incompatible with all $d-1$ other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_i .
- Thus, we have d lectures overlapping at time $s_i + \varepsilon$.  just after time s_i
- Key observation \Rightarrow all schedules use $\geq d$ classrooms.

[Greedy algorithm stays “ahead”]

Text Compression

Given: a string X

Goal: efficiently encode X into a smaller string Y
(saves memory and/or bandwidth)

Input:

WWWWWWWWWWWWWWWWBWWWWWWWWWWWWWWWWBWWWWWWWW
WW

Run length encoding (very simple approach):

12W1B12W3B24W1B14W

Text Compression

Given: a string X

Goal: efficiently encode X into a smaller string Y
(saves memory and/or bandwidth)

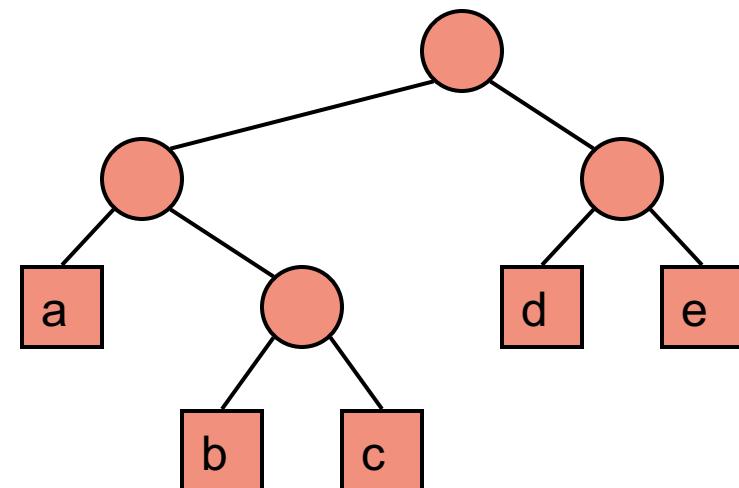
A better approach: **Huffman encoding**

- Let C be the set of characters in X
- Compute frequency $f(c)$ for each character c in C
- Encode high-frequency characters with short code words
- No code word is a prefix for another code
- Use an optimal encoding tree to determine the code words

Encoding Tree Example

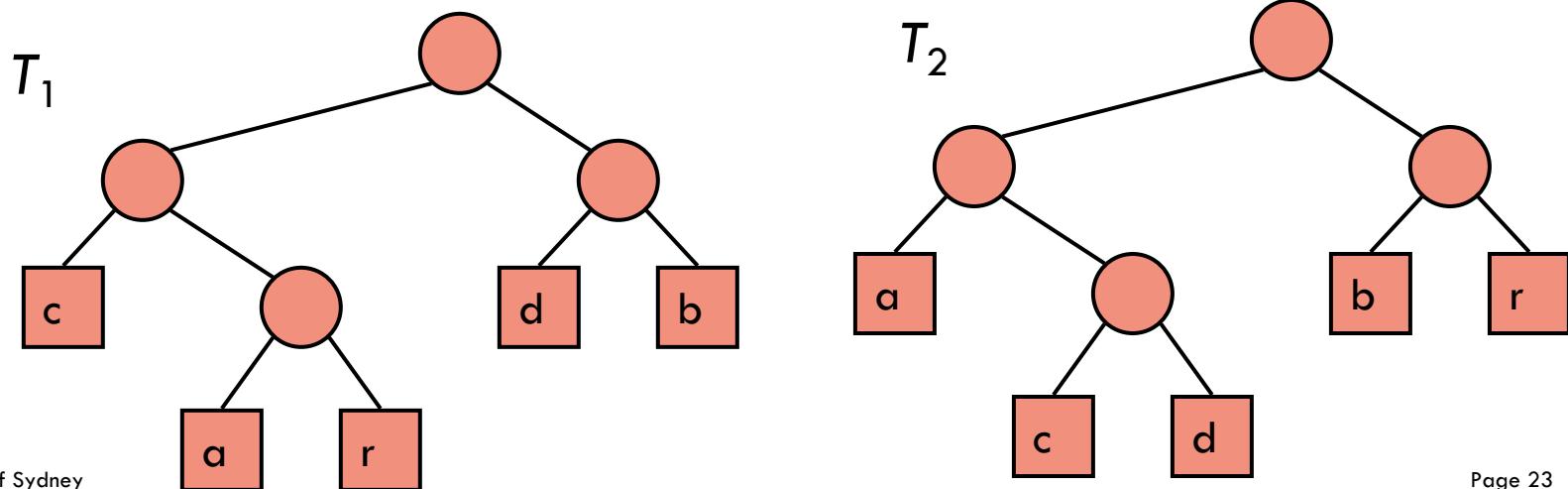
- A **code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
 - Each external node stores a character
 - The code-word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

00	010	011	10	11
a	b	c	d	e



Encoding Tree Optimization

- Given a text string X , we want to find a prefix code for the characters of X that yields a small encoding for X
 - Frequent characters should have short code-words
 - Rare characters should have long code-words
- Example
 - $X = \text{abracadabra}$
 - T_1 encodes X into 29 bits
 - T_2 encodes X into 24 bits



Huffman's Algorithm

Given a string X , Huffman's algorithm constructs a prefix code that minimizes the size of the encoding of X

It runs in time $O(n + d \log d)$, where n is the size of X and d is the number of distinct characters of X

The algorithm builds the encoding tree from the bottom up, merging trees as it goes along, using a priority queue to guide the process

End result minimizes bits needed to encode X :

$$\sum_{c \text{ in } C} f(c) * \text{depth}_T(c)$$

Huffman's Algorithm

```
def huffman(C, f):  
  
    # initialize priority queue  
    Q ← empty priority queue  
    for c in C do  
        T ← single-node binary tree storing c  
        Q.insert(f[c], T)  
  
    # merge trees while at least two trees  
    while Q.size() > 1 do  
        f1, T1 ← Q.remove_min()  
        f2, T2 ← Q.remove_min()  
        T ← new binary tree with T1/T2 as left/right subtrees  
        f ← f1 + f2  
        Q.insert(f, T)  
  
    # return last tree  
    f, T ← Q.remove_min()  
    return T
```

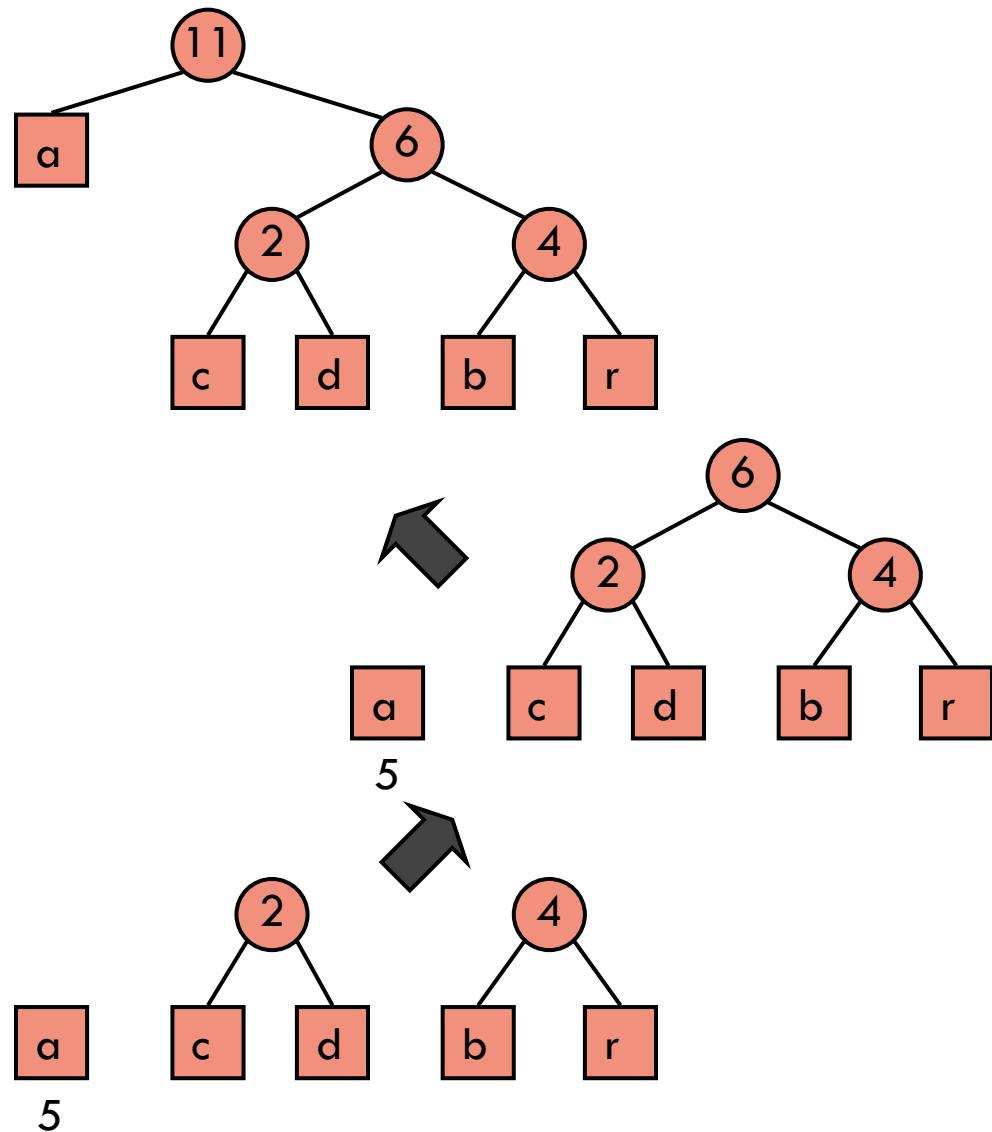
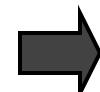
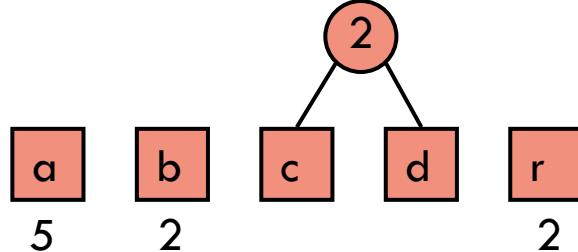
Example

$X = \text{abracadabra}$

Frequencies

a	b	c	d	r
5	2	1	1	2

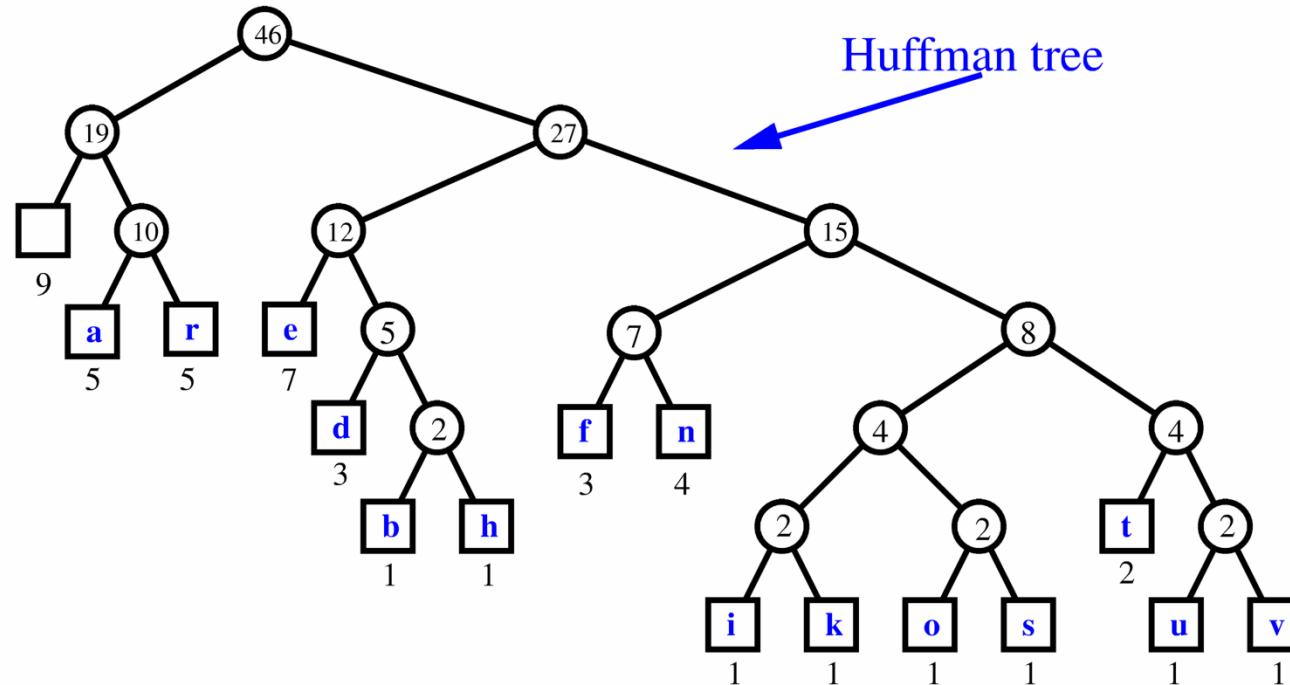
a	b	c	d	r
5	2	1	1	2



Extended Huffman Tree Example

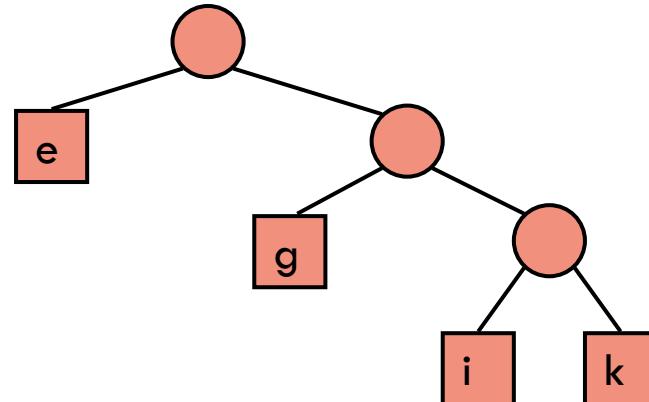
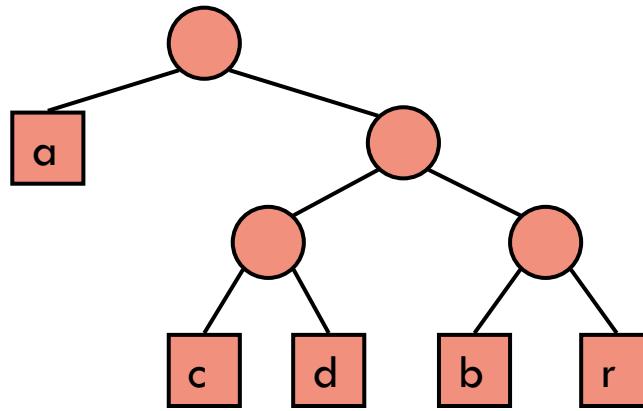
String: **a fast runner need never be afraid of the dark**

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v	
Frequency		9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



Huffman's Algorithm Correctness

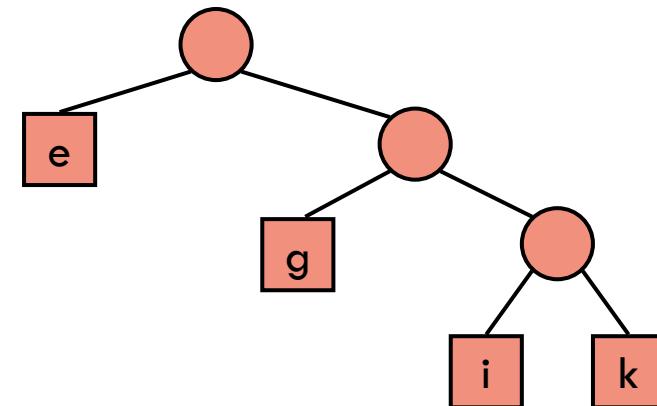
Obs: Every encoding tree has a pair of leaves that are *siblings*.



Huffman's Algorithm Correctness

Obs: In an optimal encoding tree T for any a and b in C , if $\text{depth}_T(a) < \text{depth}_T(b)$ then $f(a) \geq f(b)$.

$$\sum_{c \text{ in } C} f(c) * \text{depth}_T(c)$$

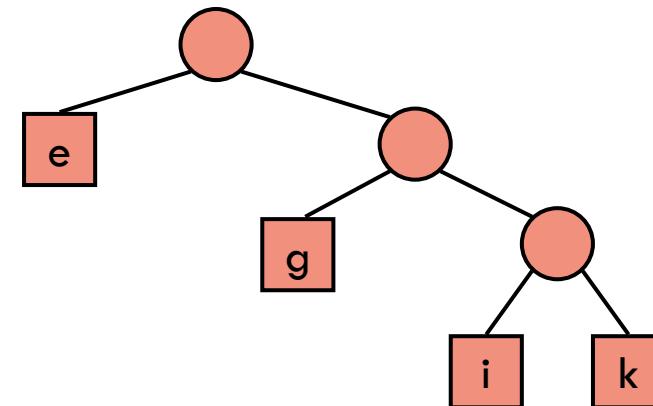


For example, if $f(e) < f(g)$
then swapping them leads
to shorter encoding

Huffman's Algorithm Correctness

Obs: There is an optimal encoding tree T where the two sibling leaves furthest from the root have lowest frequency.

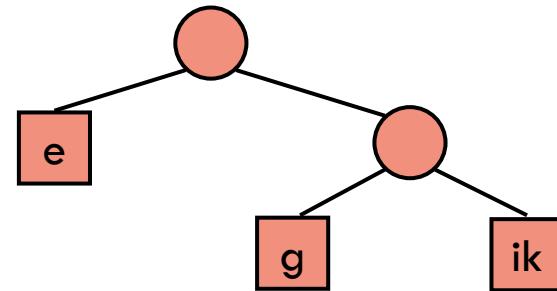
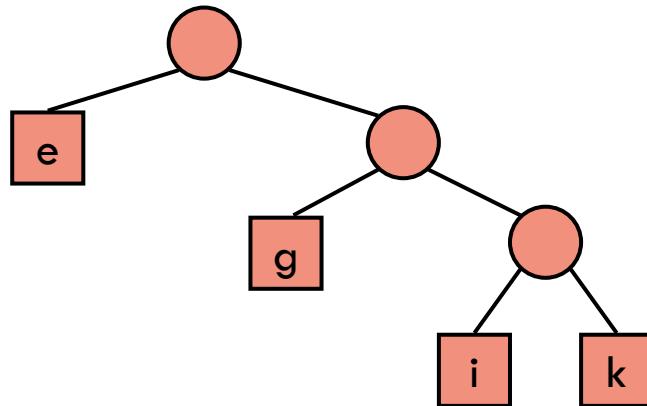
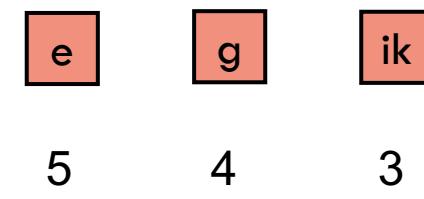
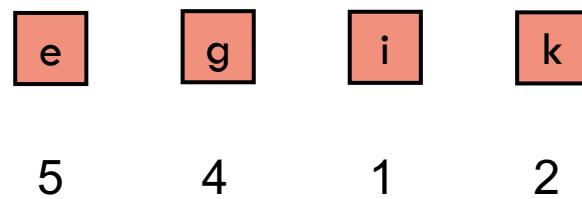
$$\sum_{c \text{ in } C} f(c) * \text{depth}_T(c)$$



For example, characters **i** and **k**
have lowest frequency

Huffman's Algorithm Correctness

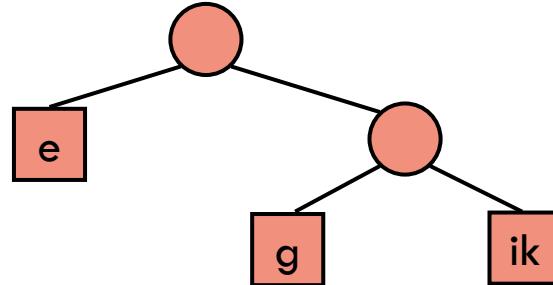
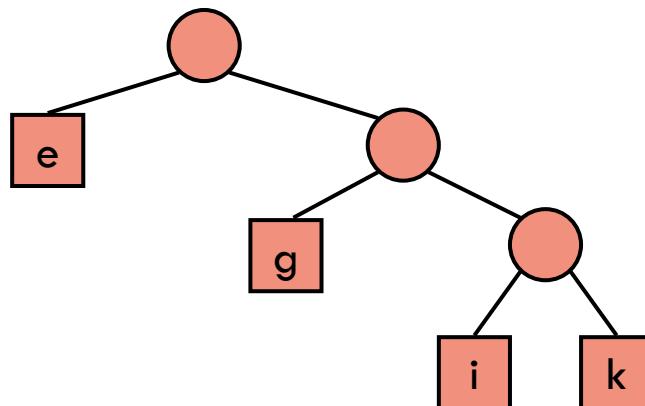
Obs: If we combine the two lowest frequency characters to get a new instance (C', f') , an optimal encoding tree T' for (C', f') can be expanded to get an optimal encoding tree T for (C, f)



Huffman's Algorithm Correctness

Obs: If we combine the two lowest frequency characters to get a new instance (C', f') , an optimal encoding tree T' for (C', f') can be expanded to get an optimal encoding tree T for (C, f)

$$\begin{aligned} \sum_{c \text{ in } C} f(c) * \text{depth}_T(c) - \sum_{c \text{ in } C'} f'(c) * \text{depth}_{T'}(c) \\ = f(i) * \text{depth}_T(i) + f(k) * \text{depth}_T(k) - f'(ik) * \text{depth}_{T'}(ik) \\ = f(i) + f(k) \end{aligned}$$



Huffman's Algorithm Correctness

Thm: Huffman's algorithm computes a minimum length encoding tree of (C, f)

Proof (by induction):

- If $|C| = 1$ then the encoding is trivially optimal
- If $|C| > 1$ then let (C', f') be the contracted instance
- By inductive hypothesis, the encoding tree T' constructed for (C', f') is optimal
- Recall that

$$\sum_{c \text{ in } C} f(c) * \text{depth}_T(c) = \sum_{c \text{ in } C'} f'(c) * \text{depth}_{T'}(c) + f(i) + f(k)$$

thus, the tree T is optimal for (C, f)

Huffman's Algorithm

```
def huffman(C, f):  
  
    # initialize priority queue  
    Q ← empty priority queue  
    for c in C do  
        T ← single-node binary tree storing c  
        Q.insert(f[c], T)  
  
    # merge trees while at least two trees  
    while Q.size() > 1 do  
        f1, T1 ← Q.remove_min()  
        f2, T2 ← Q.remove_min()  
        T ← new binary tree with T1/T2 as left/right subtrees  
        f ← f1 + f2  
        Q.insert(f, T)  
  
    # return last tree  
    f, T ← Q.remove_min()  
    return T
```

Time complexity is dominated by PQ ops, which using heap take $O(|C| \log |C|)$ time

Greedy algorithms recap

Greedy heuristics are easy to design but they are not always optimal. And when they are, small modifications of the problem can render them suboptimal:

- 0-1 knapsack is hard
- if tasks/lectures have special needs the problem is hard
- if we use non-binary encodings, Huffman does not work

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2123

Data structures and Algorithms

Lecture 10: Divide and Conquer

[GT 3.1 and 8]

