

Lecture 1.1 - Introduction to Jupyter notebooks and Google Colab	4
Lecture 1.3 - Python recap - I	27
Lecture 1.4 - Python recap - II	40
Lecture 1.5 - Python recap - III	58
Lecture 1.6 - Exception handling	75
Lecture 1.7 - Classes and Objects	95
Lecture 1.9 - Timing our code	111
Lecture 1.11 - Why efficiency matters	122
Lecture 2.1 - Analysis of Algorithms	170
Lecture 2.2 - Comparing Orders of Magnitude	212
Lecture 2.3 - Calculating Complexity	244
Lecture 2.4 - Searching in a List	262
Lecture 2.5 - Selection Sort	287
Lecture 2.6 - Inserting Sort	317
Lecture 2.7 - Merge Sort	352
Lecture 2.8 - Analysis of Merge Sort	411
Lecture 3.1 - Quicksort	438
Lecture 3.2 - Analysis of Quicksort	481
Lecture 3.4 - Concluding remarks on sorting algorithms	502
Lecture 3.5 - Difference between Lists and Arrays (Theory)	515
Lecture 3.6 - Designing a Flexible List and Operations on the same	536
Lecture 3.7 - Implementation of Lists in Python	555
Lecture 3.8 - Implementation of Dictionaries in Python	571
Lecture 4.1 - Introduction to graphs	588
Lecture 4.2 - Representing Graphs	624
Lecture 4.3 - Breadth First Search (BFS)	664
Lecture 4.4 - Depth First Search (DFS)	744
Lecture 4.5 - Applications of BFS and DFS	782
Lecture 4.6 - Introduction to Directed Acyclic Graph (DAG)	858
Lecture 4.7 - Topological Sorting	877

Lecture 4.8 - Longest Path in DAGs	915
Lecture 5.1 - Shortest Paths in Weighted Graphs	963
Lecture 5.2 - Single Source Shortest Paths (Dijkstra_s algorithm)	976
Lecture 5.3 - Single Source Shortest Paths with Negative Weights (Bellman-Ford Algorithm)	1022
Lecture 5.4 - All Pairs Shortest Paths (Floyd-Warshall algorithm)	1064
Lecture 5.5 - Minimum Cost Spanning Trees	1096
Lecture 5.6 - Minimum Cost Spanning Trees (Prim_s Algorithm)	1116
Lecture 5.7 - Minimum Cost Spanning Trees (Kruskal_s Algorithm)	1172
Lecture 6.1 - Union-Find Data Structure	1216
Lecture 6.2 - Priority Queues	1252
Lecture 6.3 - Heaps	1283
Lecture 6.4 - Using Heaps in Algorithms	1330
Lecture 6.5 - Search Trees	1362
Lecture 7.1 - Balanced Search Trees	1408
Lecture 7.2 - Greedy Algorithms-Interval Scheduling	1450
Lecture 7.3 - Greedy Algorithms-Minimizing Lateness	1517
Lecture 7.4 - Greedy Algorithms-Huffman Coding	1561
Lecture 8.1 - Divide and Conquer-Counting inversions	1652
Lecture 8.2 - Divide and Conquer-Closest Pair of Points	1701
Lecture 8.3 - Divide and Conquer-Integer Multiplication	1747
Lecture 8.4 - Divide and Conquer-Recursion Trees	1792
Lecture 8.5 - Divide and Conquer-Quick Select	1824
Lecture 9.1 - Dynamic Programming	1859
Lecture 9.2 - Memoization	1901
Lecture 9.3 - Grid Paths	1960
Lecture 9.4 - Common Subwords and Subsequences	2020
Lecture 9.5 - Edit Distance	2079
Lecture 9.6 - Matrix Multiplication	2132
Lecture 10.1 - String Matching	2185
Lecture 10.2 - String Matching-Boyer-Moore Algorithm	2199

Lecture 10.3 - String Matching-Rabin-Karp Algorithm	2233
Lecture 10.4 - String Matching Using Automata	2259
Lecture 10.5 - String Matching-Knuth-Morris-Pratt Algorithm	2288
Lecture 10.6 - String Matching-Tries	2316
Lecture 10.7 - String Matching-Regular Expressions	2355
Lecture 11.1 - Linear Programming	2391
Lecture 11.2 - Linear Programming-Production Planning	2452
Lecture 11.3 - Linear Programming-Bandwidth Allocation	2488
Lecture 11.4 - Network Flows	2513
Lecture 11.5 - Reductions	2553
Lecture 11.6 - Intractability-Checking Algorithms	2581
Lecture 11.7 - Intractability-P and NP	2649
Tutorial 12.1 - Summary of weeks 1 to 3	2695
Tutorial 12.2 - Summary of weeks 4 to 6	2712
Tutorial 12.3 - Summary of weeks 7 to 9	2733
Tutorial 12.4 - Summary of weeks 10 _ 11	2753