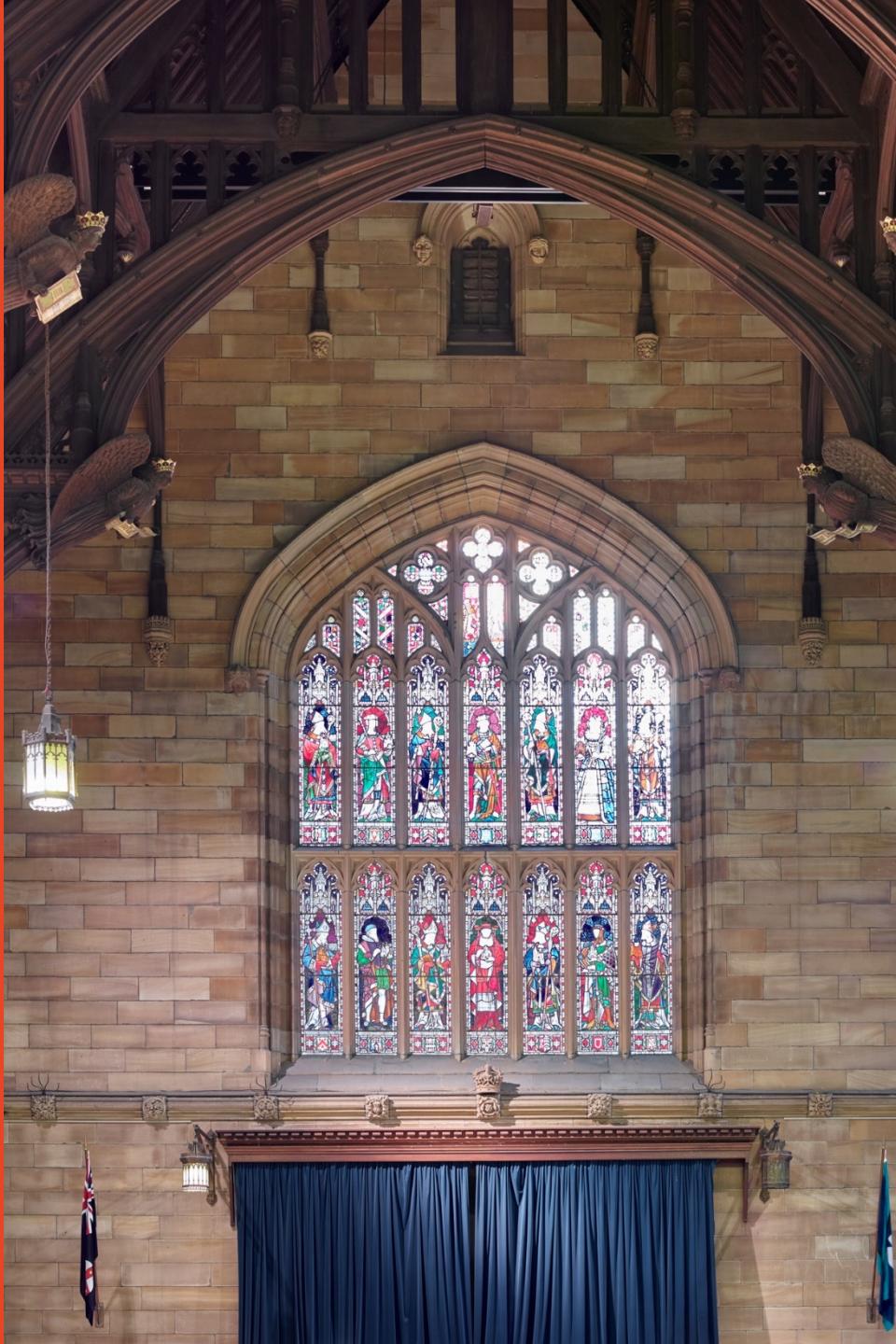
Dr. André van Renssen

School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



THE UNIVERSITY OF
SYDNEY



Divide and Conquer

Divide and Conquer algorithms can normally be broken into these three parts:

1. **Divide** If it is a base case, solve directly, otherwise break up the problem into several parts.
2. **Recur/Delegate** Recursively solve each part [each sub-problem].
3. **Conquer** Combine the solutions of each part into the overall solution.

Divide and Conquer

1. **Divide** If it is a base case, solve directly, otherwise break up the problem into several parts.

Typical base case:

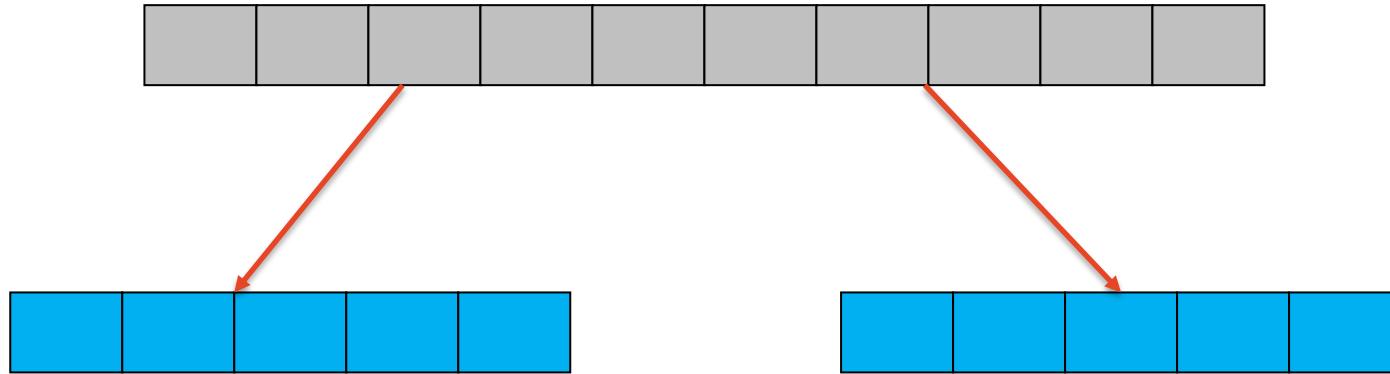
Subproblem of constant size (usually 0 or 1 elements) for which you can compute the solution explicitly



easy to compute solution

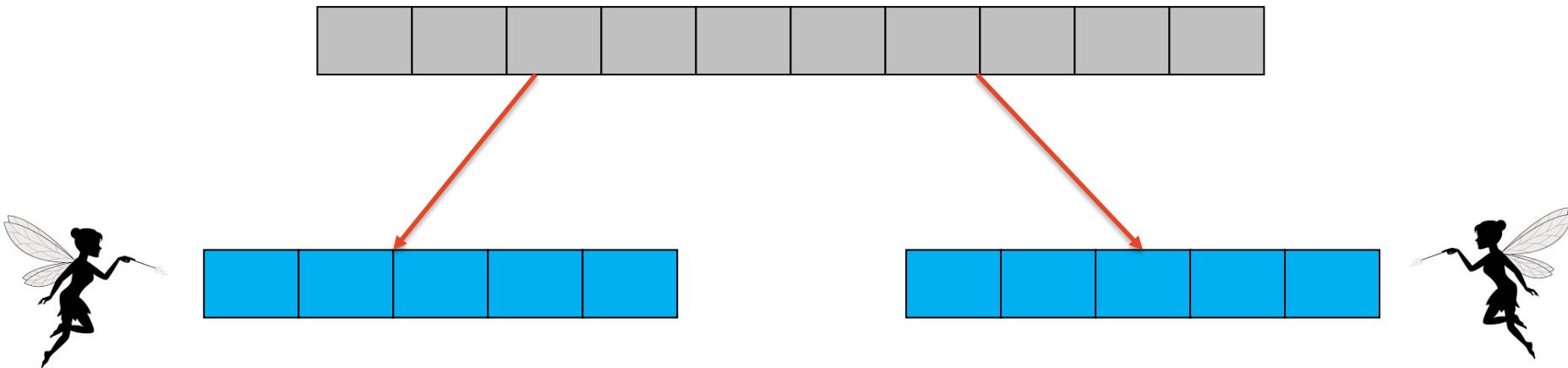
Divide and Conquer

1. **Divide** If it is a base case, solve directly, otherwise break up the problem into several parts.



Divide and Conquer

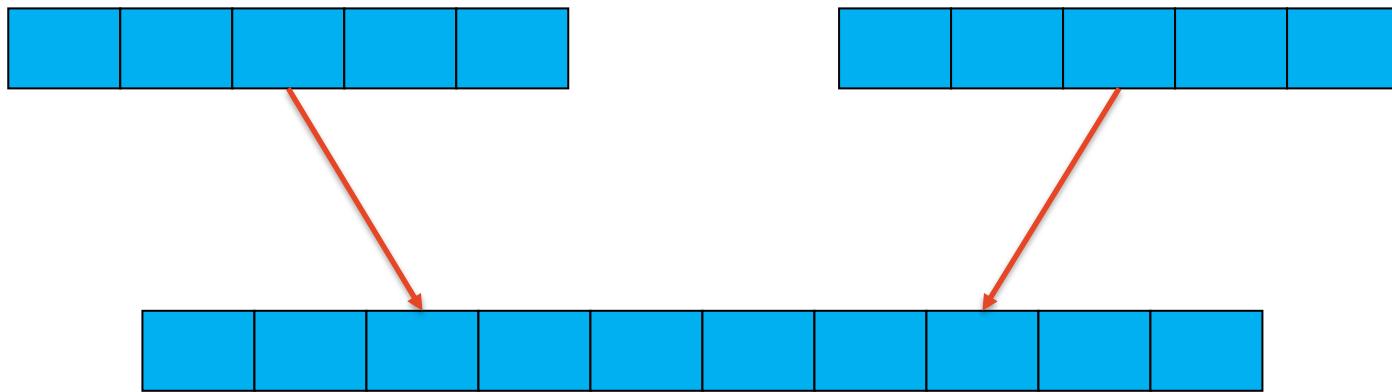
2. Recur/Delegate Recursively solve each part [each sub-problem].



The sub-problems are solved by the Recursion Fairy (similar to induction hypothesis), so we don't have to worry about them.

Divide and Conquer

3. **Conquer** Combine the solutions of each part into the overall solution.



Searching Sorted Array

Given A sorted sequence S of n numbers a_0, a_1, \dots, a_{n-1} stored in an array A[0, 1, ..., n - 1].

Problem Given a number x, is x in S?

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

Searching: Naïve Approach

Problem Given a number x , is x in S ?

Idea Check every element in turn to see if it is equal to x .

```
for e in S do
    if e equals x then
        return "Yes"
return "No"
```



Found an element equal to x in S

There was no element equal to x in S

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

Running Time $O(n)$

Binary Search in sorted A[0 to n-1]

1. If the array is empty, then return “No”
2. Compare x to the middle element, namely $A[\lfloor n/2 \rfloor]$
3. If this middle element is x , then return “Yes”
4. When the middle element is not x : if $A[\lfloor n/2 \rfloor] > x$, then recursively search $A[0$ to $\lfloor n/2 \rfloor - 1]$
5. if $A[\lfloor n/2 \rfloor] < x$, then recursively search $A[\lfloor n/2 \rfloor + 1$ to $n-1]$

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

Heads up: pseudocode textbook uses indexing from 1 to n , not 0 to $n-1$

Binary Search

- Example, search for $x=5$

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

Binary Search

- Example, search for $x=5$

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

A[6]

Binary Search

- Example, search for $x=5$

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

$$A[6] = 25 > 5 = x$$

Binary Search

- Example, search for $x=5$

0	2	5	7	12	22
---	---	---	---	----	----

A[3]

25	37	39	50	55	80
---------------	----	----	----	----	----

Binary Search

- Example, search for $x=5$

0	2	5	7	12	22
---	---	---	---	----	----

25	37	39	50	55	80
---------------	----	----	----	----	----

$$A[3] = 7 > 5 = x$$

Binary Search

- Example, search for $x=5$

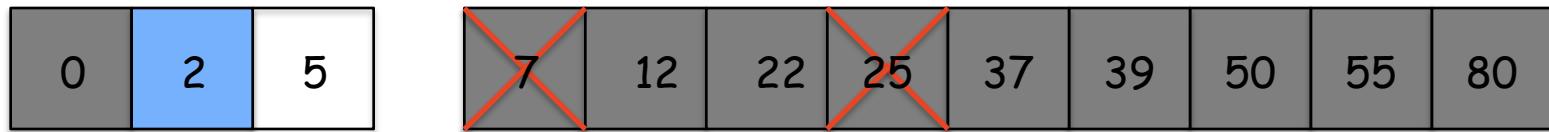
0	2	5
---	---	---



$A[1]$

Binary Search

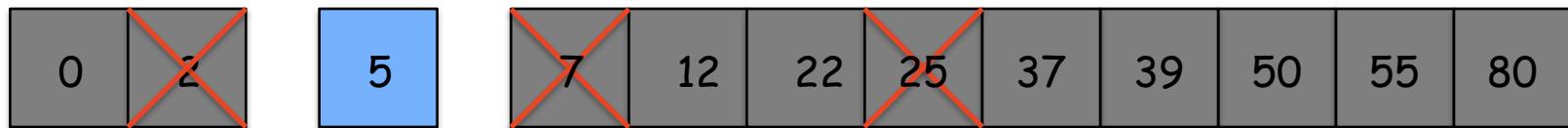
- Example, search for $x=5$



$$A[1] = 2 < 5 = x$$

Binary Search

- Example, search for $x=5$



$A[2]$

Binary search correctness

Invariant: If x is in A before the divide step, then x is in A after the divide step

- if $A[\lfloor n/2 \rfloor] > x$, then x must be $A[0 \text{ to } \lfloor n/2 \rfloor - 1]$
- if $A[\lfloor n/2 \rfloor] < x$, then x must be in $A[\lfloor n/2 \rfloor + 1 \text{ to } n - 1]$

Every divide step leads to a smaller array.

Thus, if x in A , we will eventually inspect its position due to the invariant and return “Yes”.

Thus, if x in not in A , then eventually we reach the empty array and return “No”.

Recurrence formula

An easy way to analyze the time complexity of a divide-and-conquer algorithm is to define and solve a recurrence

Let $T(n)$ be the running time of the algorithm, we need to find out:

- Divide step cost in terms of n
- Recur step(s) cost in terms of $T(\text{smaller values})$
- Conquer step cost in terms of n

Together with information about the base case, we can set up a recurrence for $T(n)$ and then solve it.

$$T(n) = \begin{cases} \text{“Recur” + “Divide and Conquer”} & \text{for } n > 1 \\ \text{“Base case” (typically } O(1)\text{)} & \text{for } n = 1 \end{cases}$$

Binary search on an array complexity analysis

Divide step (find middle and compare to x) takes $O(1)$

Recur step (solve left or right subproblem) takes $T(n/2)$

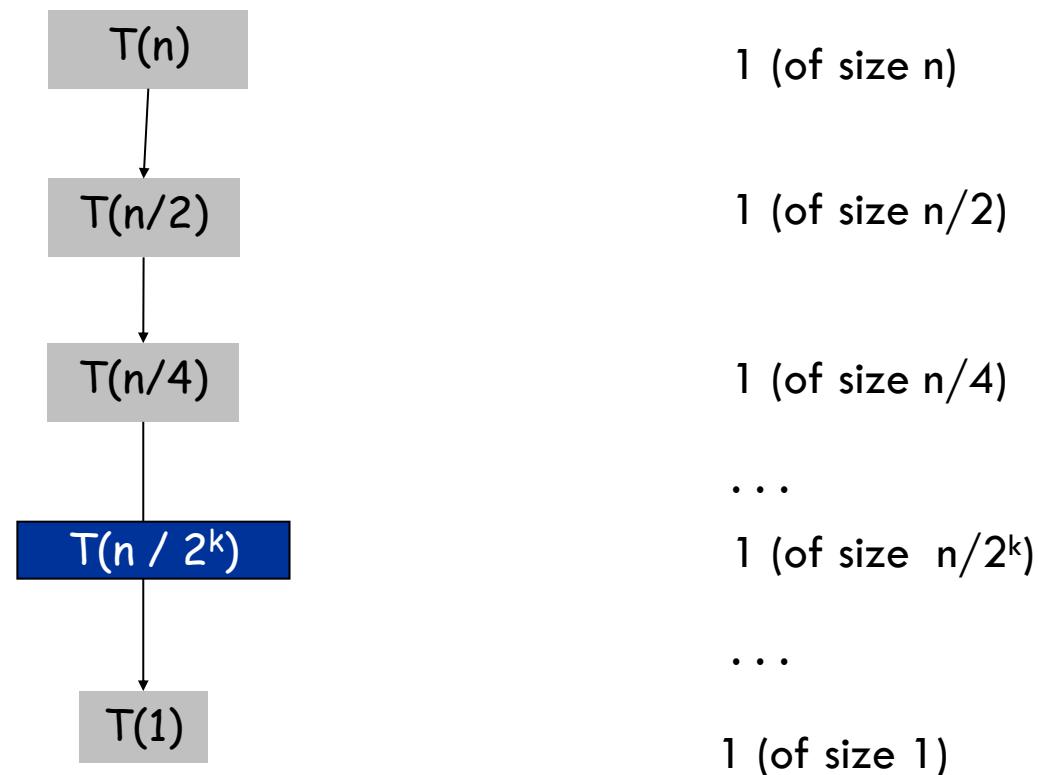
Conquer step (return answer from recursion) takes $O(1)$

Now we can set up the recurrence for $T(n)$:

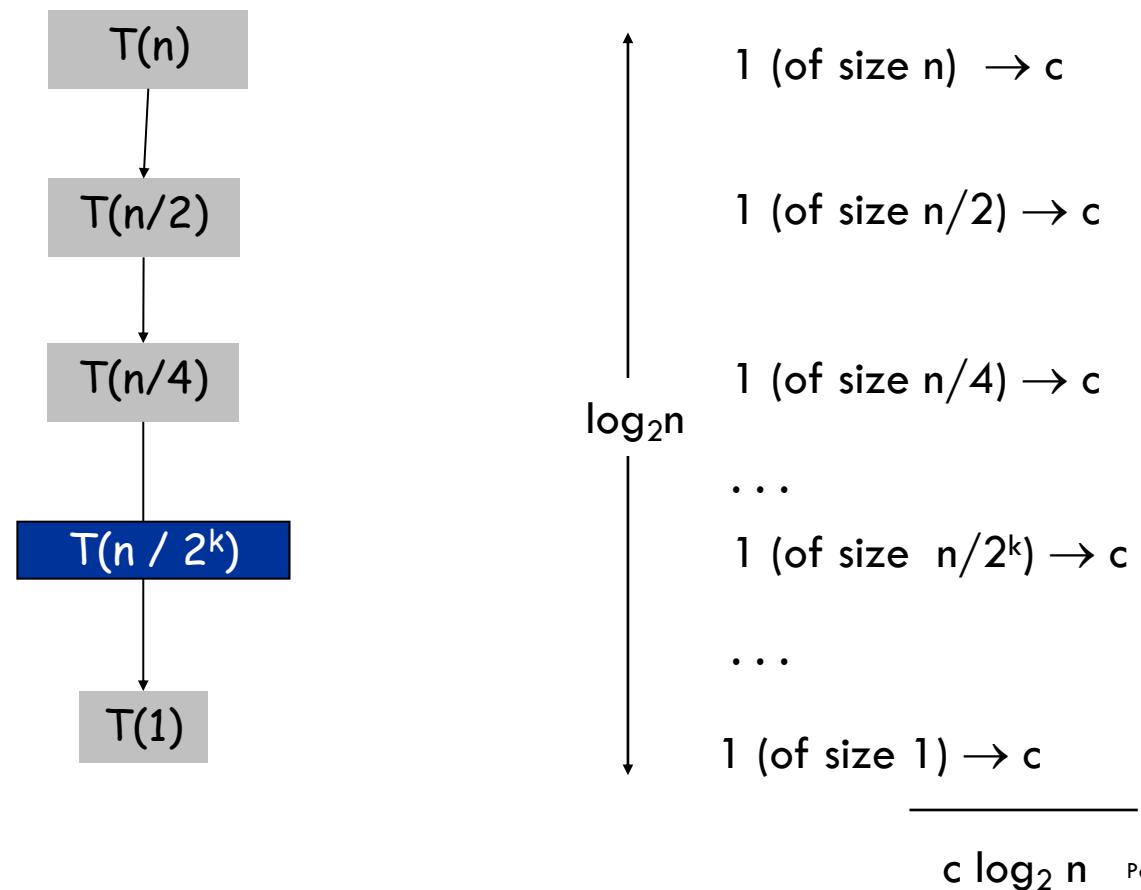
$$T(n) = \begin{cases} T(n/2) + O(1) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(\log n)$, since we can only halve the input $O(\log n)$ times before reaching a base case

Proof by unrolling



Proof by unrolling



Binary search on a linked list complexity analysis

Divide step (find middle and compare to x) takes $O(n)$

Recur step (solve left or right subproblem) takes $T(n/2)$

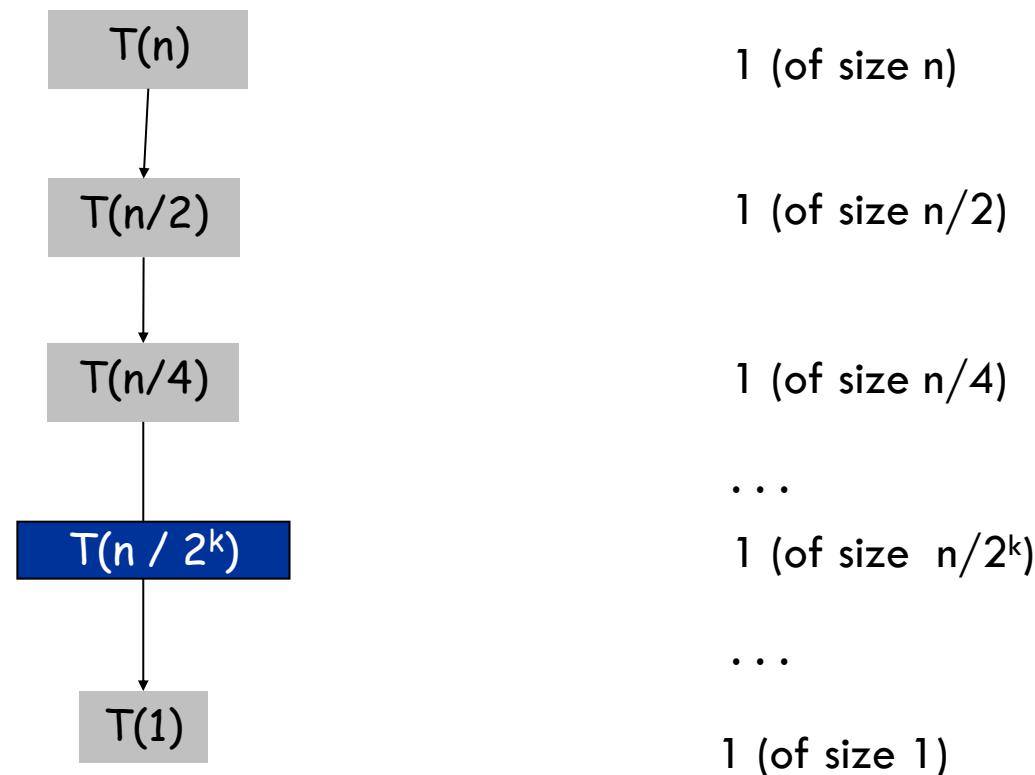
Conquer step (return answer from recursion) takes $O(1)$

Now we can set up the recurrence for $T(n)$:

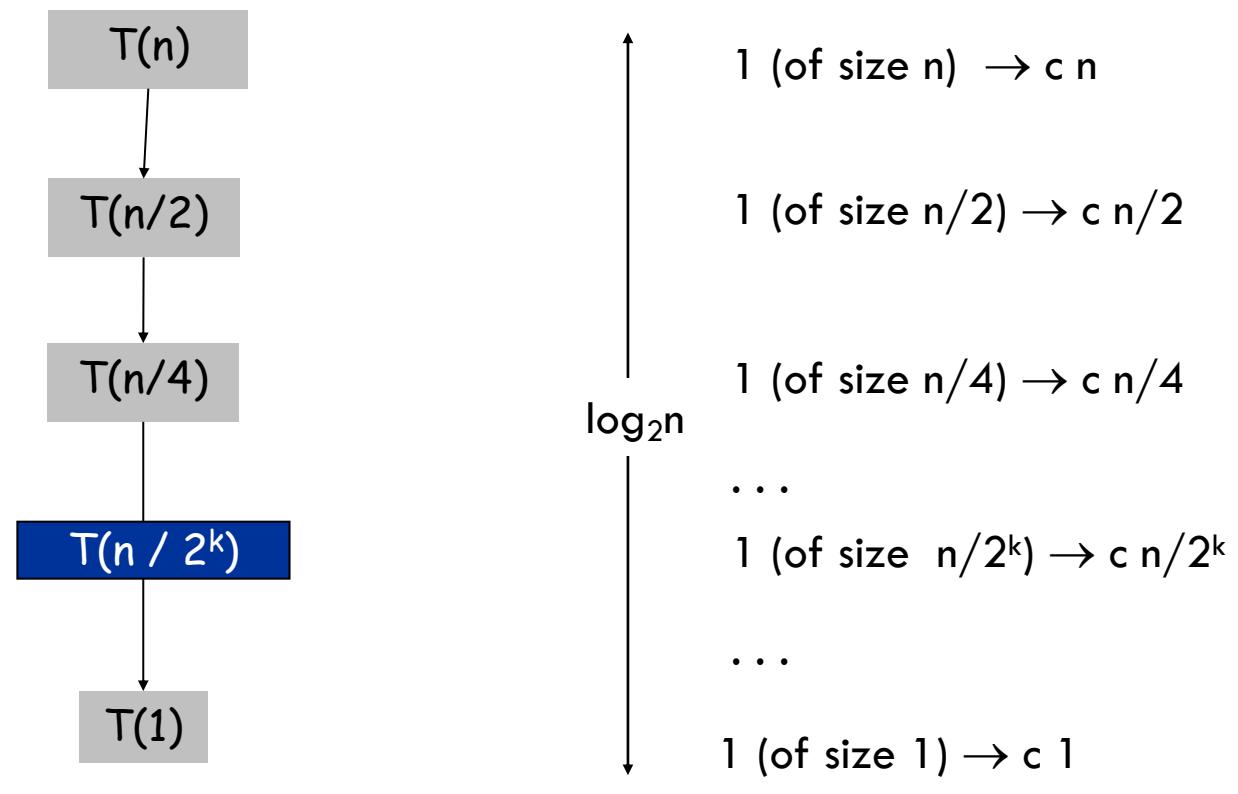
$$T(n) = \begin{cases} T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n)$, since to access the next index we end up with $n/2 + n/4 + n/8 + \dots$

Proof by unrolling



Proof by unrolling



Merge-Sort

1. **Divide** the array into two halves.
2. **Recur** recursively sort each half.
3. **Conquer** two sorted halves to make a single sorted array.