Jupyter Notebooks

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

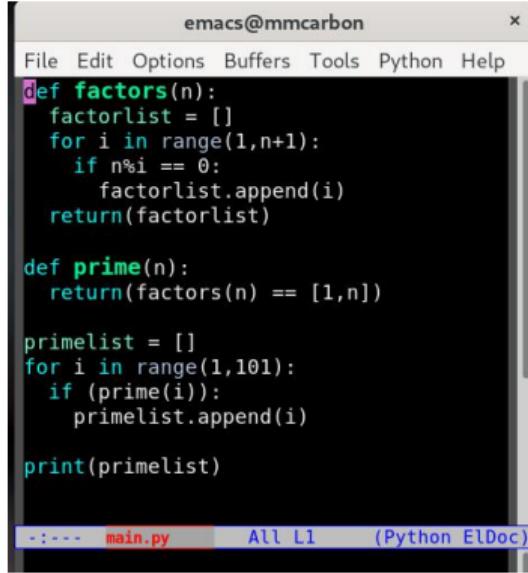
Programming, Data Structures and Algorithms using Python

Week 1

Writing and running code

■ Manual

- Text editor to write code
- Run at the command line



The screenshot shows an Emacs window titled "emacs@mmcarbon". The menu bar includes File, Edit, Options, Buffers, Tools, Python, and Help. The code buffer contains the following Python script:

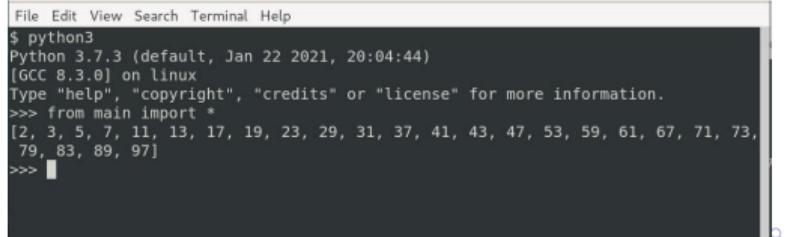
```
def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n%i == 0:
            factorlist.append(i)
    return(factorlist)

def prime(n):
    return(factors(n) == [1,n])

primelist = []
for i in range(1,101):
    if (prime(i)):
        primelist.append(i)

print(primelist)
```

The status bar at the bottom shows "-:--- main.py All L1 (Python ElDoc)".



The screenshot shows a terminal window with the following session:

```
File Edit View Search Terminal Help
$ python3
Python 3.7.3 (default, Jan 22 2021, 20:04:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from main import *
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
 79, 83, 89, 97]
>>> █
```

Writing and running code

■ Manual

- Text editor to write code
- Run at the command line

■ Integrated Development Environment (IDE)

- Single application to write and run code
- On desktop or online, [replit](#)
- Quick update-run cycle
- Debugging, testing, ...

The screenshot shows a Python IDE interface. The main window displays a file named 'main.py' with the following code:

```
1 def factors(n):
2     factorlist = []
3     for i in range(1,n+1):
4         if n%i == 0:
5             factorlist.append(i)
6     return(factorlist)
7
8 def prime(n):
9     return(factors(n) == [1,n])
10
11 primelist = []
12 for i in range(1,101):
13     if (prime(i)):
14         primelist.append(i)
15
16 print(primelist)
```

To the right of the code editor are two tabs: 'Console' and 'Shell'. The 'Console' tab contains the output of the code execution:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23 Q x
31, 37, 41, 43, 47, 53, 59, 61, 67, 7
1, 73, 79, 83, 89, 97]
> []
```

Writing and running code

■ Manual

- Text editor to write code
- Run at the command line

■ Integrated Development Environment (IDE)

- Single application to write and run code
- On desktop or online, [replit](#)
- Quick update-run cycle
- Debugging, testing, ...

■ What more could one want?

The screenshot shows a Jupyter Notebook interface with a code cell containing Python code and its output. The code defines two functions: `factors(n)` which returns a list of factors for a given number, and `prime(n)` which checks if a number is prime by comparing its factors to [1, n]. The output cell shows the prime numbers from 2 to 101.