1	12	5	16	19	7	23	6	13	20
---	----	---	----	----	---	----	---	----	----

1	12	5	16	19	7	23	6	13	20
---	----	---	----	----	---	----	---	----	----

Divide

1	5	12	16	19	6	7	13	20	23
---	---	----	----	----	---	---	----	----	----

Recur

1	5	6	7	12	13	16	19	20	23
---	---	---	---	----	----	----	----	----	----

Conquer

Merge-Sort pseudocode

```
def merge_sort(S):
    # base case
    if |S| < 2 then
        return S

    # divide
    mid ← ⌊|S|/2⌋
    left ← S[:mid]      # doesn't include S[mid]
    right ← S[mid:]     # includes S[mid]

    # recur
    sorted_left ← merge_sort(left)
    sorted_right ← merge_sort(right)

    # conquer
    return merge(sorted_left, sorted_right)
```

How?

Merge

Input Two sorted lists.

Output A new merged sorted list.

To merge, we use:

- $O(n)$ comparisons.
- An array to store our results.



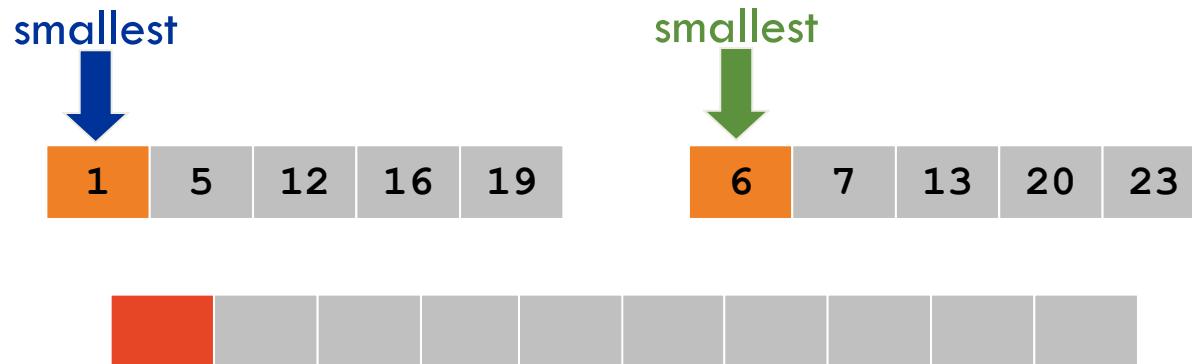
Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.

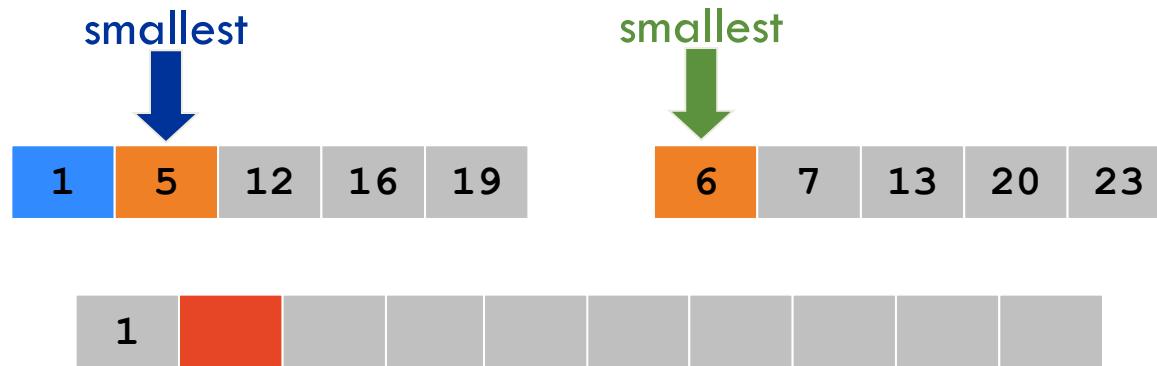


Result:

Merge

Merge Algorithm

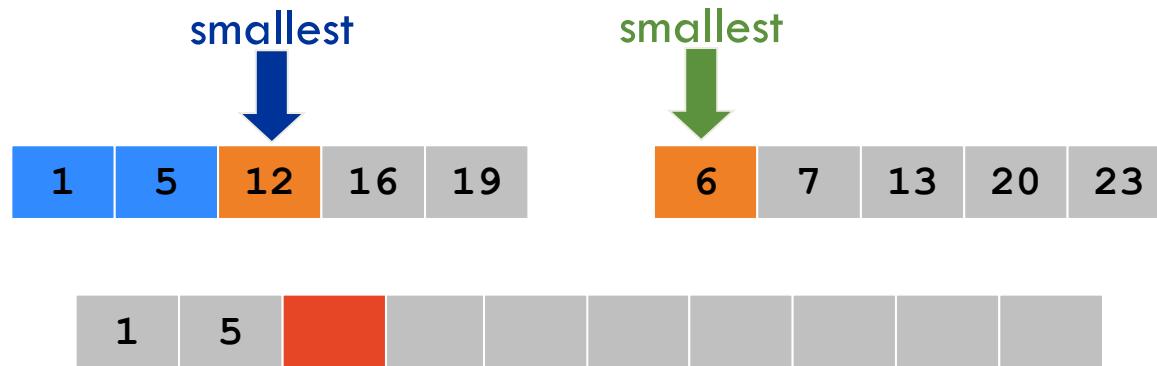
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



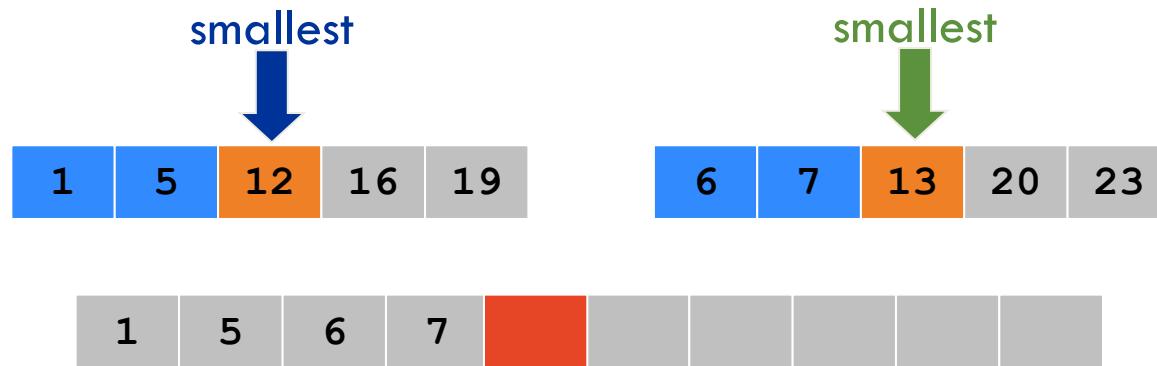
Result:



Merge

Merge Algorithm

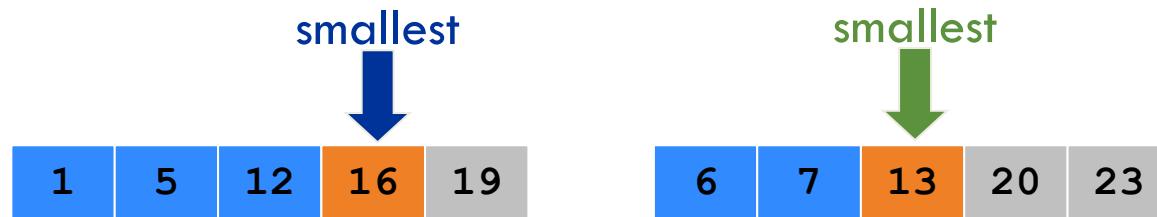
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



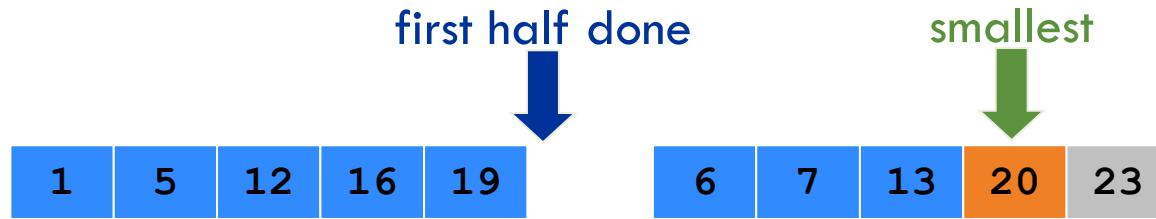
Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



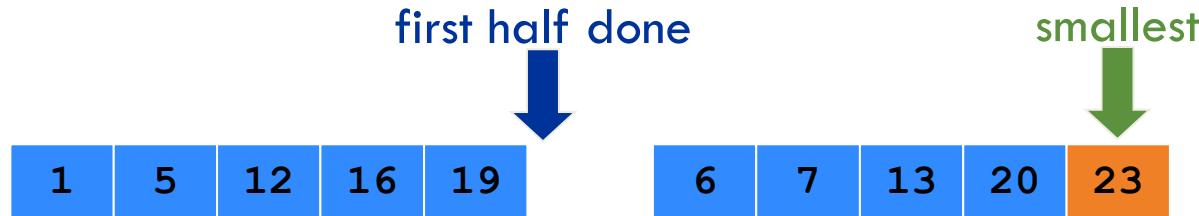
Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Result:



Merge: Implementation

```
def merge(L, R):
    result ← array of length (|L| + |R|)
    l, r ← 0, 0
    while l + r < |result| do
        index ← l + r
        if r ≥ |R| or (l < |L| and L[l] < R[r]) then
            result[index] ← L[l]
            l ← l + 1
        else
            result[index] ← R[r]
            r ← r + 1
    return result
```

Merge: Correctness

Induction hypothesis:

- After the i -th iteration, our result contains the i smallest elements in sorted order

Base case:

- After 0 iterations, our result is empty, so it contains the 0 smallest elements in sorted order

Induction:

- Assume IH after iteration k , to prove it after iteration $k+1$
- Since both halves are sorted and we add the smallest element not already in result, result now contains the $k+1$ smallest elements
- Sorted order follows from the fact that both halves are sorted, thus adding the smallest element implies sorted order of result

Merge-Sort

1. **Divide** array into two halves.
2. **Recur** Recursively sort each half.
3. **Conquer** Merge two sorted halves to make a sorted whole.

1	12	5	16	19	7	23	6	13	20
---	----	---	----	----	---	----	---	----	----

1	12	5	16	19	7	23	6	13	20
---	----	---	----	----	---	----	---	----	----

divide

1	5	12	16	19	6	7	13	20	23
---	---	----	----	----	---	---	----	----	----

recur

1	5	6	7	12	13	16	19	20	23
---	---	---	---	----	----	----	----	----	----

conquer

Merge-Sort: Correctness

Induction hypothesis:

- Merge-Sort correctly sorts an array of size i

Base case:

- If our array has size 0 or 1, it's already sorted

Induction:

- Assume IH for all arrays up to size k , to prove it for array of size $k+1$
- Splitting the array in half gives us two array of size at most k , so by IH those are sorted correctly
- We proved that given two sorted arrays, Merge returns a correctly sorted array containing the elements of both arrays
- Hence, by running Merge on the two stored halves, we sort the original array

Merge sort complexity analysis

Divide step (find middle and split) takes $O(n)$

Recur step (solve left and right subproblem) takes $2 T(n/2)$

Conquer step (merge subarrays) takes $O(n)$

Now we can set up the recurrence for $T(n)$:

$$T(n) = \begin{cases} 2 T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n \log n)$

Solving recurrences by unrolling

General strategy:

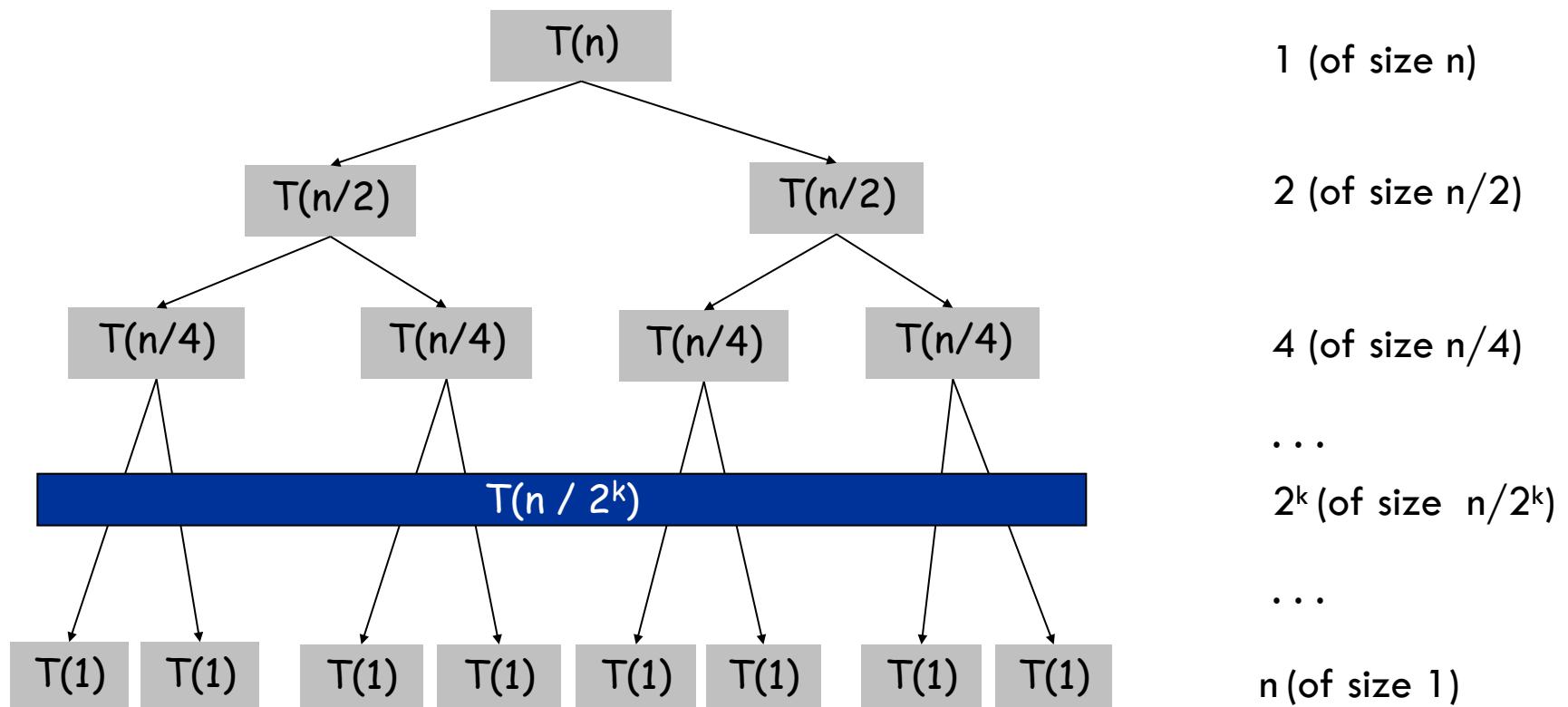
- Analyze first few levels
- Identify the pattern for a generic level
- Sum up over all levels

To verify the solution, we can substitute guess into the recurrence and prove it formally using induction

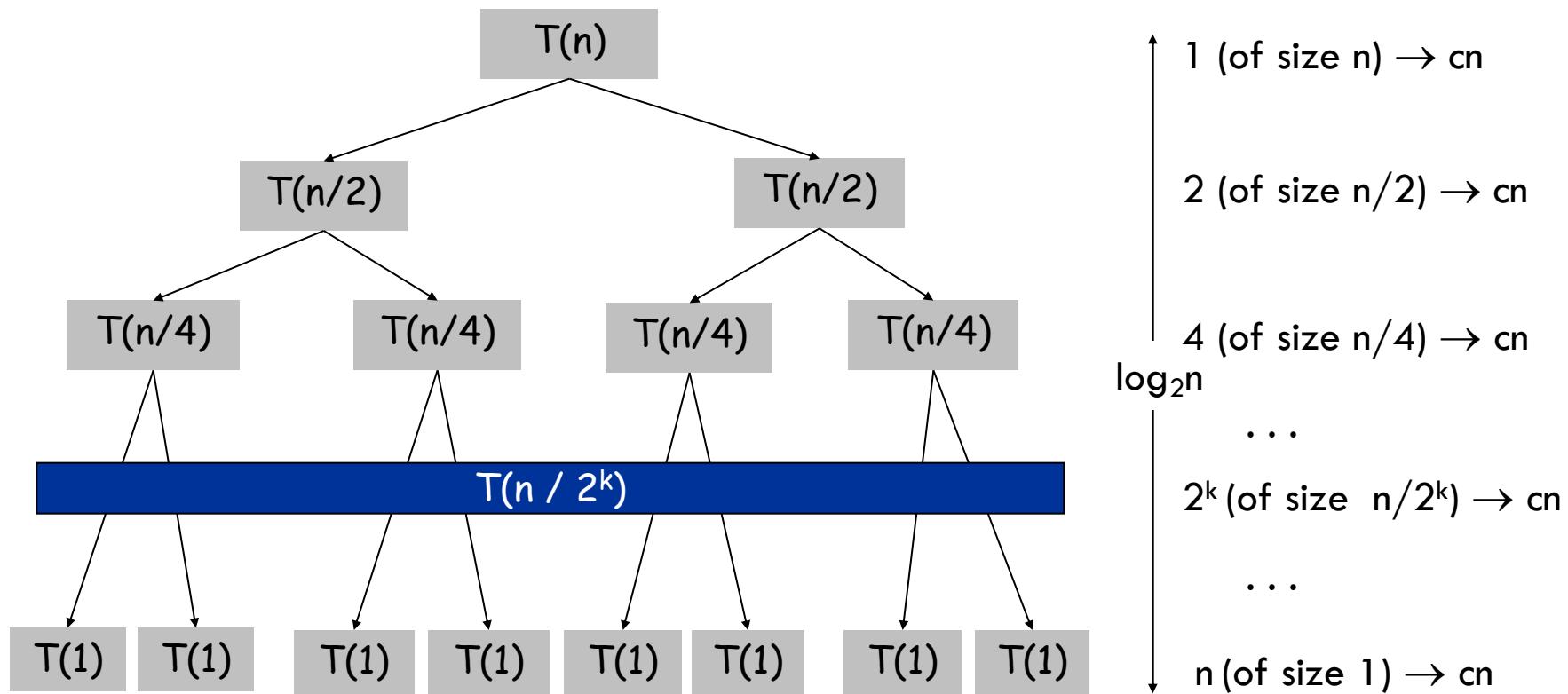
For Merge sort this method yields $T(n) = O(n \log n)$

There is a “Master theorem” (see textbook) that can handle most recurrences of interest, but unrolling is enough for our purposes

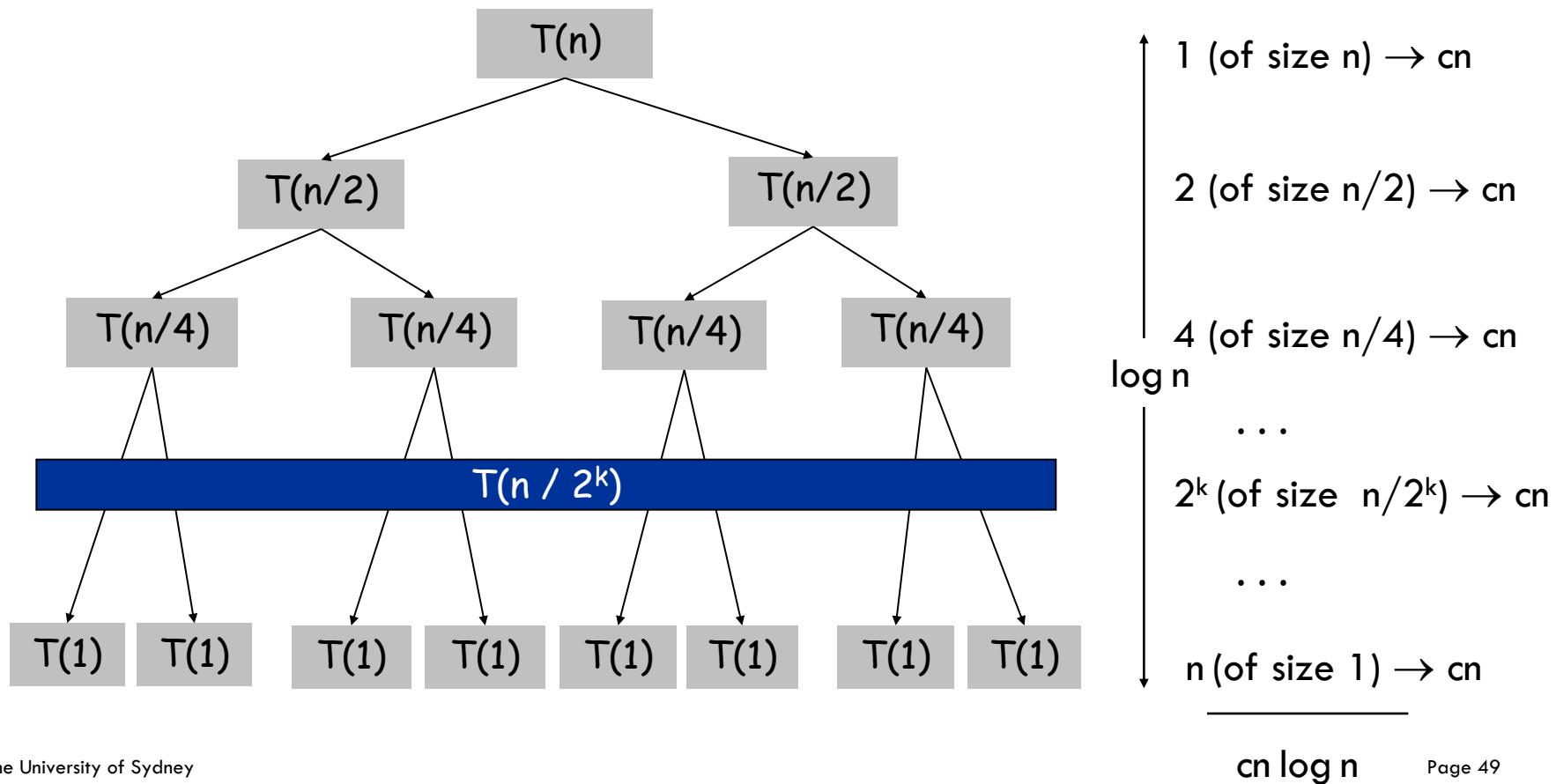
Proof by unrolling



Proof by unrolling



Proof by unrolling

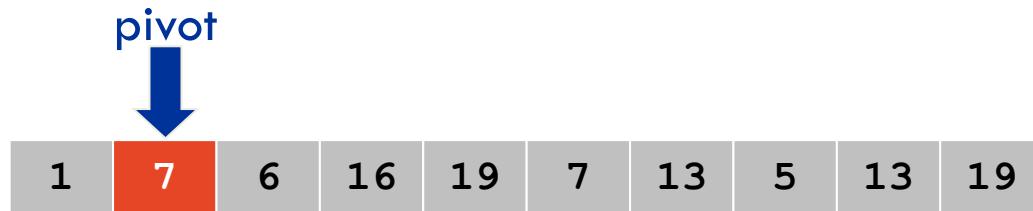


Some recurrence formulas with solutions

Recurrence	Solution
$T(n) = 2 T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = 2 T(n/2) + O(\log n)$	$T(n) = O(n)$
$T(n) = 2 T(n/2) + O(1)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(n)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$

Quick sort

1. **Divide** Choose a random element from the list as the **pivot**
Partition the elements into 3 lists:
(i) less than, (ii) equal to and (iii) greater than the **pivot**
2. **Recur** Recursively sort the **less than** and **greater than** lists
3. **Conquer** Join the sorted 3 lists together



Quick sort complexity analysis

Divide step (pick pivot and split) takes $O(n)$

Recur step (solve left and right subproblem) takes $T(n_L) + T(n_R)$

Conquer step (merge subarrays) takes $O(n)$

Now we can set up the recurrence for $T(n)$:

$$E[T(n)] = \begin{cases} E[T(n_L) + T(n_R)] + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $E[T(n)] = O(n \log n)$ expected time
(details available on the textbook but not examinable)

Interlude: Comparison sorting lower bound

So far we've seen many sorting algorithms. Some run in $O(n^2)$ time while others run in $O(n \log n)$ time.

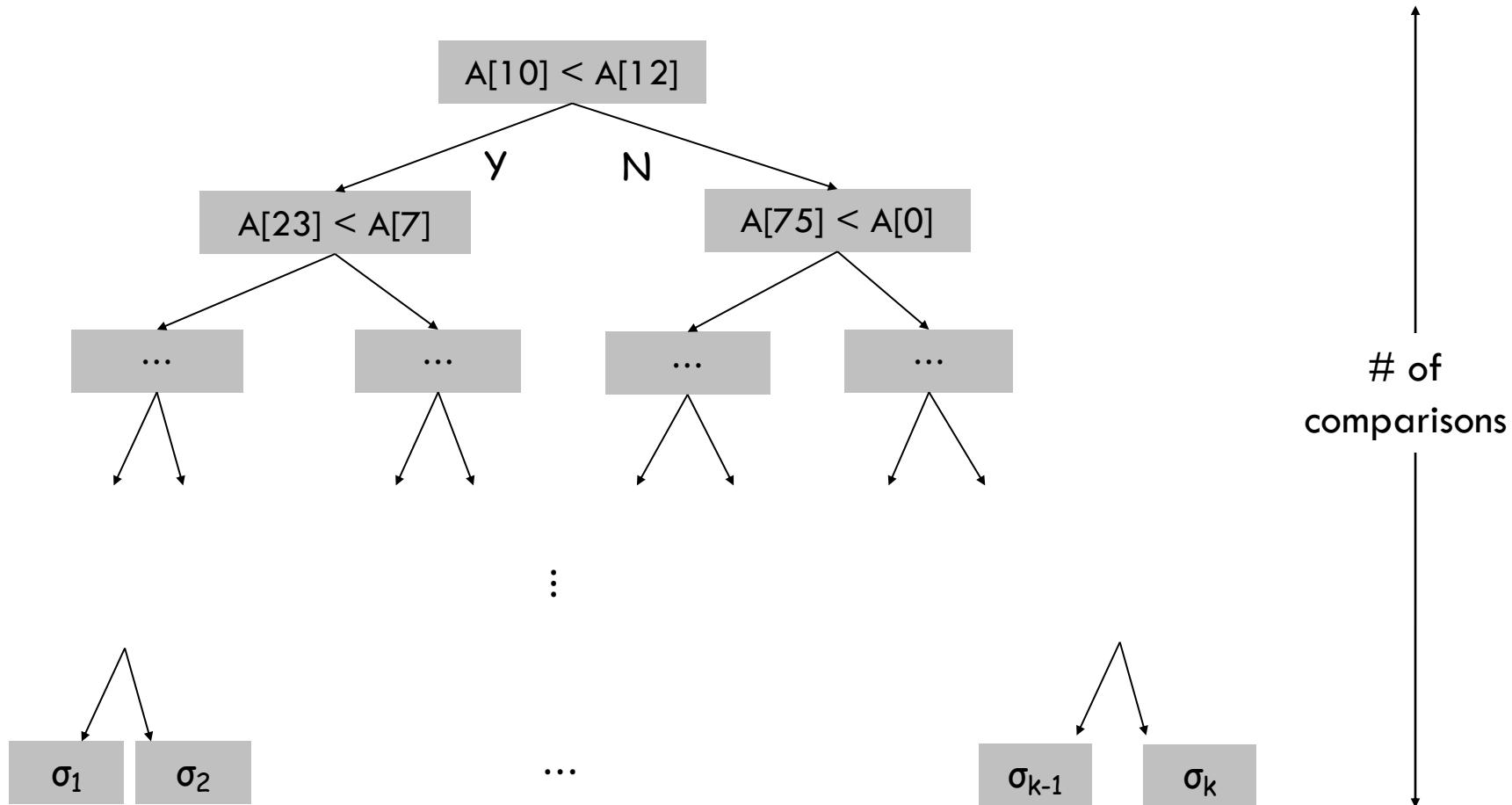
These algorithms work by performing pair-wise comparisons between elements of the sequence we are trying to sort

Such algorithms can be viewed as a decision tree where:

- each internal node compares two indices of the input array
- each external node corresponds to a permutation of $\{1, \dots, n\}$

The height of the decision tree is a lower bound on the running time of the algorithm, since it only counts number of comparisons

Decision tree



The output of a leaf is $A[\sigma(1)], A[\sigma(2)], \dots, A[\sigma(n)]$

Interlude: Comparison sorting lower bound

Fact: Comparison-based sorting algorithms take $\Omega(n \log n)$ time

Proof:

The decision tree associated with a comparison-based sorting algorithm is binary and has $n!$ external nodes. Thus the height is $\log n!$ which is $\Omega(n \log n)$

$$\begin{aligned}\log n! &= \log (n * (n-1) * \dots * 1) \\&= \log n + \log(n-1) + \dots + \log 1 \\&> n/2 * (\log n/2) \\&= \Omega(n \log n)\end{aligned}$$

Remember

Important:

Simply using Merge-Sort in your algorithm doesn't make your algorithm a divide and conquer algorithm.

Example:

A greedy algorithm first sorts the input in some way and then processes the items one by one in that order. Using Merge-Sort for the sorting step doesn't change the fact that the algorithm computes the solution in a greedy way.

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2123

Data structures and Algorithms

Lecture 11: Divide and Conquer

[GT 11 and 9]

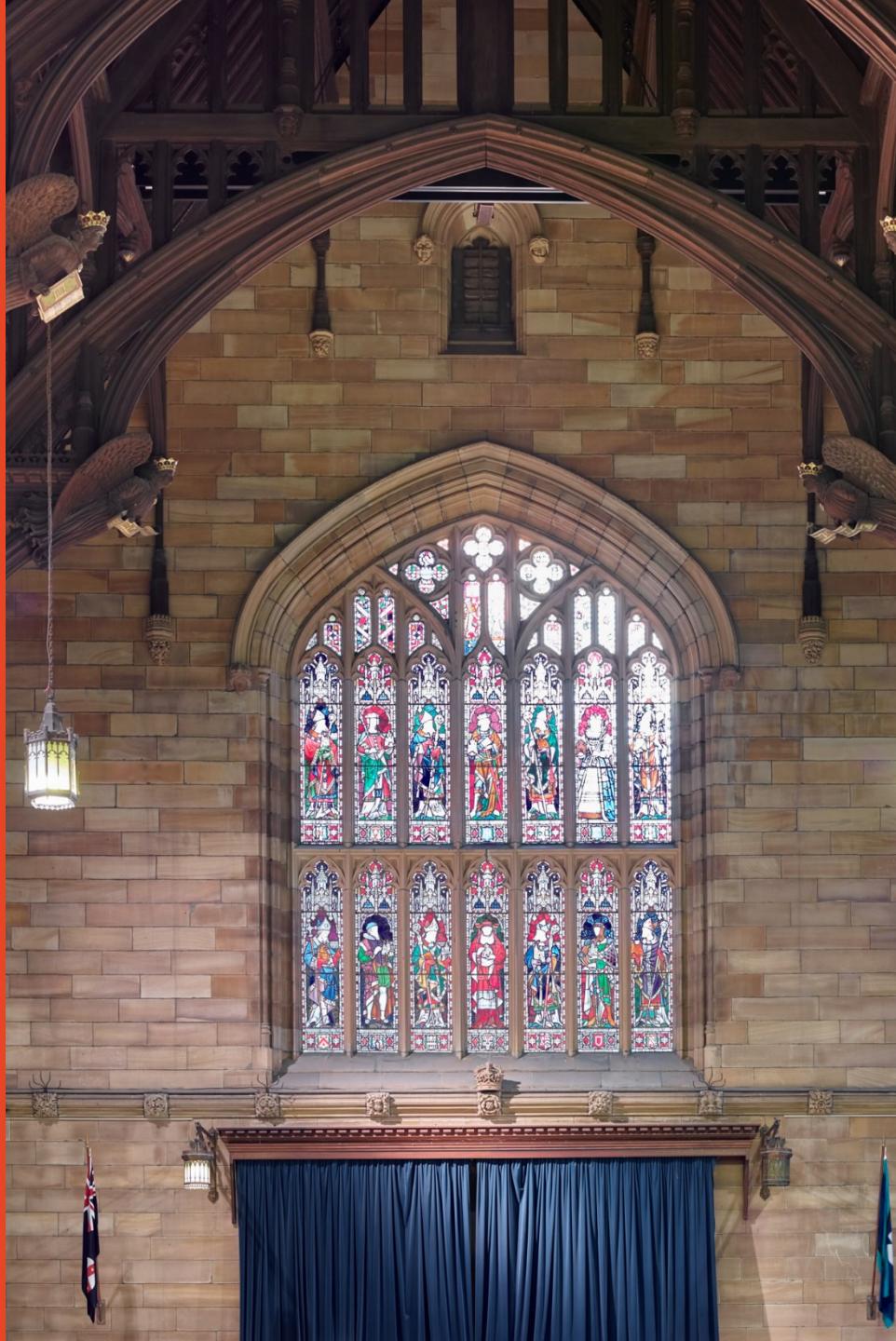
Dr. André van Renssen

School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



THE UNIVERSITY OF
SYDNEY



Divide and Conquer

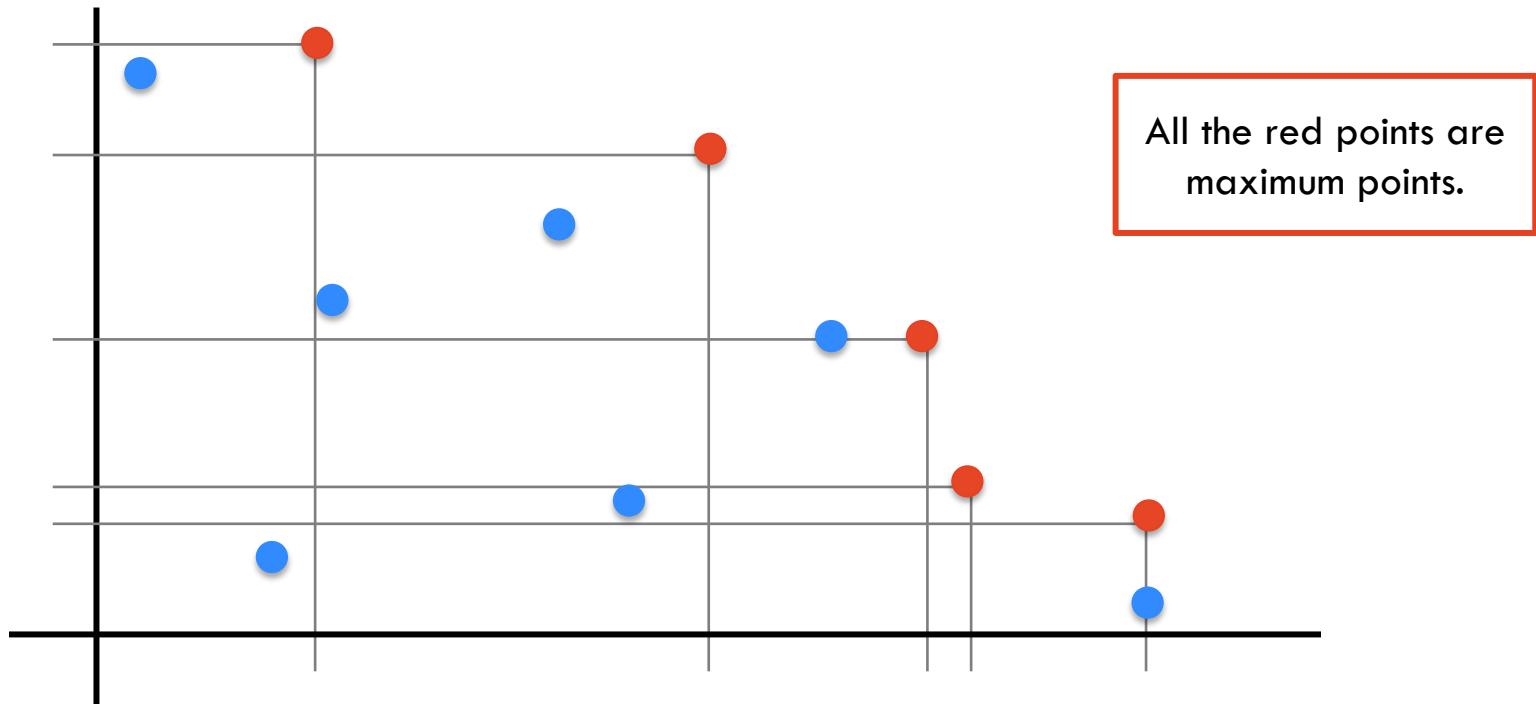
Divide and Conquer algorithms can normally be broken into these three parts:

1. **Divide** If it is a base case, solve directly, otherwise break up the problem into several parts.
2. **Recur/Delegate** Recursively solve each part [each sub-problem].
3. **Conquer** Combine the solutions of each part into the overall solution.

Maxima-Set (Pareto frontier)

Definition A point is maximum in a set if all other points in the set have either a smaller x- or smaller y-coordinate.

Problem Given a set S of n distinct points in the plane (2D), find the set of all maximum points.



Maxima-Set: Naïve Solution

Idea: Check every point (one at a time) to see if it is a maximum point in the set S .

To check if point p is a maximum point in S :

```
for q in S do
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y then
        return "No"
return "Yes"
```



There is a point q that dominates p

Maxima-Set: Naïve Solution

Idea: Check every point (one at a time) to see if it is a maximum point in the set S.

To check if point p is a maximum point in S:

```
for q in S do
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y then
        return "No"
return "Yes"
```

Naïve algorithm to find the maxima-set of S:

```
maximaSet ← empty list
for p in S do
    if p is a maximum point in S then
        add p to the maximaSet
return maximaSet
```

Maxima-Set: Naïve Solution

Idea: Check every point (one at a time) to see if it is a maximum point in the set S .

To check if point p is a maximum point in S :

```
for q in S do
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y then
        return "No"
return "Yes"
```

$O(n)$

Naïve algorithm to find the maxima-set of S :

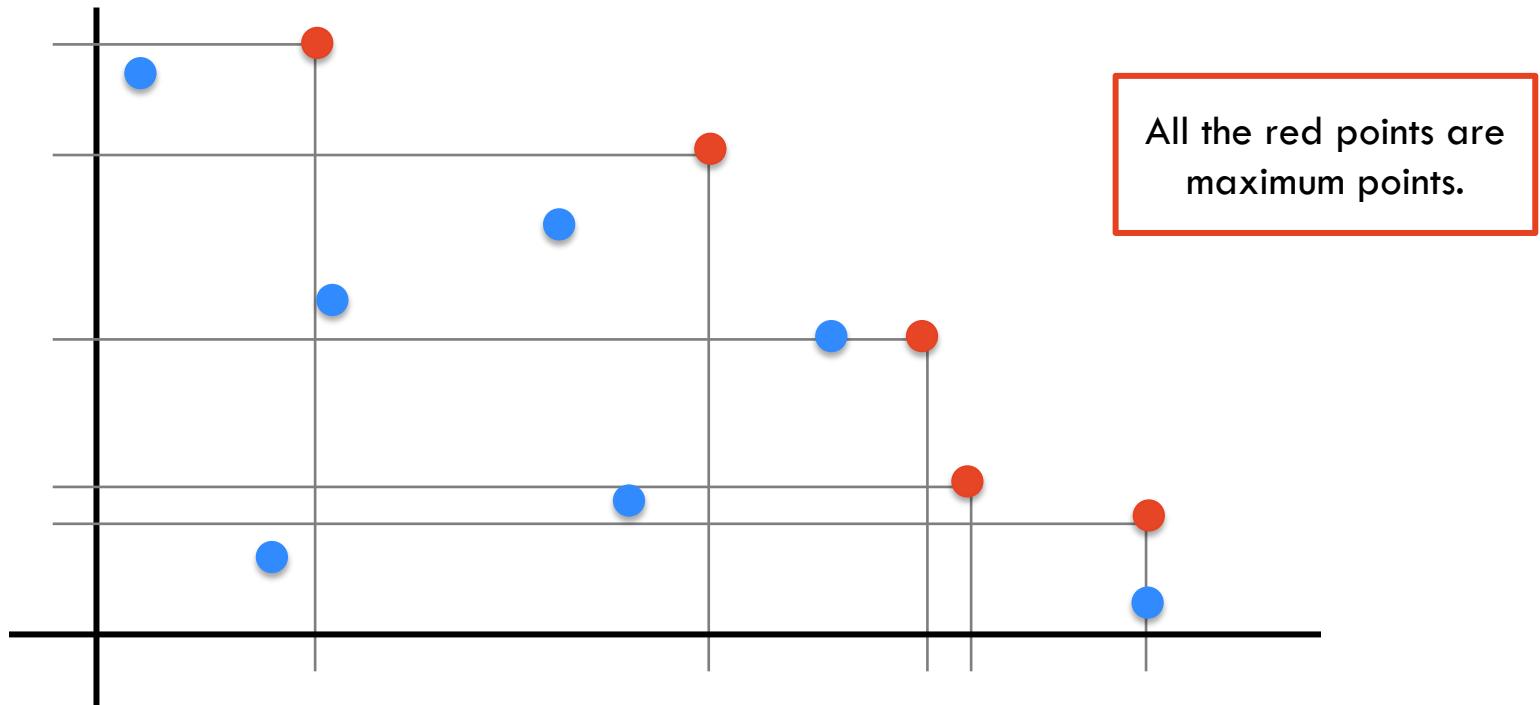
```
maximaSet ← empty list
for p in S do
    if p is a maximum point in S then
        add p to the maximaSet
return maximaSet
```

$O(n^2)$

Maxima-Set

Definition A point is maximum in a set if all other points in the set have either a smaller x or smaller y coordinate.

Problem Given a set S of n distinct points in the plane (2D), find the set of all maximum points.



Maxima-Set

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once. Break ties in x by sorting by increasing y coordinate.

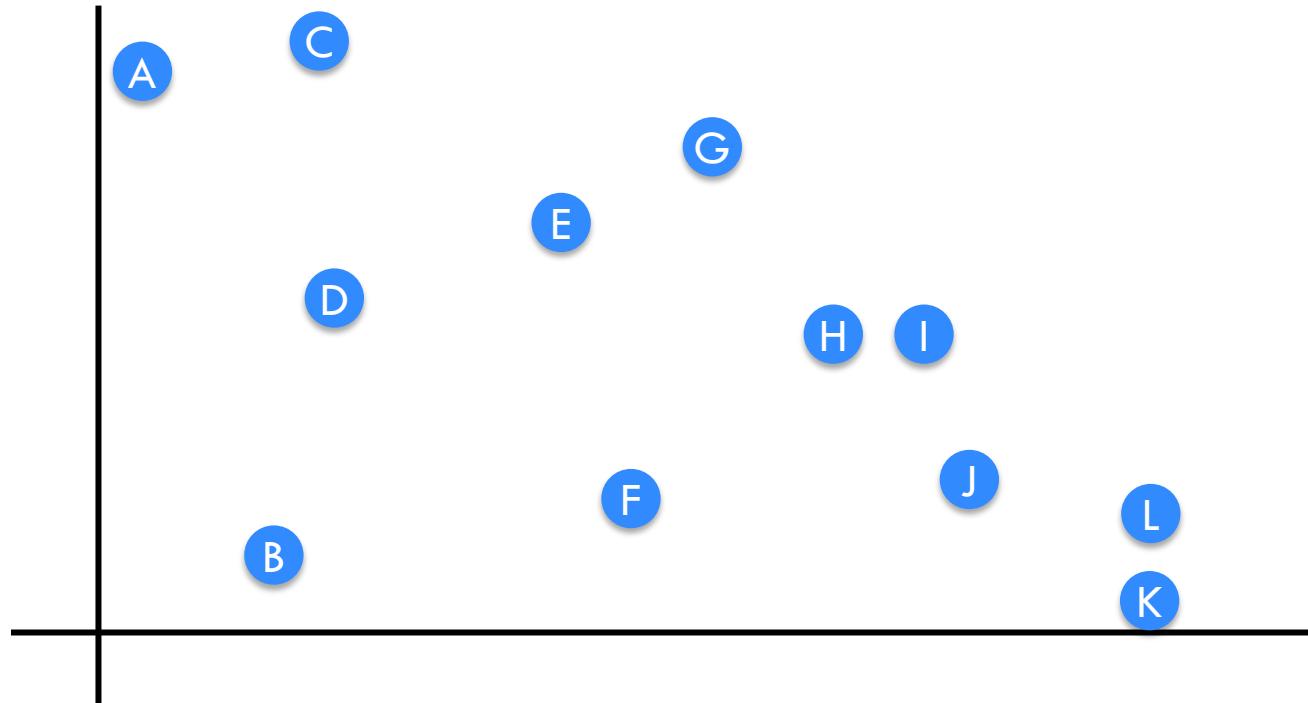
Divide sorted array into two halves.