```
def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n%i == 0:
            factorlist.append(i)
    return(factorlist)

def prime(n):
    return(factors(n) == [1,n])

primelist = []
for i in range(1,101):
    if (prime(i)):
        primelist.append(i)

print(primelist)
```

[2, 3, 5, 7, 11, 13, 17, 19, 23 Q x
31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

Collaboration

- Share your code
 - Collaborative development
 - Report your results

jupyter PDSA Week 1 Lecture 1 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Compute primes from 1 to 100

```
In [1]: def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n%i == 0:
            factorlist.append(i)
    return(factorlist)

def prime(n):
    return(factors(n) == [1,n])

primelist = []
for i in range(1,101):
    if (prime(i)):
        primelist.append(i)

print(primelist)
```

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

Collaboration

- Share your code
 - Collaborative development
 - Report your results
- Documentation
 - Interleave with the code

jupyter PDSA Week 1 Lecture 1 (autosaved) Logout Trusted Python 3

Compute primes from 1 to 100

```
In [1]: def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n%i == 0:
            factorlist.append(i)
    return(factorlist)

def prime(n):
    return(factors(n) == [1,n])

primelist = []
for i in range(1,101):
    if (prime(i)):
        primelist.append(i)

print(primelist)
```

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

Collaboration

- Share your code
 - Collaborative development
 - Report your results
- Documentation
 - Interleave with the code
- Switch between different versions of code

The screenshot shows a Jupyter Notebook interface with the title "jupyter PDSA Week 1 Lecture 1 (unsaved changes)". The notebook has a "Trusted" status and is using a "Python 3" kernel. The code in the notebook is as follows:

```
In [1]: def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n % i == 0:
            factorlist.append(i)
    return(factorlist)

In [3]: def prime(n):
    return(factors(n) == [1,n])

In [2]: def prime(n):
    return(len(factors(n)) == 2)

In [4]: primelist = []
for i in range(1,101):
    if (prime(i)):
        primelist.append(i)

print(primelist)
```

The output of the final cell (In [4]) is:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Collaboration

- Share your code
 - Collaborative development
 - Report your results
- Documentation
 - Interleave with the code
- Switch between different versions of code
- Export and import your project
- Preserve your output

jupyter PDSA Week 1 Lecture 1 (unsaved changes) Logout Trusted Python 3

In [6]:

```
import matplotlib.pyplot as plt
import numpy as np

# Data for plotting
t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

fig, ax = plt.subplots()
ax.plot(t, s)

plt.show()
```

0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00

0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00

Jupyter notebook

- A sequence of cells
 - Like a one dimensional spreadsheet

jupyter PDSA Week 1 Lecture 1 (unsaved changes) Logout Trusted Python 3

File Edit View Insert Cell Kernel Widgets Help

Compute primes from 1 to 100

```
In [1]: def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n%i == 0:
            factorlist.append(i)
    return(factorlist)

In [3]: def prime(n):
    return(factors(n) == [1,n])

In [2]: def prime(n):
    return(len(factors(n)) == 2)

In [4]: primelist = []
for i in range(1,101):
    if (prime(i)):
        primelist.append(i)

print(primelist)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Jupyter notebook

- A sequence of cells
 - Like a one dimensional spreadsheet
- Cells hold code or text
 - Markdown notation for formatting
 - <https://www.markdownguide.org/>

The screenshot shows a Jupyter Notebook interface with the title "jupyter PDSA Week 1 Lecture 1 (unsaved changes)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, and Python 3. The notebook contains the following code:

```
In [1]: def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n % i == 0:
            factorlist.append(i)
    return(factorlist)

In [3]: def prime(n):
    return(factors(n) == [1,n])

In [2]: def prime(n):
    return(len(factors(n)) == 2)

In [4]: primelist = []
for i in range(1,101):
    if (prime(i)):
        primelist.append(i)

print(primelist)
```

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

Jupyter notebook

- A sequence of cells
 - Like a one dimensional spreadsheet
- Cells hold code or text
 - Markdown notation for formatting
 - <https://www.markdownguide.org/>
- Edit and re-run individual cells to update environment

The screenshot shows a Jupyter Notebook interface with the title "jupyter PDSA Week 1 Lecture 1 (unsaved changes)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, and Python 3. The notebook contains the following code:

```
In [1]: def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n % i == 0:
            factorlist.append(i)
    return(factorlist)

In [3]: def prime(n):
    return(factors(n) == [1,n])

In [2]: def prime(n):
    return(len(factors(n)) == 2)

In [4]: primelist = []
for i in range(1,101):
    if (prime(i)):
        primelist.append(i)

print(primelist)
```

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

Jupyter notebook ...

- Supports different **kernels**
 - **Julia, Python, R**
 - We will use it only for Python

jupyter PDSA Week 1 Lecture 1 (unsaved changes)  Logout Trusted Python 3

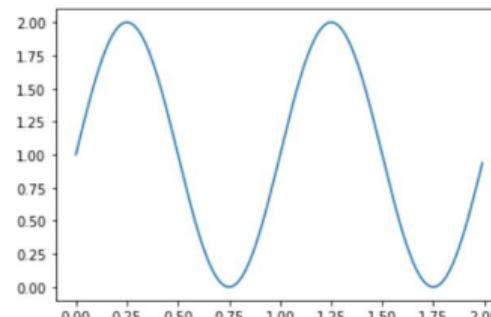
In [6]:

```
import matplotlib.pyplot as plt
import numpy as np

# Data for plotting
t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

fig, ax = plt.subplots()
ax.plot(t, s)

plt.show()
```



Jupyter notebook ...

- Supports different **kernels**
 - **Julia, Python, R**
 - We will use it only for Python
- Widely used to document and disseminate ML projects
 - Solutions to problems posed on platforms like Kaggle <https://www.kaggle.org>

jupyter PDSA Week 1 Lecture 1 (unsaved changes) Logout Trusted Python 3

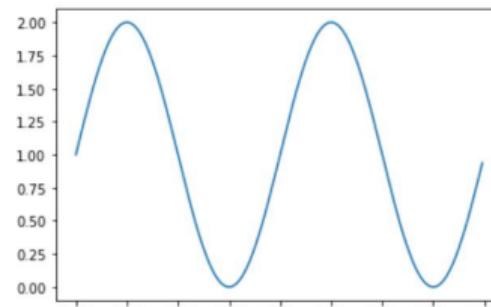
In [6]:

```
import matplotlib.pyplot as plt
import numpy as np

# Data for plotting
t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

fig, ax = plt.subplots()
ax.plot(t, s)

plt.show()
```



Navigation icons: back, forward, search, etc.

Jupyter notebook ...

- Supports different **kernels**
 - **Julia, Python, R**
 - We will use it only for Python
- Widely used to document and disseminate ML projects
 - Solutions to problems posed on platforms like Kaggle <https://www.kaggle.org>
- ACM Software Systems Award 2017

jupyter PDSA Week 1 Lecture 1 (unsaved changes) Logout Trusted Python 3

```
In [6]: import matplotlib.pyplot as plt
import numpy as np

# Data for plotting
t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

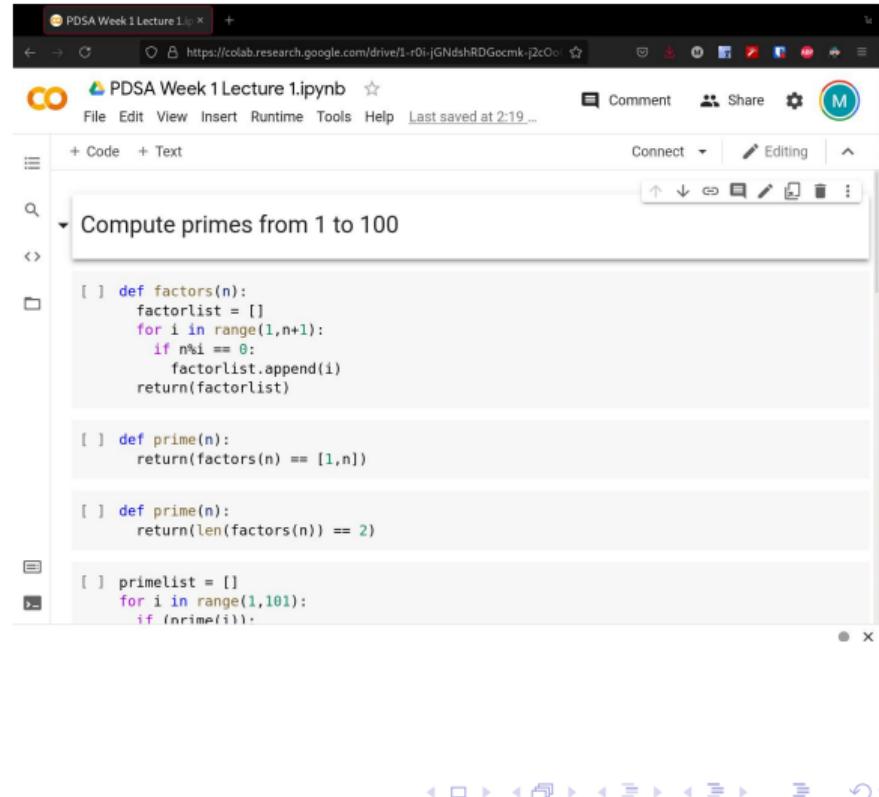
fig, ax = plt.subplots()
ax.plot(t, s)

plt.show()
```