Recur recursively find the MS of each half.

Conquer compute the MS of the union of Left and Right MS

Maxima-Set

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once.
Break ties in x by sorting by increasing y coordinate.

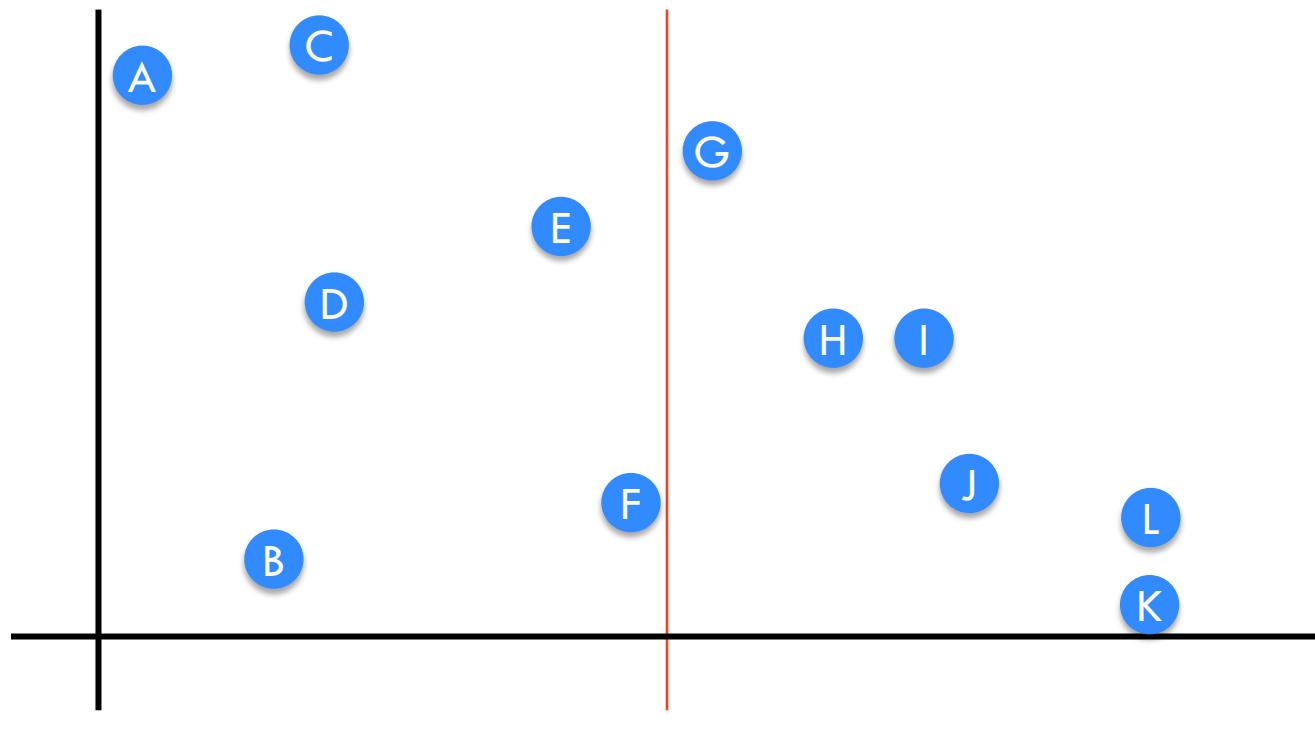


Sorted Points

A	B	C	D	E	F	G	H	I	J	K	L
---	---	---	---	---	---	---	---	---	---	---	---

Maxima-Set

Divide array into two halves.

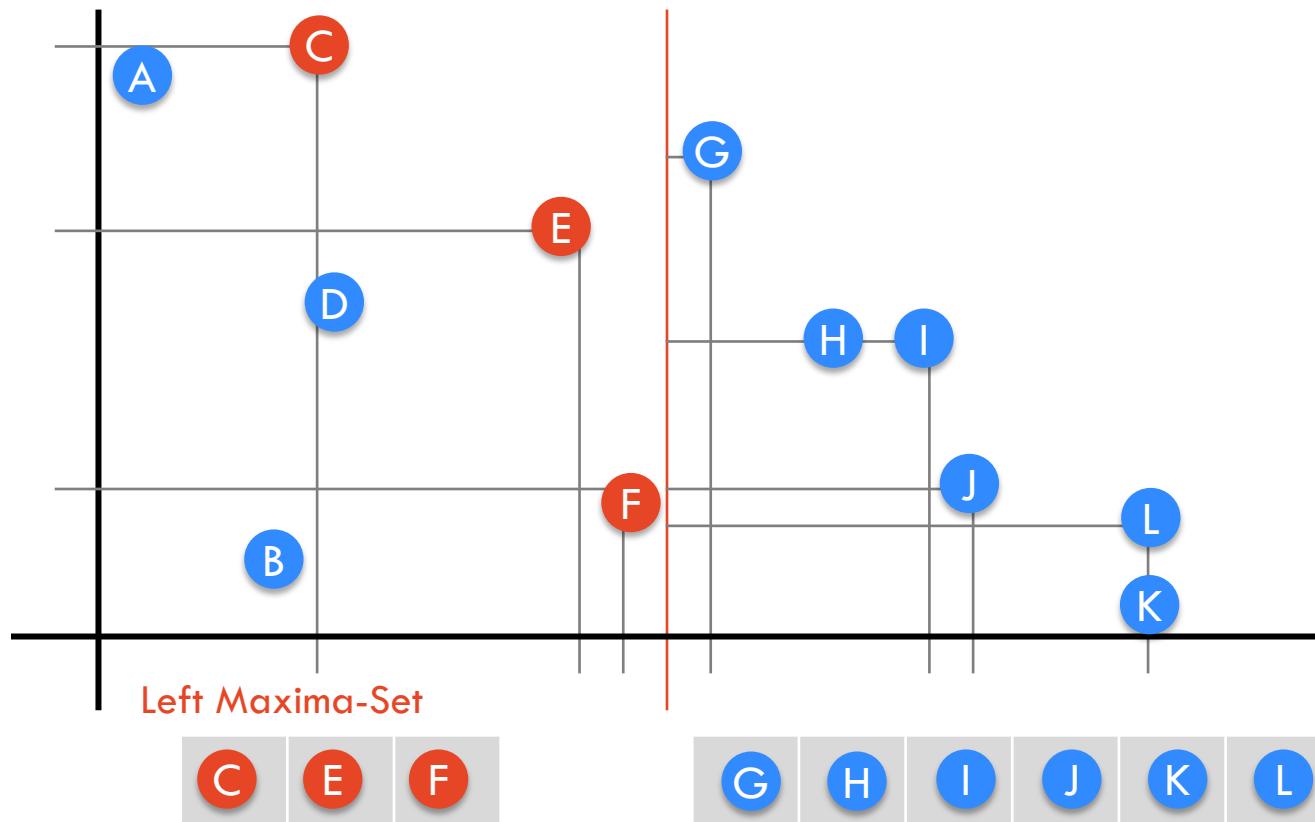


Sorted Points



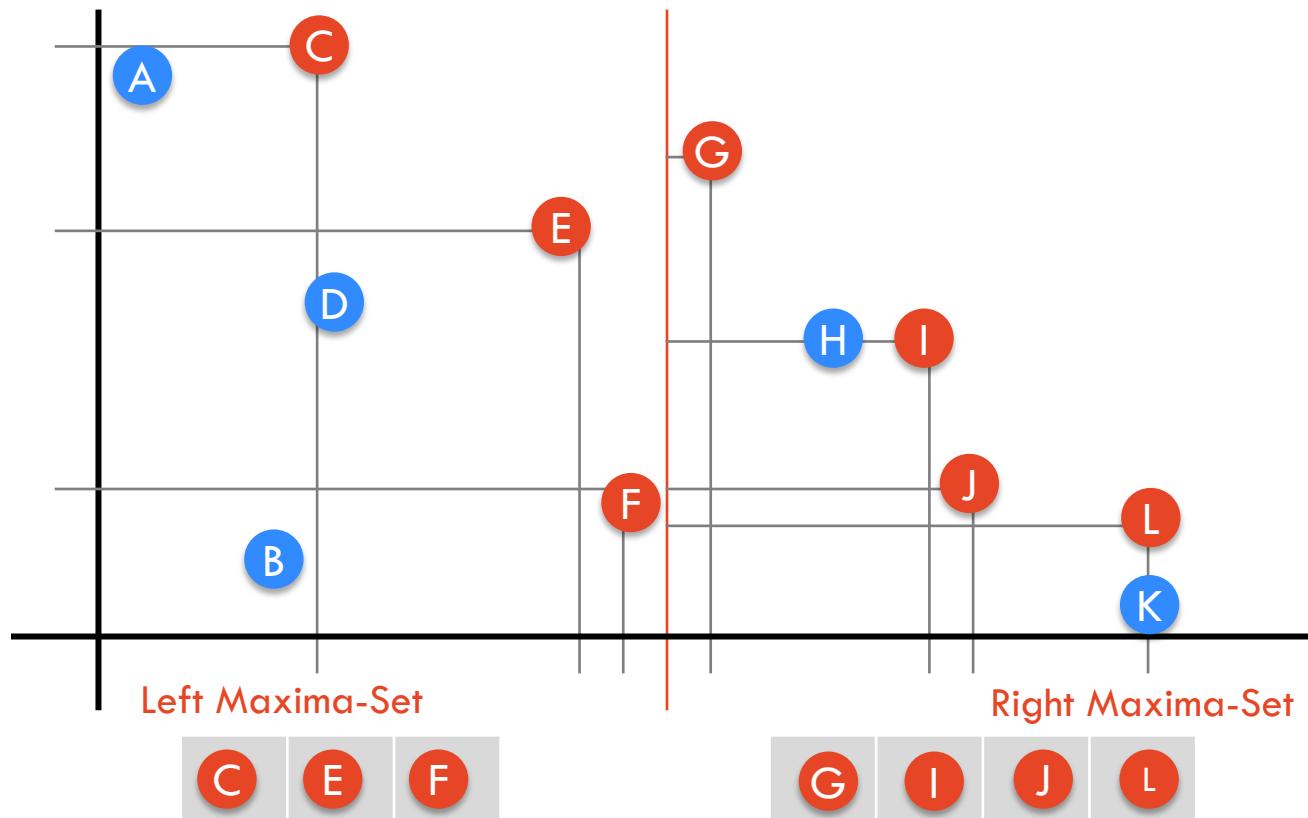
Maxima-Set

Recur recursively find the Maxima-Set of each half.



Maxima-Set

Recur recursively find the Maxima-Set of each half.



Maxima-Set

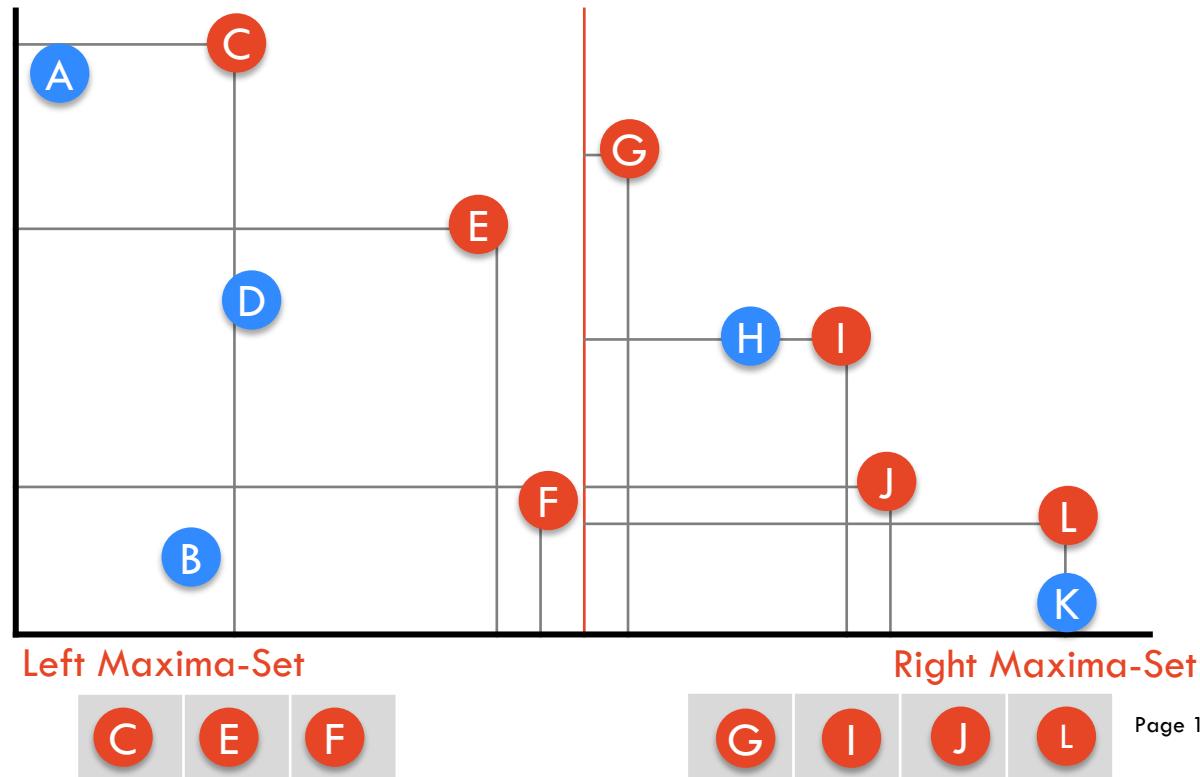
Conquer

1. Find the highest point p in the Right MS

$$p = G$$

Observations:

1. Every point in MS of the whole is in Left MS or Right MS
2. Every point in Right MS is in MS of the whole
3. Every point in Left MS is either in MS of the whole or is dominated by p



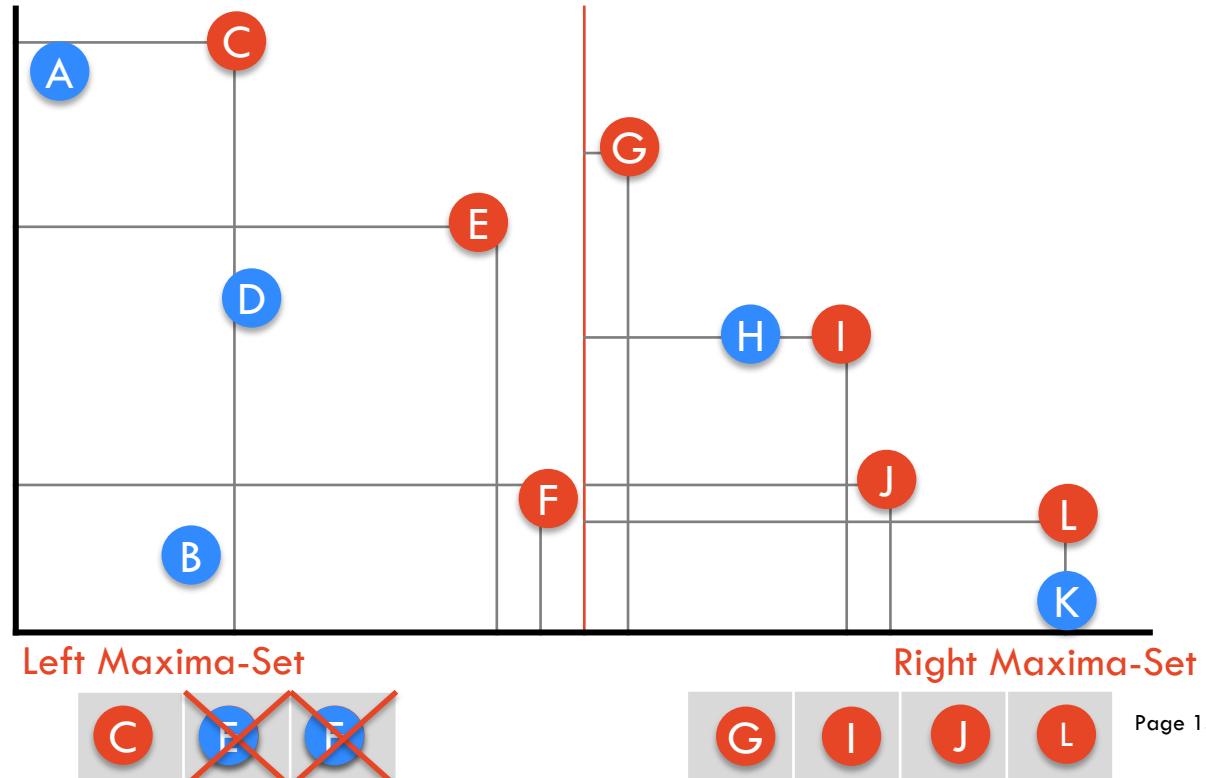
Maxima-Set

Conquer

1. Find the highest point p in the Right MS
2. Compare every point q in the Left MS to this point.
If $q.y > p.y$, add q to the Merged MS
3. Add every point in the Right MS to the Merged MS

$$p = G$$

Merged Maxima-Set



Maxima-Set

Base case a single point.

The MS of a single point is the point itself.



Maxima-Set: Analysis

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once.
Break ties in x by sorting by increasing y coordinate.

$O(n \log n)$

Divide sorted array into two halves.

$O(n)$

Recur recursively find the MS of each half.

$2T(n/2)$

Conquer compute the MS of the union of Left and Right MS

1. Find the highest point p in the Right MS
2. Compare every point q in the Left MS to this point.
If $q.y > p.y$, add q to the Merged MS
3. Add every point in the Right MS to the Merged MS

$O(n)$

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

Overall Running Time: pre-processing + $T(n) = O(n \log n)$

Maxima-Set: Correctness

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once.
Break ties in x by sorting by increasing y coordinate.

Divide sorted array into two halves.

Recur recursively find the MS of each half.

Conquer compute MS of union of Left/Right MS

1. Find the highest point p in the Right MS
2. Compare every point q in the Left MS to this point.
If $q.y > p.y$, add q to the Merged MS
3. Add every point in the Right MS to the Merged MS

Observations:

1. Every point in MS of the whole is in Left MS or Right MS
2. Every point in Right MS is in MS of the whole
3. Every point in Left MS is either in MS of the whole or is dominated by p

Integer multiplication

Given two n -digit integers x and y

Problem compute the product $x \cdot y$

While this seems like recreational mathematics, it does have real applications: Public key encryption is based on manipulating integers with thousands of bits.

Integer multiplication: Naïve approach

Given two n -digit integers x and y

Problem compute the product $x \cdot y$

Suppose we wanted to do it by hand. We assume that two digits can be multiplied or added in constant time

In primary school we all learn an algorithm for this problem that performs $\Theta(n^2)$ operations

Integer multiplication: Divide and conquer

Let $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$

Then $x \cdot y = x_1 \cdot y_1 \cdot 2^n + x_1 \cdot y_0 \cdot 2^{n/2} + x_0 \cdot y_1 \cdot 2^{n/2} + x_0 \cdot y_0$

We can compute the product of two n -digit numbers by making 4 recursive calls on $n/2$ -digit numbers and then combining the solutions to the subproblems.