Navigation icons: back, forward, search, etc.

Google Colab

- Google Colaboratory (Colab)
 - colab.research.google.com
 - Free to use



The screenshot shows a Google Colab notebook titled "PDSA Week 1 Lecture 1.ipynb". The code cell contains the following Python code:

```
[ ] def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n%i == 0:
            factorlist.append(i)
    return(factorlist)

[ ] def prime(n):
    return(factors(n) == [1,n])

[ ] def prime(n):
    return(len(factors(n)) == 2)

[ ] primelist = []
for i in range(1,101):
    if (prime(i)):
```

Google Colab

- Google Colaboratory (Colab)
 - colab.research.google.com
 - Free to use
- Customized Jupyter notebook

The screenshot shows a web browser window for Google Colab. The title bar reads "PDSA Week 1 Lecture 1.ipynb". The main area displays a Jupyter notebook with the following code:

```
[ ] def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n%i == 0:
            factorlist.append(i)
    return(factorlist)

[ ] def prime(n):
    return(factors(n) == [1,n])

[ ] def prime(n):
    return(len(factors(n)) == 2)

[ ] primelist = []
for i in range(1,101):
    if (prime(i)):
```

Google Colab

- Google Colaboratory (Colab)
 - colab.research.google.com
 - Free to use
- Customized Jupyter notebook
- All standard packages required for ML are preloaded
 - [scikit-learn](#), [tensorflow](#)
 - Access to GPU hardware

The screenshot shows a web browser window for Google Colab. The title bar reads "PDSA Week 1 Lecture 1.ipynb". The main area displays a Jupyter notebook cell with the following Python code:

```
[ ] def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n%i == 0:
            factorlist.append(i)
    return(factorlist)

[ ] def prime(n):
    return(factors(n) == [1,n])

[ ] def prime(n):
    return(len(factors(n)) == 2)

[ ] primelist = []
for i in range(1,101):
    if (prime(i)):
```

Summary

- Jupyter notebook is a convenient interface to develop Python code

Summary

- Jupyter notebook is a convenient interface to develop Python code
- Incrementally update and run

Summary

- Jupyter notebook is a convenient interface to develop Python code
- Incrementally update and run
- Embed documentation using Markdown

Summary

- Jupyter notebook is a convenient interface to develop Python code
- Incrementally update and run
- Embed documentation using Markdown
- Preserve outputs when exporting

Summary

- Jupyter notebook is a convenient interface to develop Python code
- Incrementally update and run
- Embed documentation using Markdown
- Preserve outputs when exporting
- Useful for collaboration, sharing

Summary

- Jupyter notebook is a convenient interface to develop Python code
- Incrementally update and run
- Embed documentation using Markdown
- Preserve outputs when exporting
- Useful for collaboration, sharing
- Google Colab — free to use version configured for ML

Python Recap – I

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 1

Computing gcd

- $\text{gcd}(m, n)$ — greatest common divisor
 - Largest k that divides both m and n
 - $\text{gcd}(8, 12) = 4$
 - $\text{gcd}(18, 25) = 1$
 - Also **hcf** — highest common factor

Computing gcd

- $\text{gcd}(m, n)$ — greatest common divisor
 - Largest k that divides both m and n
 - $\text{gcd}(8, 12) = 4$
 - $\text{gcd}(18, 25) = 1$
 - Also **hcf** — highest common factor
- $\text{gcd}(m, n)$ always exists
 - 1 divides both m and n