Integer multiplication: Divide and conquer

```
def multiply(x, y):
    // x and y are positive integers represented in binary
    if x = 0 or y = 0 then return 0
    if x = 1 then return y
    if y = 1 then return x

    // recursive case
    let x1 and x0 be such that x = x1 2n/2 + x0
    let y1 and y0 be such that y = y1 2n/2 + y0

    return multiply(x1, y1) 2n +
           (multiply(x1, y0) + multiply(x0, y1)) 2n/2 +
           multiply(x0, y0)
```

Integer multiplication: Correctness

Let $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$

Then $x \cdot y = x_1 \cdot y_1 \cdot 2^n + x_1 \cdot y_0 \cdot 2^{n/2} + x_0 \cdot y_1 \cdot 2^{n/2} + x_0 \cdot y_0$

Straight forward application of induction to prove
that $\text{multiply}(x, y) = x \cdot y$

Integer multiplication: Complexity analysis

Recall $x \cdot y = x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0$

Divide step (produce halves) takes $O(n)$

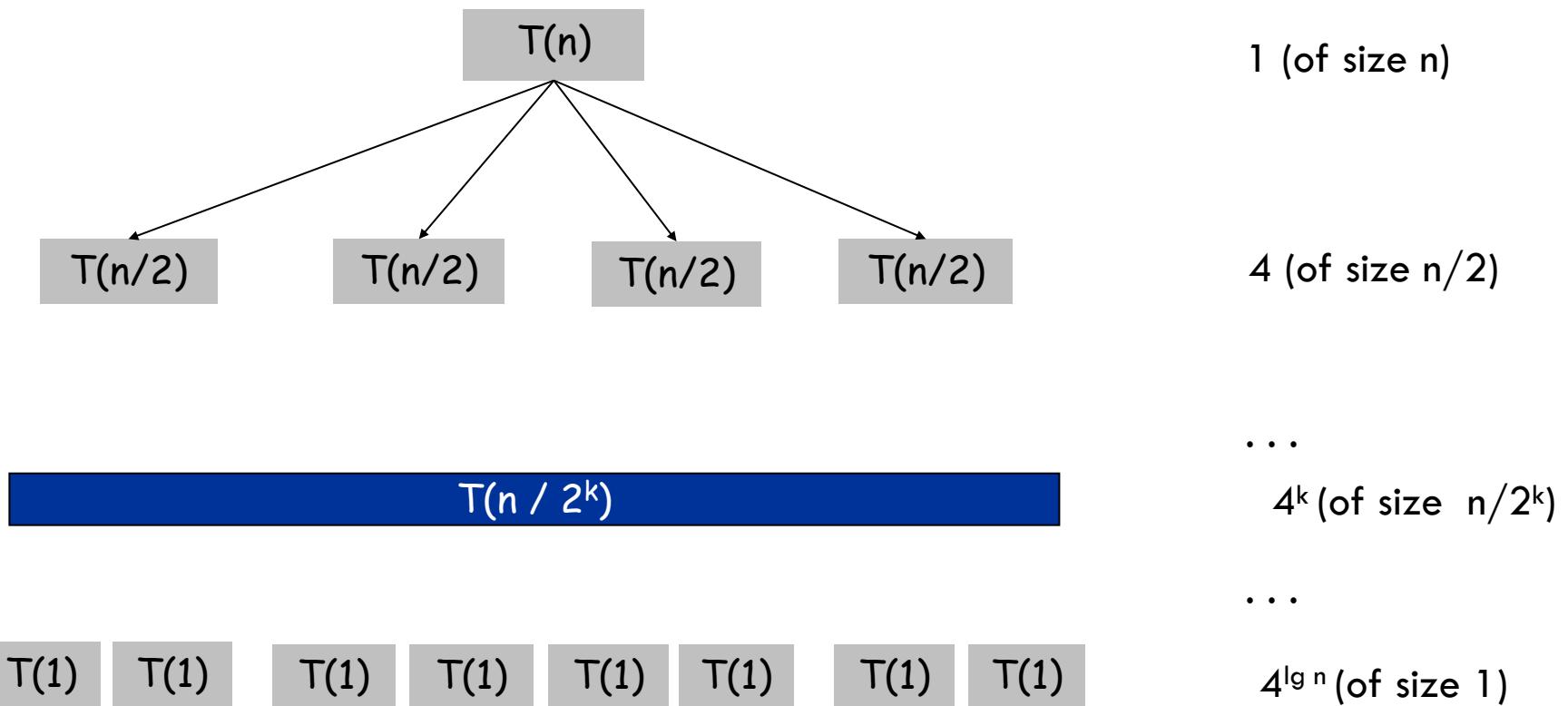
Recur step (solve subproblems) takes $4 T(n/2)$

Conquer step (add up results) takes $O(n)$

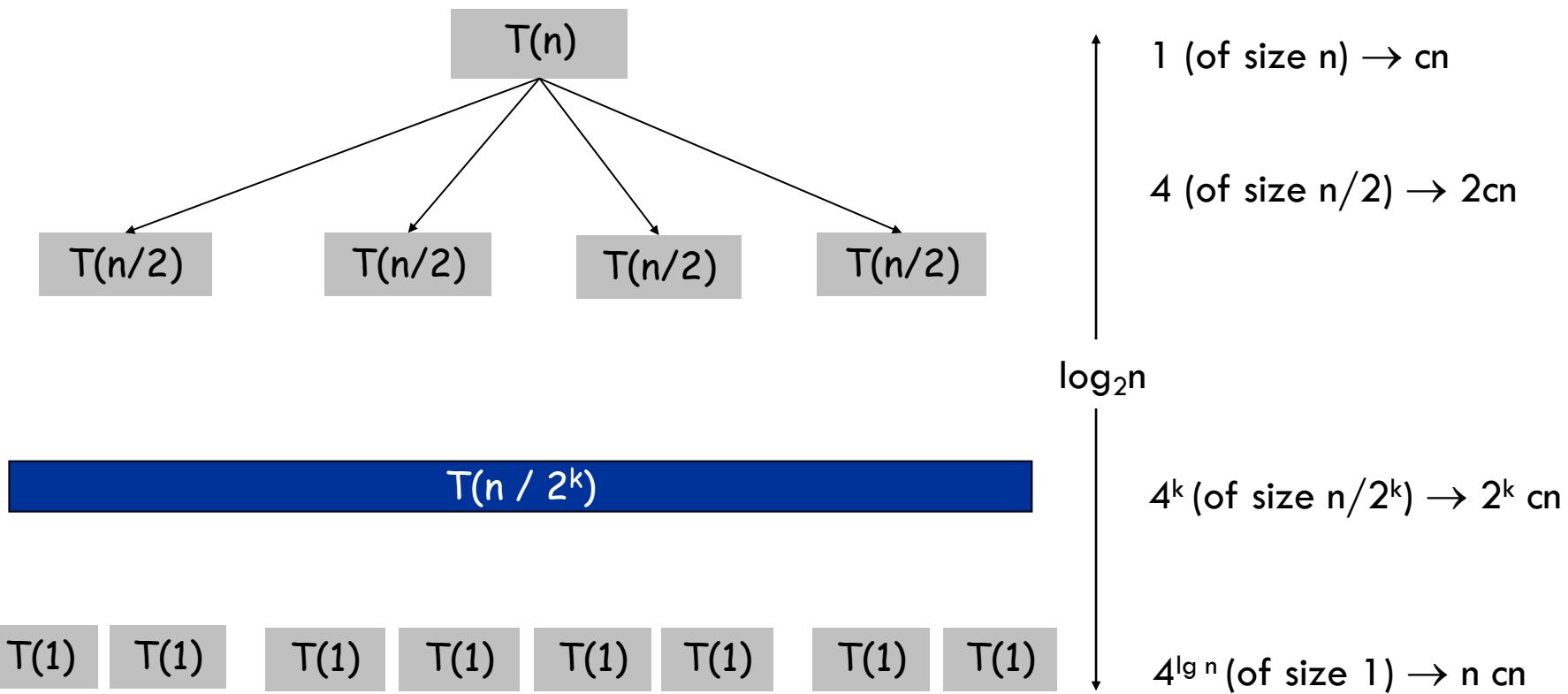
$$T(n) = \begin{cases} 4 T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n^2)$. No better than naïve!!!

Proof by unrolling



Proof by unrolling



Integer multiplication: Divide and conquer v2.0

Let $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$

$$\begin{aligned}x \cdot y &= x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0 \\(x_1 + x_0)(y_1 + y_0) &= x_1 \cdot y_1 + x_1 \cdot y_0 + x_0 \cdot y_1 + x_0 \cdot y_0\end{aligned}$$

We can compute the product of two n -digit numbers by making 3 recursive calls on $n/2$ -digit numbers and then combining the solutions to the subproblems.

Integer multiplication: Divide and conquer

```
def multiply(x, y):
    // base case
    :
    // recursive case
    let  $x_1$  and  $x_0$  be such that  $x = x_1 \cdot 2^{n/2} + x_0$ 
    let  $y_1$  and  $y_0$  be such that  $y = y_1 \cdot 2^{n/2} + y_0$ 

    first_term ← multiply( $x_1$ ,  $y_1$ )
    last_term ← multiply( $x_0$ ,  $y_0$ )
    other_term ← multiply( $x_1 + x_0$ ,  $y_1 + y_0$ )

    return first_term  $\cdot 2^n$  +
                    (other_term - first_term - last_term)  $\cdot 2^{n/2}$  +
                    last_term
```

Integer multiplication: Complexity analysis

Divide step (produce halves) takes $O(n)$

Recur step (solve subproblems) takes $3 T(n/2)$

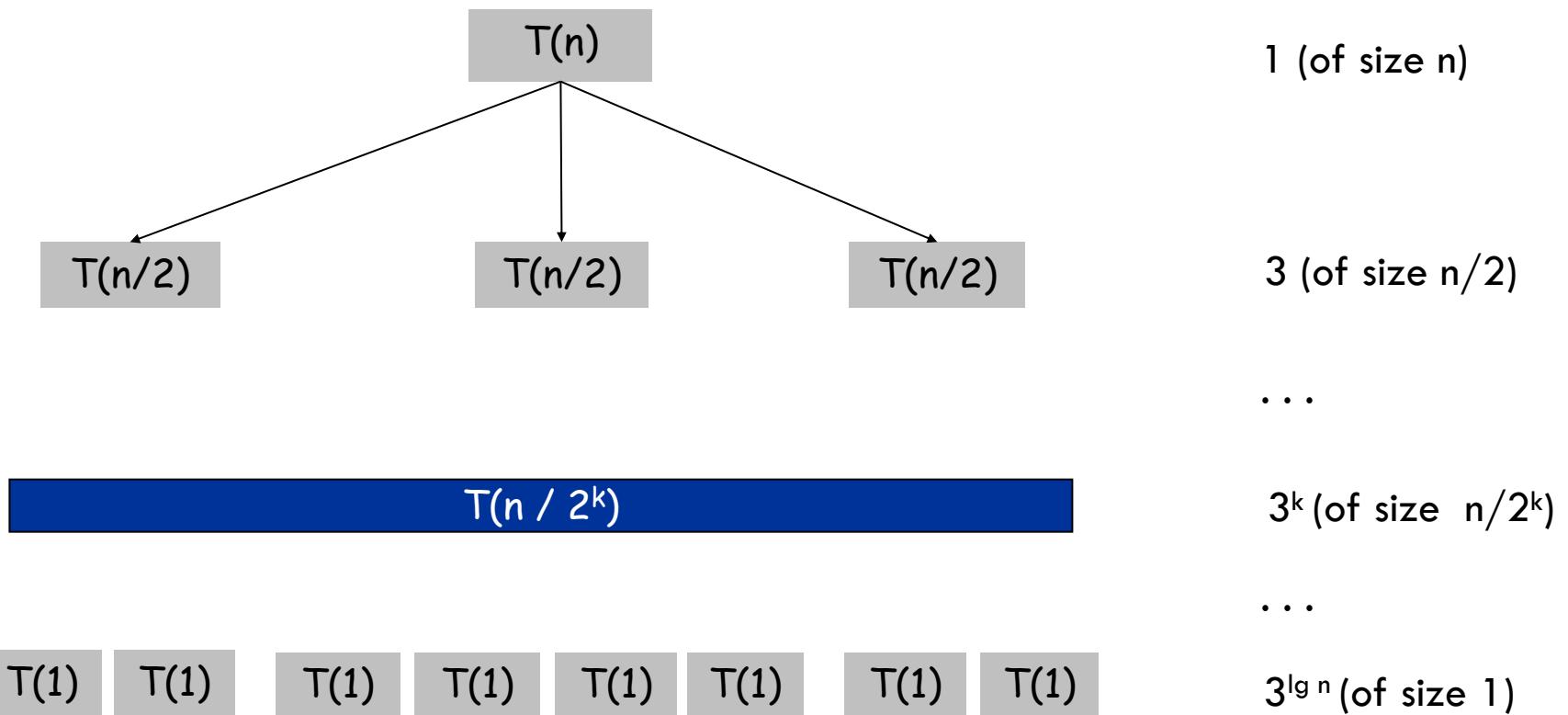
Conquer step (add up results) takes $O(n)$

$$T(n) = \begin{cases} 3 T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

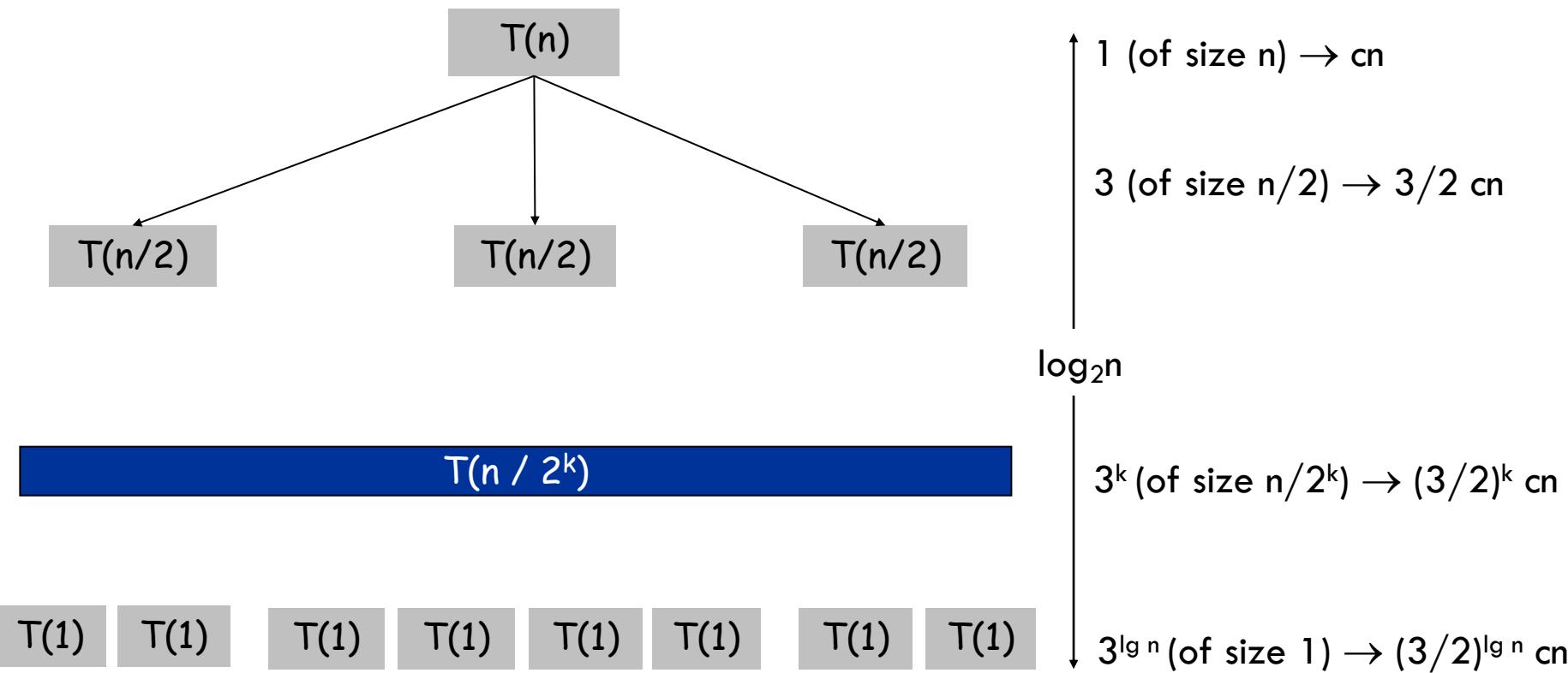
This solves to $T(n) = O(n^{\log_2 3})$, where $\log_2 3 \approx 1.6$

Better than naïve!!!

Proof by unrolling



Proof by unrolling



Geometric series facts

Let r be a positive real and k a positive integer then

$$1 + r + r^2 + \dots + r^k = (r^{k+1} - 1)/(r-1)$$

Consequently if $r > 1$ then

$$1 + r + r^2 + \dots + r^k < r^{k+1} / (r-1)$$

and if $r < 1$ then

$$1 + r + r^2 + \dots + r^k < 1 / (1-r)$$

Logarithms facts

Base exchange rule:

$$\log_a x = (\log_b x) / (\log_b a)$$

Product rule:

$$\log_a (xy) = (\log_a x) + (\log_a y)$$

Power rule:

$$\log_a x^b = b \log_a x$$

Master Theorem

Let $f(n)$ and $T(n)$ be defined as follows:

$$T(n) = \begin{cases} a T(n/b) + f(n) & \text{for } n \geq d \\ c & \text{for } n < d \end{cases}$$

Depending on a , b and $f(n)$ the recurrence solves to:

1. if $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$,
2. if $f(n) = \Theta(n^{\log_b a} \log^k n)$ for $k \geq 0$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$,
3. if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $a f(n/b) \leq \delta f(n)$ for $\varepsilon > 0$ and $\delta < 1$ then $T(n) = \Theta(f(n))$,

Note: If $f(n)$ is given as big-O, you can only conclude $T(n)$ as big-O (not Θ).

Note: You should be able to solve all recurrences in this class using unrolling, but if you are comfortable using the Master Theorem, go for it.

The Master Theorem

Examples

1. $T(n) = 8T(n/2) + n^2$

$a=8, b=2, f(n)=n^2, \log_b(a) \rightarrow \log_2(8) = 3; \text{ so } f(n) = O(n^{\log_b a - \varepsilon})$ (case 1)

$$T(n) \in \Theta(n^3)$$

2. $T(n) = 2T(n/2) + O(n)$

$a=2, b=2, f(n)=n, \log_b(a) \rightarrow \log_2(2) = 1; \text{ so } f(n) = \Theta(n^{\log_b a} \log^k n)$ (case 2 with $k=0$)

$$T(n) \in O(n \log(n))$$

3. $T(n) = 2T(n/2) + O(n^2)$

$a=2, b=2, f(n)=n^2, \log_b(a) \rightarrow \log_2(2) = 1; \text{ so } f(n) = \Omega(n^{\log_b a + \varepsilon})$ (case 3)

$$T(n) \in O(n^2)$$

Selection

Given an unsorted array A holding n numbers and an integer k ,
find the k th smallest number in A

Trivial solution: Sort the elements and return k th element

Can we do better than $O(n \log n)$?

Yes, with divide and conquer!

First attempt

1. **Divide** find the median ($\lfloor n/2 \rfloor$ th element for simplicity) and split array on the halves, \leq and $>$ than the median
2. **Recur** if $k \leq \lfloor n/2 \rfloor$ find k th element on smaller half
if $k > \lfloor n/2 \rfloor$ find $(k - \lfloor n/2 \rfloor)$ th element on larger half
3. **Conquer** return value of the recursive call



$k = 6$
 $n = 10$



Divide



Recur

$k = 1$

13

Conquer

Selection time complexity

Divide step (find median and split) takes at least $O(n)$

Recur step (solve left or right subproblem) takes $T(n/2)$

Conquer step (return recursive result) takes $O(1)$

If we could compute the median in $O(n)$ time then:

$$T(n) = \begin{cases} T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n)$ but only if we can solve the median problem, which is in fact a special case of selection with $k=[n/2]$

Second attempt: Approximating the median

We don't need the exact median. Suppose we could find in $O(n)$ time an element x in A such that

$$|A| / 3 \leq \text{rank}(A, x) \leq 2 |A| / 3$$

Then we get the recurrence

$$T(n) = \begin{cases} T(2n/3) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

Which again solves to $T(n) = O(n)$

To approximate the median we can use a recursive call!

Median of 3 medians

Consider the following procedure

- Partition A into $|A| / 3$ groups of 3
- For each group find the median (bruteforce)
- Let x be the median of the medians (computed recursively)

We claim that x has the desired property

$$|A| / 3 \leq \text{rank}(A, x) \leq 2|A| / 3$$

Half of the groups have a median that is smaller/larger than x , and each group has two elements smaller/larger than x , thus

$$\begin{aligned}\# \text{ elements smaller than } x &> 2(|A| / 6) = |A| / 3 \\ \# \text{ elements greater than } x &> 2(|A| / 6) = |A| / 3\end{aligned}$$

Median of 3 medians

Let x be the median of the medians, then

$$|A| / 3 \leq \text{rank}(A, x) \leq 2|A| / 3$$

1	12	5	16	19	7	23	6	13
---	----	---	----	----	---	----	---	----

1	12	5
---	----	---

16	19	7
----	----	---

23	6	13
----	---	----

1	5	12
---	---	----

7	16	19
---	----	----

6	13	23
---	----	----

1	5	12
---	---	----

6	13	23
---	----	----

7	16	19
---	----	----

elements smaller than $x > 2(|A| / 6) = |A| / 3$
elements greater than $x > 2(|A| / 6) = |A| / 3$

Median of 3 median time complexity

We don't need the exact median. With a recursive call on $n/3$ elements, we can find x in A such that

$$|A|/3 < \text{rank}(A, x) < 2|A|/3$$

Then we get the recurrence

$$T(n) = T(2n/3) + T(n/3) + O(n)$$

Which solves to $T(n) = O(n \log n)$

No better than sorting!

Median of 5 medians

We don't need the exact median. With a recursive call on $n/5$ elements, we can find x in A such that

$$3|A|/10 < \text{rank}(A, x) < 7|A|/10$$

Then we get the recurrence

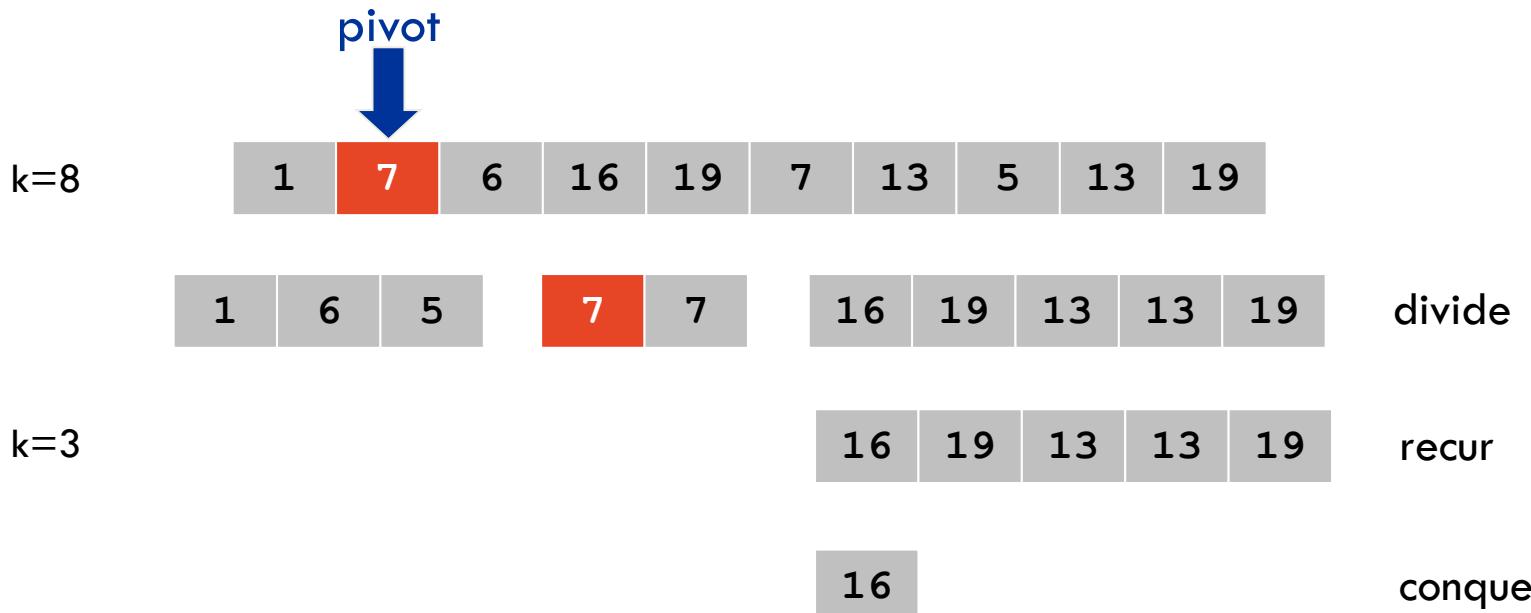
$$T(n) = T(7n/10) + T(n/5) + O(n)$$

Which solves to $T(n) = O(n)$

Asymptotically faster than sorting!

Quick selection

1. **Divide** Choose a random element from the list as the **pivot**
Partition the elements into 3 lists:
(i) less than, (ii) equal to and (iii) greater than the **pivot**
2. **Recur** Recursively select right element from correct list
3. **Conquer** Return solution to recursive problem



Quick selection complexity analysis

Divide step (pick pivot and split) takes $O(n)$

Recur step (solve left and right subproblem) takes $T(n')$

Conquer step (return solution) takes $O(1)$

Now we can set up the recurrence for $T(n)$:

$$E[T(n)] = \begin{cases} E[T(n')] + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $E[T(n)] = O(n)$

(details available on the textbook but not examinable)

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

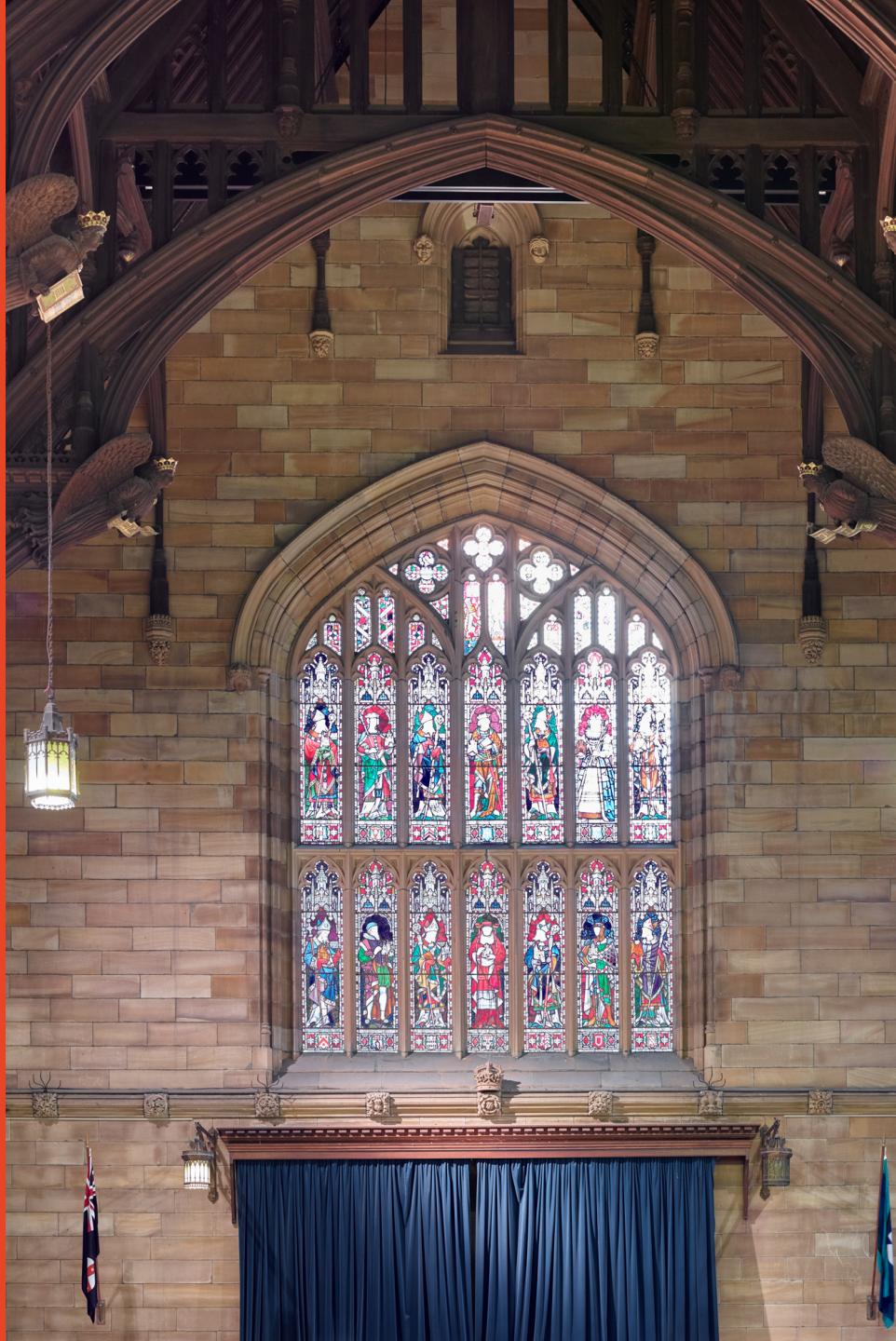
COMP2123

Data structures and Algorithms

Lecture 12: Randomized Algorithms
[GT 19.1 and 19.6]

Dr. André van Renssen
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



Randomized algorithms

Randomized algorithms are algorithms where the behaviour doesn't depend solely on the input. It also depends (in part) on random choices or the values of a number of random bits.

Reasons for using randomization:

- Sampling data from a large population or dataset
- Avoid pathological worst-case examples
- Avoid predictable outcomes
- Allow for simpler algorithms

Randomized algorithms

Randomized algorithms are algorithms where the behaviour doesn't depend solely on the input. It also depends (in part) on random choices or the values of a number of random bits.

Today:

- Generating random permutations
- Skip lists

Generating random permutations

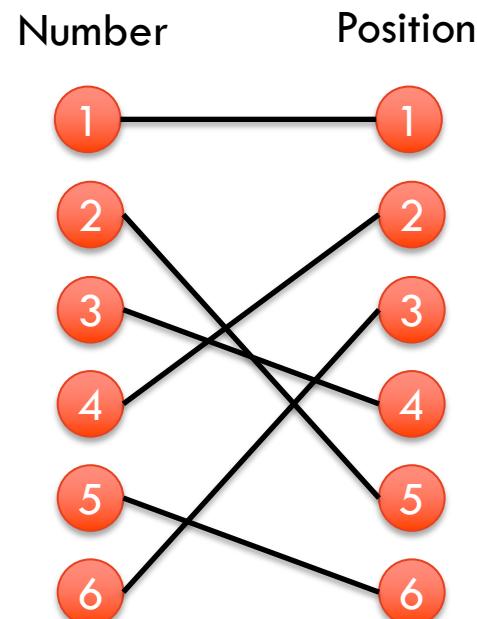
Input: An integer n .

Output: A permutation of $\{1, \dots, n\}$ chosen uniformly at random, i.e., every permutation has the same probability of being generated.

Example:

$n = 6$

$<1,4,6,3,2,5>$



Generating random permutations

What are random permutations used for?

- Many algorithms whose input is an array perform better in practice after randomly permuting the input (for example, QuickSort).
- Can be used to sample k elements without knowing k in advance by picking the next element in the permuted order when needed.
- Can be used to assign scarce resources.
- Can be a building block for more complex randomized algorithms.

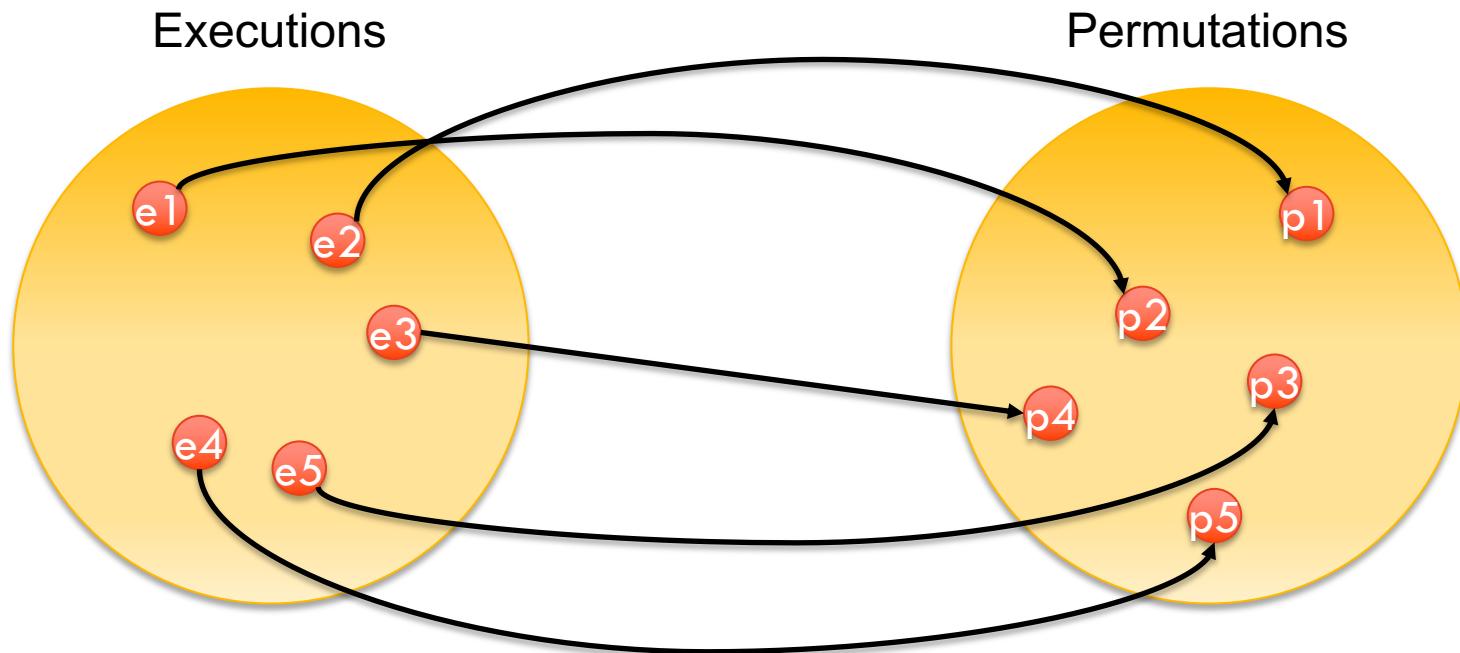
First (incorrect) attempt

```
def permute(A):  
  
    # permute A in place  
    n ← length of array A  
  
    for i in {0, ..., n-1} do  
        # swap A[i] with random position  
        j ← random number in {0, ..., n-1}  
        A[i], A[j] ← A[j], A[i]  
  
    return A
```

Note that since j is picked at random, different executions lead to different outcomes

So, why is this incorrect?

For all permutations to be equally likely, we want that every permutation is generated by the same number of possible executions.



First (incorrect) attempt: Analysis

Number of executions:

$$n * n * n * \dots * n = n^n$$


n times

Number of permutations:

$$1 * 2 * 3 * \dots * n = n!$$

n^n isn't divisible by $n!$

Example:

$$n = 3$$

$$n^n = 27$$

$$n! = 6$$

27 isn't a multiple of 6, so some permutations are more likely than others.

```
def permute(A):  
    # permute A in place  
    n ← length of array A  
  
    for i in {0, ..., n-1} do  
        # swap A[i] with random position  
        j ← random number in {0, ..., n-1}  
        A[i], A[j] ← A[j], A[i]  
  
    return A
```

Second attempt

```
def FisherYates(A):  
  
    # permute A in place  
    n ← length of array A  
  
    for i in {0, ..., n-1} do  
        # swap A[i] with random position  
        j ← random number in {i, ..., n-1}  
        A[i], A[j] ← A[j], A[i]  
  
    return A
```

Note that since j is picked at random, different executions lead to different outcomes

Second attempt: Analysis

```
def FisherYates(A):
```

Number of executions:

$$1 * 2 * 3 * \dots * n = n!$$

Number of permutations:

$$1 * 2 * 3 * \dots * n = n!$$

Observation: Every execution leads to a different permutation.

```
# permute A in place  
n ← length of array A  
  
for i in {0, ..., n-1} do  
    # swap A[i] with random position  
    j ← random number in {i, ..., n-1}  
    A[i], A[j] ← A[j], A[i]  
  
return A
```

Example: To generate $<3,2,4,1>$ starting from $<1,2,3,4>$

$<1,2,3,4> \rightarrow <3,2,1,4>$, $i=0$ and $j=2$

$<3,2,1,4> \rightarrow <3,2,1,4>$, $i=1$ and $j=1$

$<3,2,1,4> \rightarrow <3,2,4,1>$, $i=2$ and $j=3$

$<3,2,4,1> \rightarrow <3,2,4,1>$, $i=3$ and $j=3$

Second attempt: Analysis

Theorem:

The Fisher-Yates algorithm generates a permutation uniformly at random.

Proof:

- Every execution of the algorithm happens with probability $1/n!$.
- Each execution generates a different permutation.
- Hence, the probability that a specific permutation is generated is $1/n!$, for all possible permutations of $\langle 1, 2, \dots, n \rangle$.

Skip lists

Another way to implement a Map.

So, why are we looking at another different way of doing this?

- Relatively simple data structure that's built in a randomized way
- No need for rebalancing like in AVL trees
- Still has $O(\log n)$ expected worst-case time (this is NOT average case)

Applications:

- Various database systems use it
- Concurrent/parallel computing environments

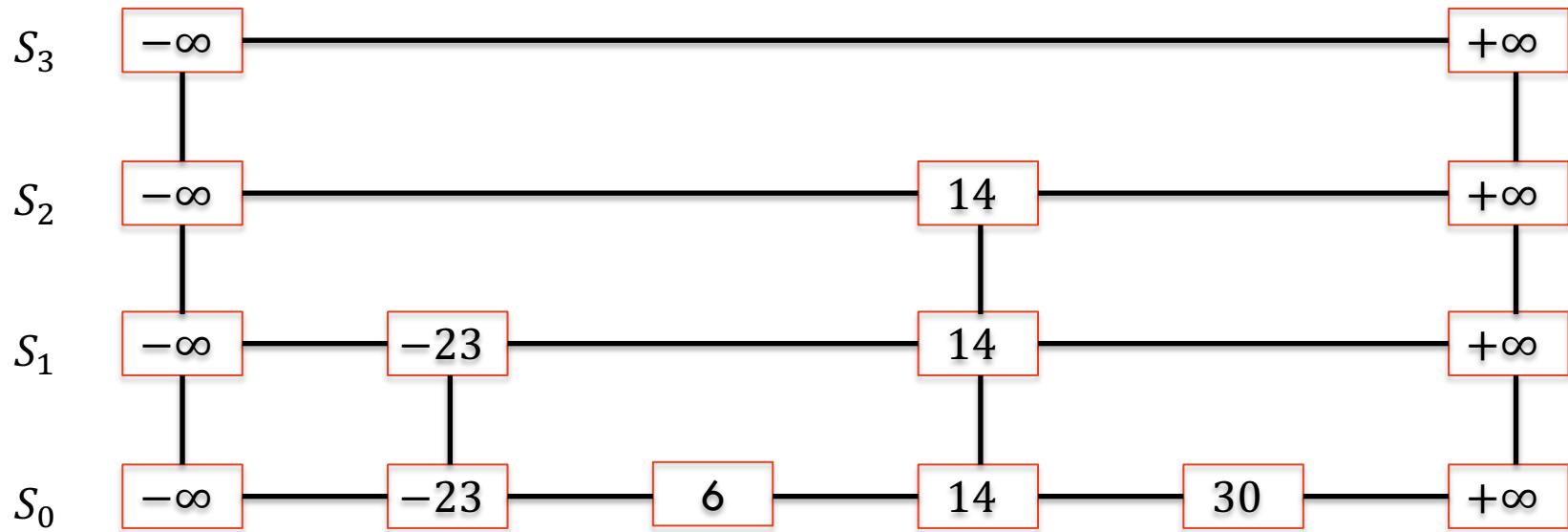
The Map ADT (recap)

- **get(k):** if the map M has an entry with key k , return its associated value
- **put(k, v):** if key k is not in M , then insert (k, v) into the map M ; else, replace the existing value associated to k with v
- **remove(k):** if the map M has an entry with key k , remove it
- **size(), isEmpty()**
- **entrySet():** return an iterable collection of the entries in M
- **keySet():** return an iterable collection of the keys in M
- **values():** return an iterable collection of the values in M

Skip lists

Leveled structure, where every level is a subset of the one below it.

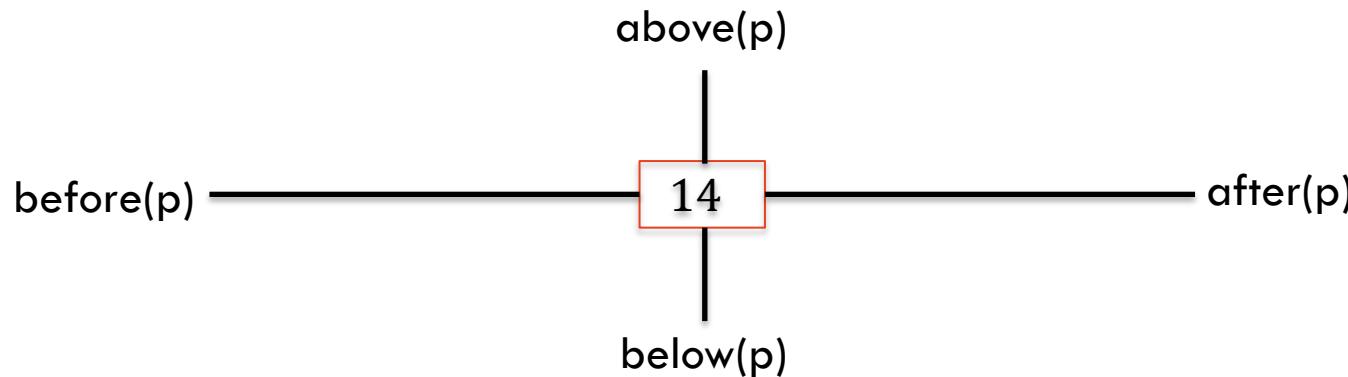
Next level's elements determined by coin flips.



Skip lists

A node p has pointer to:

- $\text{after}(p)$: Node following p on same level.
- $\text{before}(p)$: Node preceding p in the same level.
- $\text{above}(p)$: Node above p in the same tower.
- $\text{below}(p)$: Node below p in the same tower.



Search

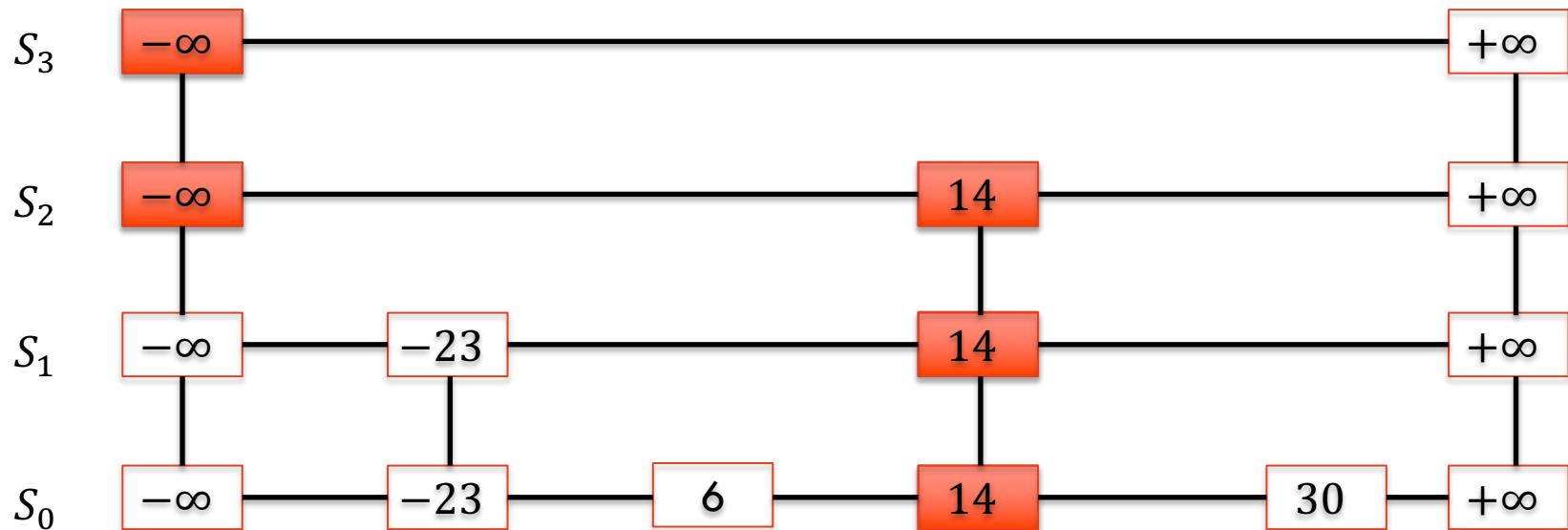
```
def search(p, k):
    while below(p) ≠ null do
        p ← below(p)
        while key(after(p)) ≤ k do
            p ← after(p)
    return p
```



Example: `search(topleft node, 30)`

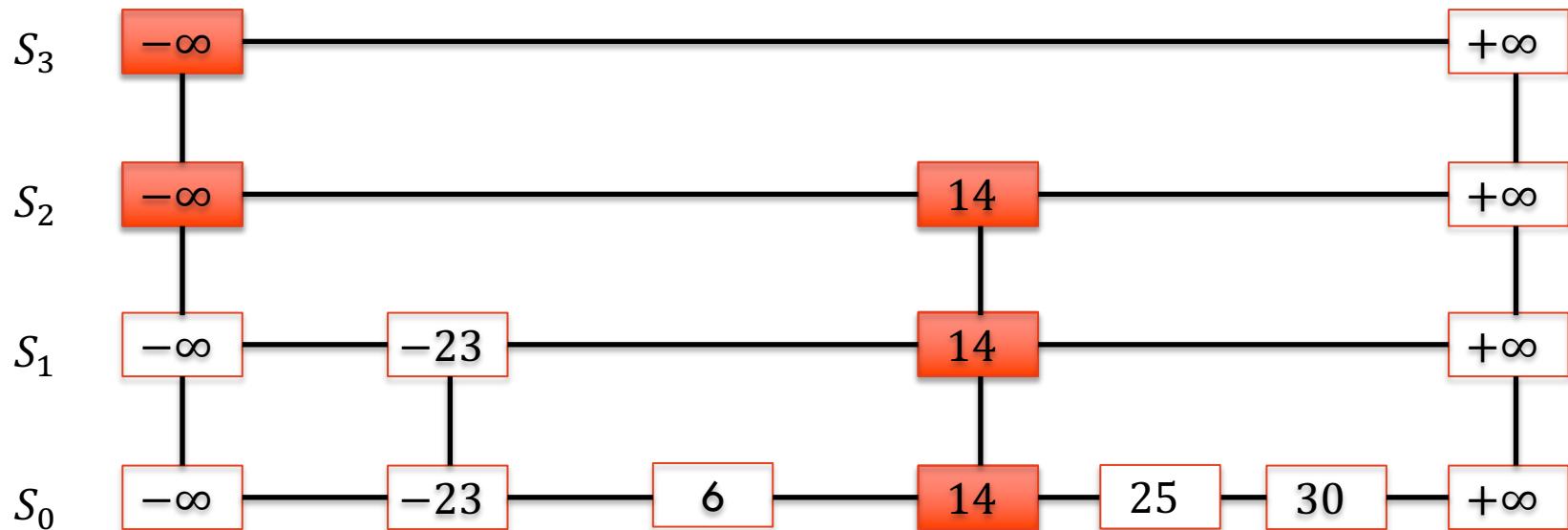
Insertion

```
def insert(p,k):
    p ← search(p,k)
    q ← insertAfterAbove(p,null,k)
    while coin flip is heads do
        while above(p) = null do
            p ← before(p)
        p ← above(p)
    q ← insertAfterAbove(p,q,k)
```



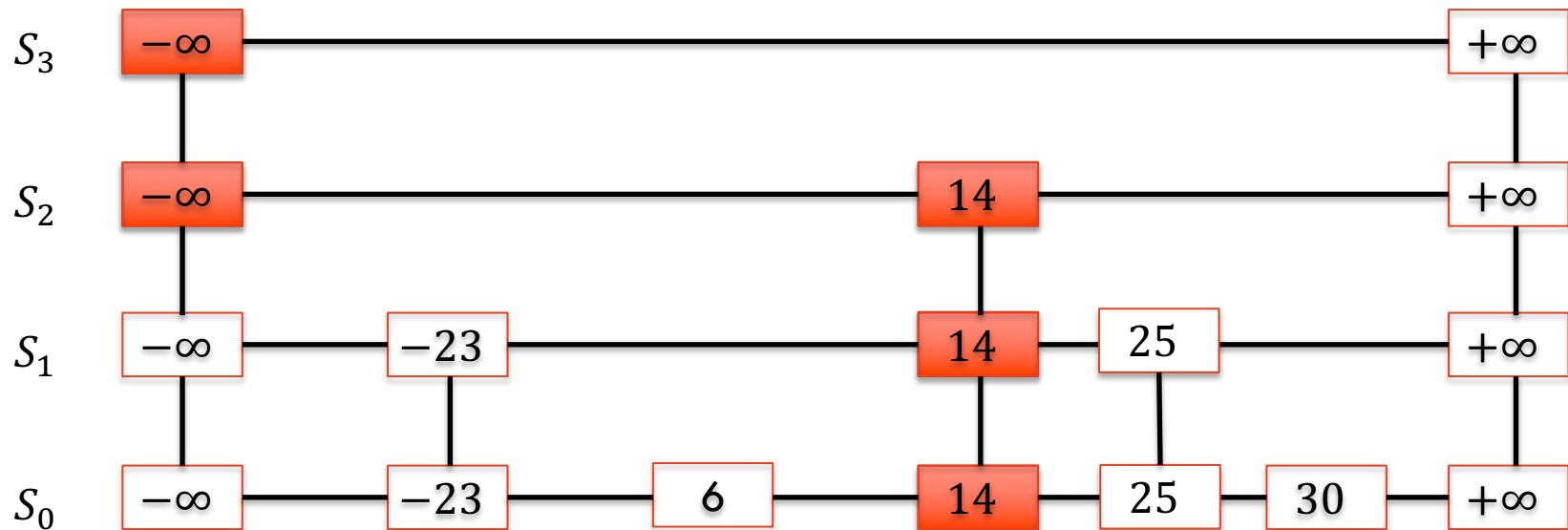
Insertion

```
def insert(p,k):
    p ← search(p,k)
    q ← insertAfterAbove(p,null,k)
    while coin flip is heads do
        while above(p) = null do
            p ← before(p)
        p ← above(p)
    q ← insertAfterAbove(p,q,k)
```



Insertion

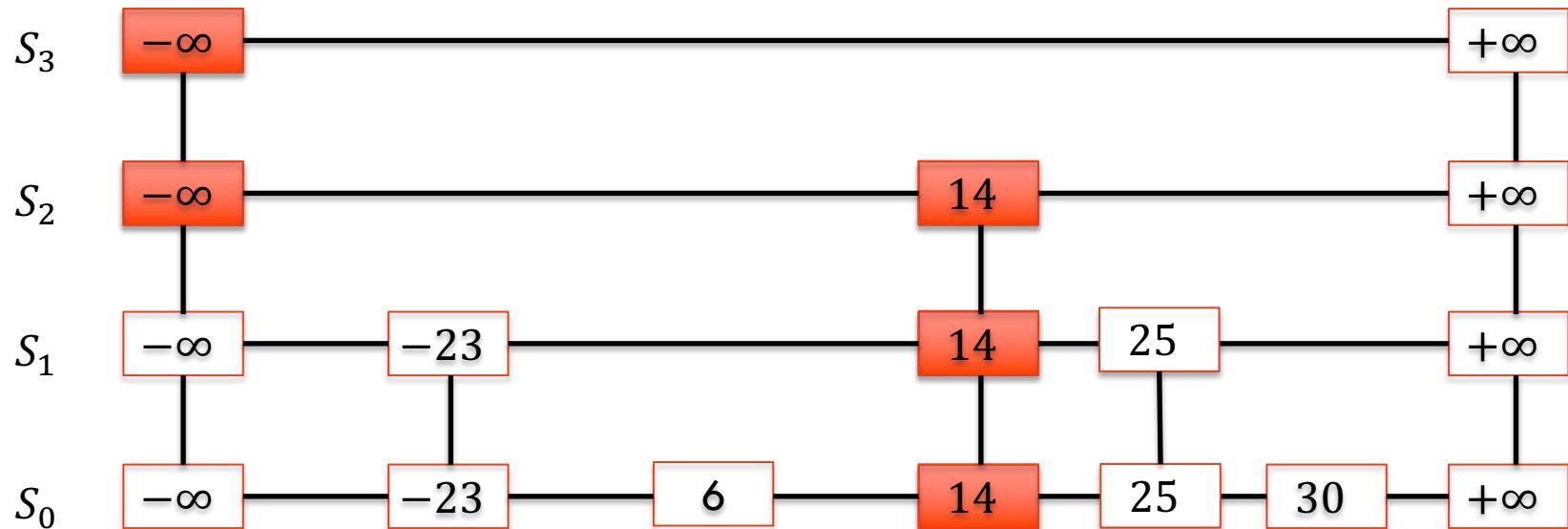
```
def insert(p,k):
    p ← search(p,k)
    q ← insertAfterAbove(p,null,k)
    while coin flip is heads do
        while above(p) = null do
            p ← before(p)
        p ← above(p)
    q ← insertAfterAbove(p,q,k)
```



Example: `insert(topleft node, 25)`

Removal

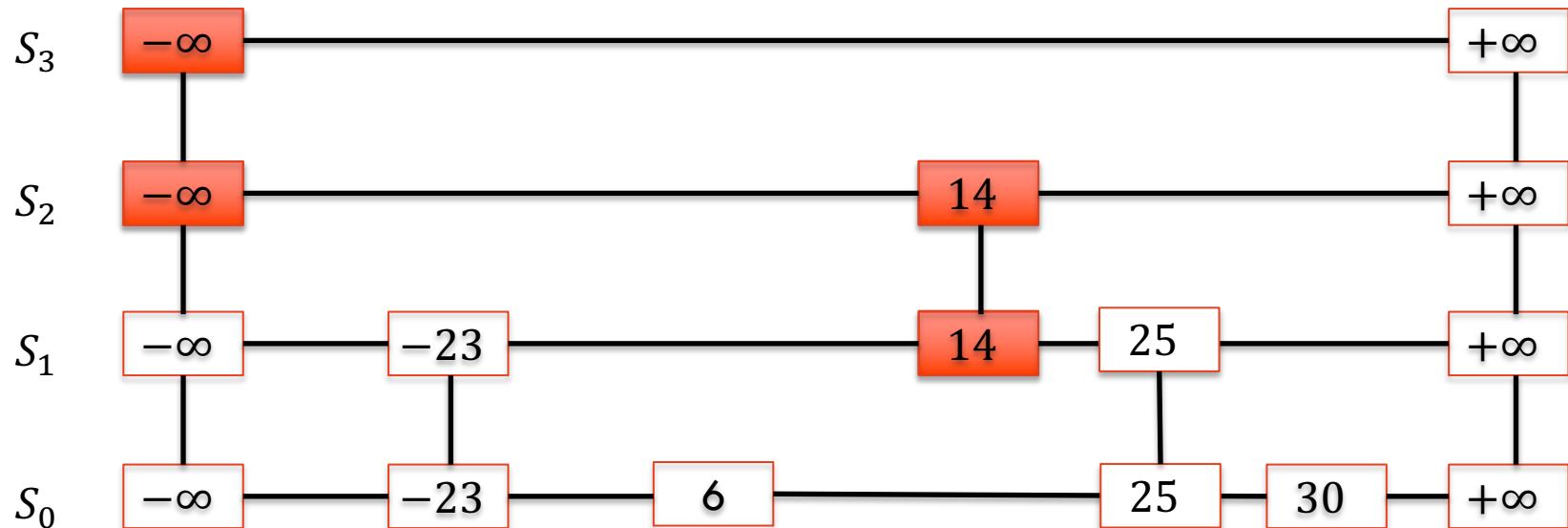
```
def remove(p, k):
    p ← search(p, k)
    if key(p) ≠ k then return null
    repeat
        remove p
        p ← above(p)
    until above(p) = null
```



Example: `remove(topleft node, 14)`

Removal

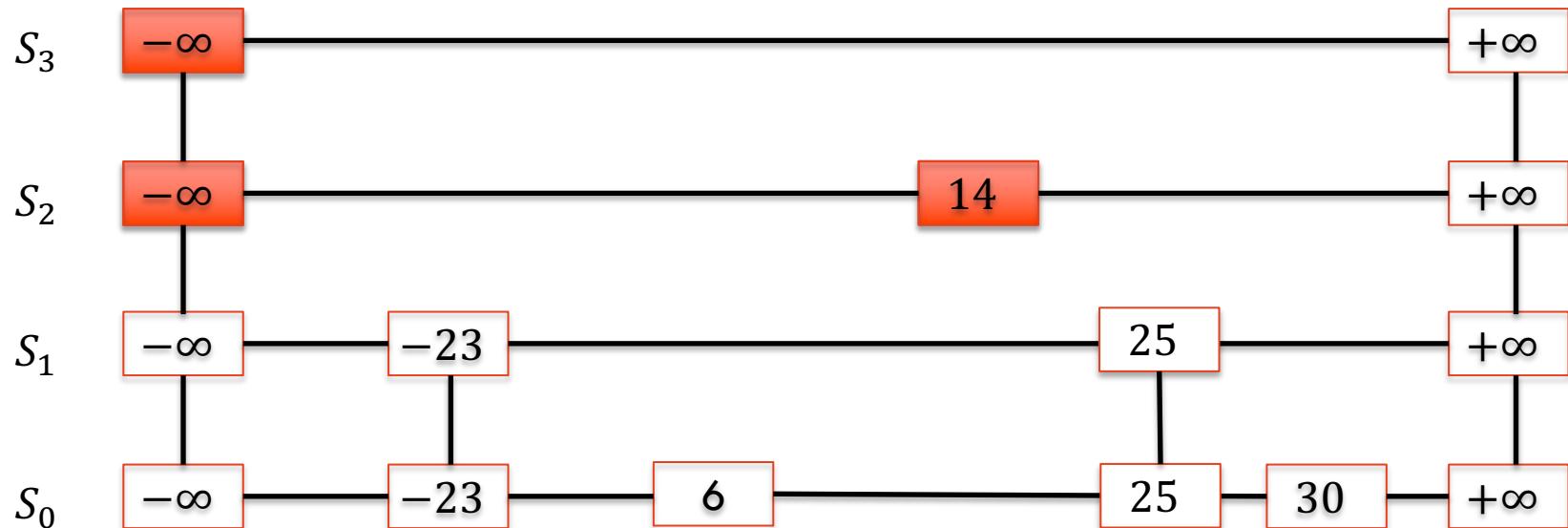
```
def remove(p, k):
    p ← search(p, k)
    if key(p) ≠ k then return null
    repeat
        remove p
        p ← above(p)
    until above(p) = null
```



Example: `remove(topleft node, 14)`

Removal

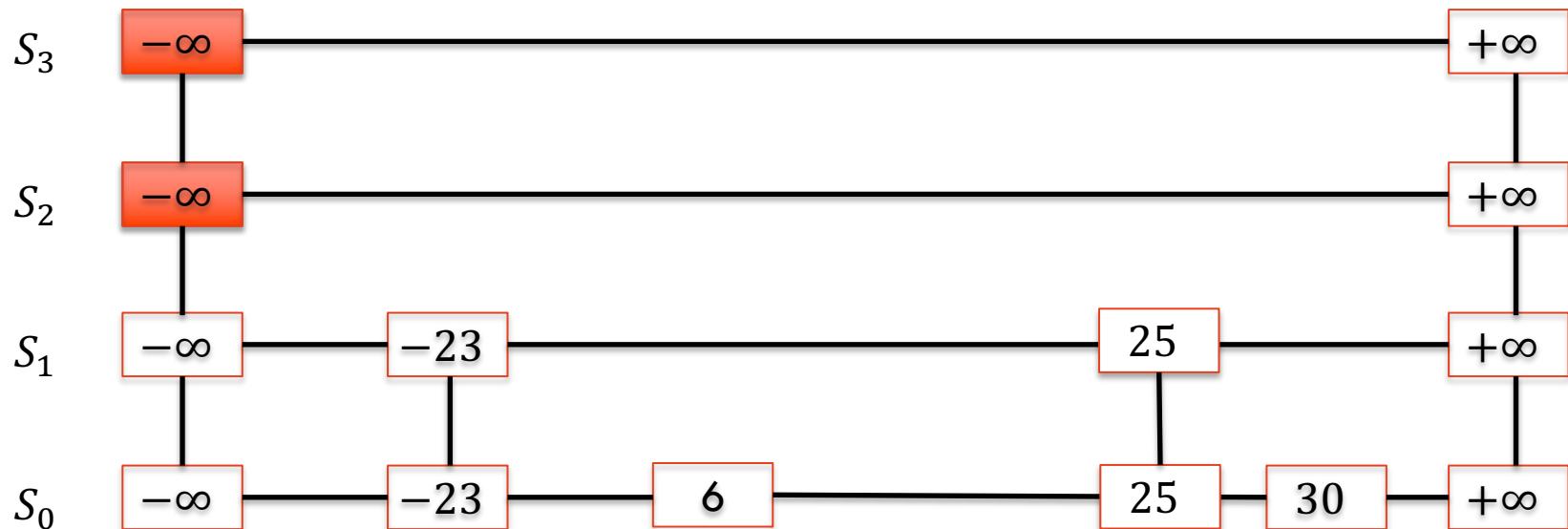
```
def remove(p, k):
    p ← search(p, k)
    if key(p) ≠ k then return null
    repeat
        remove p
        p ← above(p)
    until above(p) = null
```



Example: `remove(topleft node, 14)`

Removal

```
def remove(p, k):
    p ← search(p, k)
    if key(p) ≠ k then return null
    repeat
        remove p
        p ← above(p)
    until above(p) = null
```



Example: `remove(topleft node, 14)`

Skip lists: Top layer

Keep a pointer to the topleft node.

Choices for the top layer:

- Keep at a fixed level, say $\max\{10, 3[\log(n)]\}$
 - Insertion needs to take this into account
- Variable level
 - Continue insertion until coin comes up tails
 - No modification required
 - Probability that this gives more than $O(\log n)$ levels is very low

Skip lists: Analysis

Theorem:

The expected height of a skip list is $O(\log n)$.

Proof:

- The probability that an element is present at height i is $1/2^i$.
 - I.e., the probability that the coin comes up heads i times.
- The probability that level i has at least one item is at most $n/2^i$.
- The probability that skip list has height h is probability that level h has at least one element.
- So, probability that skip list has height larger than $c \log n$ is at most

$$\frac{n}{2^{c \log n}} = \frac{n}{n^c} = \frac{1}{n^{c-1}}$$

- So, probability that skip list has height $O(\log n)$ is at least

$$1 - \frac{1}{n^{c-1}}$$

Skip lists: Search Analysis

Theorem:

The expected search time of a skip list is $O(\log n)$.

Proof:

- Searching consists of horizontal and vertical steps.
- There are h vertical steps, so $O(\log n)$ with high probability.
- To have a horizontal step on level i , the next node can't be on level $i+1$.
- The probability of this is $1/2$.
- This means that the expected number of horizontal steps per level is 2.
- So we expect to spend $O(1)$ time per level.
- Expected search time: $O(\log n)$ time with high probability.

Insertion and deletion take expected $O(\log n)$ time using similar analysis.

Skip lists: Space Analysis

Theorem:

The expected space used by a skip list is $O(n)$.

Proof:

- Space per node: $O(1)$
- Expected number of nodes at level i is $n/2^i$.
- Thus expected number of nodes is

$$\sum_{i=0}^h \frac{n}{2^i} = n \sum_{i=0}^h \frac{1}{2^i} < 2n$$

Skip lists: Summary

Expected space: $O(n)$

Expected search/insert/delete time: $O(\log n)$

Works very well in practice and doesn't require any complicated rebalancing operations.

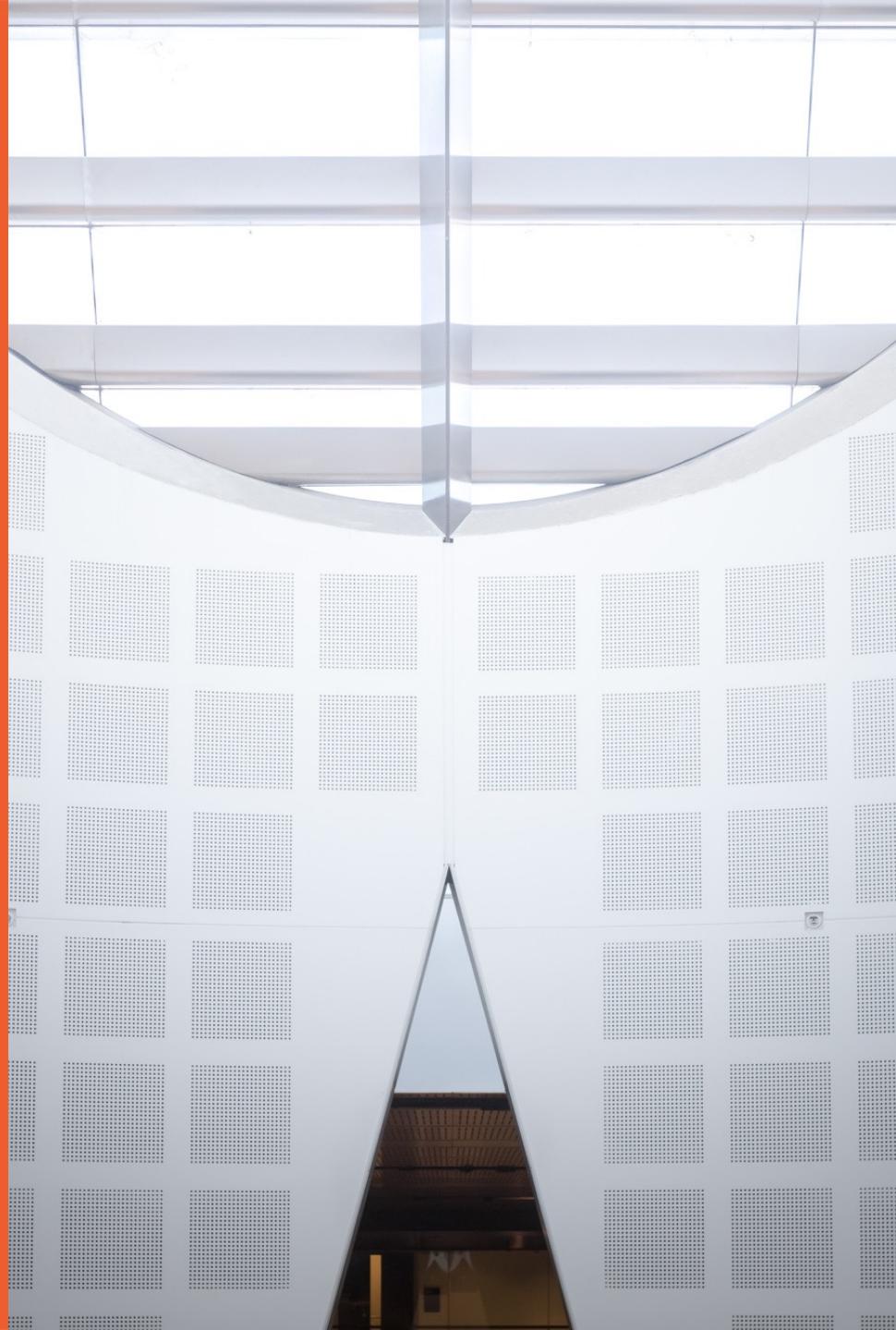
COMP2123

Week 13: Recap and Exam Review

Dr. André van Renssen
School of Computer Science



THE UNIVERSITY OF
SYDNEY



Quick announcements

Fill out online Unit of Study Survey

- <https://student-surveys.sydney.edu.au/students/>
- Use the free text to help us make this better for next years students. “Pay it forward”

Examples of changes based on last year's feedback:

- wrote the guide on how to approach algorithmic problems
- designed programming exercises

Week 13 Quiz

Quiz 10 will be about the final. Unlike previous quizzes you will be able to attempt it multiple times.

It's available until the start of the exam.

Looking back

Lecture 1a - Introduction

Lecture 1b - Analysis

Lecture 2 - Lists

Lecture 3 - Trees

Lecture 4 - Binary Search Trees

Lecture 5 - Priority Queues

Lecture 6 - Hashing

Lecture 7 - Graphs

Lecture 8 - Graph Algorithms

Lecture 9 - Greedy Algorithms

Lecture 10 - Divide and Conquer I

Lecture 11 - Divide and Conquer II

We covered a
lot of ground!

Core concept 1: Abstraction layers

Abstract Data Type



Data Structure



Computer code

Problem definition



Algorithm



Computer code

Core concept 2: Algorithm analysis

A principled framework for evaluating algorithms:

- measuring performance of resource use
- proving correctness

These should inform your design and implementation choices

Learning outcomes

1. Proficiency in organising, presenting and discussing professional ideas [...]
2. Using mathematical methods to evaluate the performance of an algorithm.
3. Using notation of big-O to represent asymptotic growth of cost functions.
4. Understanding of commonly used data structures, including lists, stacks, queues, priority queues, search trees, hash tables, and graphs. This covers the way information is represented in each structure, algorithms for manipulating the structure, and analysis of asymptotic complexity of the operations.
5. Understanding of basic algorithms related to data structures, such as algorithms for sorting, tree traversals, and graph traversals.
6. Ability to write code that recursively performs an operation on a data structure.
7. Experience designing an algorithmic solution to a problem, coding it, and analysing its complexity.
8. Ability to apply basic algorithmic techniques (e.g. divide-and-conquer, greedy) to given design tasks.

Beyond this unit of study

SCS offers many algorithmic units:

- COMP2022 Models of Computation (S2)
- COMP3027 Algorithm Design (S1)
- COMP3530 Discrete Optimization (S2)
- COMP5045 Computational Geometry (S1)

Sydney Algorithms and Computation Theory group:

- weekly seminar on Algorithms research
- do a research project with us
- we are always looking for motivated honours students

What is examinable?

Everything from the lectures, the referenced sections of the textbooks, the tutorials, the quizzes, the assignments. Exceptions to this rule:

- when explicitly labeled as non-examinable.
- probabilistic analysis of randomized algorithms

In general though, if it happened during this unit, you are expected to know about it!

Focus on the things we put most emphasis on, as seen in tutorials and assignments

Final Exam Structure

2 hours writing plus 10 minutes reading

4 questions worth in total 60 points

Worth 60% of overall COMP2123 grade

Final exam has a 40% barrier

Do's and Don'ts

Open book exam:

- Can refer to slides, tutorial solutions, assignment solutions, books used in the unit
 - Making a 2-page summary is highly recommended
- Can't use the internet to look things up
- **Never copy text verbatim from anywhere, including the slides (few points and potential academic dishonesty)**
 - If you refer to anything from the permitted material, write in your own words

Type your answers and submit it as an assignment in **Canvas**

Handwritten/scanned answers will **not** be accepted.

Start your submission with your student ID

- Don't include your name

Problem 1

10 points

Analysis of given algorithms

Easy problem. Make sure you nail it!

Problem 2

10 points

Tracing algorithms/data structures on a given example

Easy problem. Make sure you nail it!

Problem 3

20 points

Design or modify an ADT

Medium/Hard problem.

Remember to:

- Describe your approach
- Prove correctness
- Analyze complexity (if there's a space requirement, don't forget to analyze this as well)

Problem 4

20 points

Design an algorithm that solves a problem

Medium/Hard problem.

Remember to:

- Describe your algorithm
- Prove correctness
- Analyze complexity

Problem 3 & 4

Check if you're supposed to use a specific technique:

- “design a greedy algorithm”
- “design a divide and conquer algorithm”

(Using a different technique will cost you a significant number of marks, but may still be better than a poorly explained incorrect attempt)

Let the running time requirement guide you:

- If we ask $O(1)$ time, this limits your options considerably
- If we ask $O(n)$ time, you can't sort the input

Exam technique

Read all questions to see which ones you can answer quickly

Plan how you will allocate time (wisely)

Start with easy problems and move to harder ones

Write clearly and efficiently

- Start with outline/bullet points, then expand if you have time
- No need for fancy style or overly formal

Figures take a lot of time to draw

- Describing in text is faster (a graph is a bunch of vertices and edges)
- **No scanned handwriting or paragraphs of text in figures allowed**

Pragmatic Advice

- Practice submitting a file in Canvas!
- Be alone in your room to avoid distractions
- Let housemates know when your exam is to avoid distractions
- Bring water
- Have clothing in layers
- Start your submission with your student ID (can prepare file in advance)
- Do not write your name on the exam (marking is anonymous)
- Breathe
- Relax
- Do not contact teaching staff during the exam

Good Luck!!!