Computing gcd

- $\text{gcd}(m, n)$ — greatest common divisor

- Largest k that divides both m and n
- $\text{gcd}(8, 12) = 4$
- $\text{gcd}(18, 25) = 1$
- Also **hcf** — highest common factor

- $\text{gcd}(m, n)$ always exists

- 1 divides both m and n

- Computing $\text{gcd}(m, n)$

- $\text{gcd}(m, n) \leq \min(m, n)$
- Compute list of common factors from 1 to $\min(m, n)$
- Return the last such common factor

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])
```

Computing gcd

Points to note

- Need to initialize `cf` for `cf.append()` to work
 - Variables (names) derive their type from the value they hold

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])
```

Computing gcd

Points to note

- Need to initialize `cf` for `cf.append()` to work
 - Variables (names) derive their type from the value they hold
- Control flow
 - Conditionals (`if`)
 - Loops (`for`)

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])
```

Computing gcd

Points to note

- Need to initialize `cf` for `cf.append()` to work
 - Variables (names) derive their type from the value they hold
- Control flow
 - Conditionals (`if`)
 - Loops (`for`)
- `range(i,j)` runs from `i` to `j-1`

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])
```

Computing gcd

Points to note

- Need to initialize `cf` for `cf.append()` to work
 - Variables (names) derive their type from the value they hold
- Control flow
 - Conditionals (`if`)
 - Loops (`for`)
- `range(i,j)` runs from `i` to `j-1`
- List indices run from `0` to `len(l) - 1` and backwards from `-1` to `-len(l)`

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])
```

Computing gcd

Eliminate the list

- Only the last value of `cf` is important

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])
```

Computing gcd

Eliminate the list

- Only the last value of `cf` is important
- Keep track of most recent common factor (`mrcf`)

```
def gcd(m,n):  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            mrcf = i  
    return(mrcf)
```

Computing gcd

Eliminate the list

- Only the last value of `cf` is important
- Keep track of most recent common factor (`mrcf`)
- Recall that `1` is always a common factor
 - No need to initialize `mrcf`

```
def gcd(m,n):  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            mrcf = i  
    return(mrcf)
```

Computing gcd

Eliminate the list

- Only the last value of `cf` is important
- Keep track of most recent common factor (`mrcf`)
- Recall that `1` is always a common factor
 - No need to initialize `mrcf`

Efficiency

- Both versions of `gcd` take time proportional to $\min(m, n)$

```
def gcd(m,n):  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            mrcf = i  
    return(mrcf)
```

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])
```

Computing gcd

Eliminate the list

- Only the last value of `cf` is important
- Keep track of most recent common factor (`mrcf`)
- Recall that `1` is always a common factor
 - No need to initialize `mrcf`

Efficiency

- Both versions of `gcd` take time proportional to $\min(m, n)$
- Can we do better?

```
def gcd(m,n):  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            mrcf = i  
    return(mrcf)
```

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])
```

Python Recap – II

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 1

Checking primality

- A prime number n has exactly two factors, 1 and n
 - Note that 1 is **not** a prime

Checking primality

- A prime number n has exactly two factors, 1 and n
 - Note that 1 is **not** a prime
- Compute the list of factors of n

```
def factors(n):  
    fl = []    # factor list  
    for i in range(1,n+1):  
        if (n%i) == 0:  
            fl.append(i)  
    return(fl)
```

Checking primality

- A prime number n has exactly two factors, 1 and n
 - Note that 1 is **not** a prime
- Compute the list of factors of n
- n is a prime if the list of factors is precisely [1, n]

```
def factors(n):  
    fl = []    # factor list  
    for i in range(1,n+1):  
        if (n%i) == 0:  
            fl.append(i)  
    return(fl)  
  
def prime(n):  
    return(factors(n) == [1,n])
```

Counting primes

- List all primes upto m

```
def primesupto(m):  
    pl = [] # prime list  
    for i in range(1,m+1):  
        if prime(i):  
            pl.append(i)  
    return(pl)
```

Counting primes

- List all primes upto m
- List the first m primes
 - Multiple simultaneous assignment

```
def primesupto(m):  
    pl = [] # prime list  
    for i in range(1,m+1):  
        if prime(i):  
            pl.append(i)  
    return(pl)
```

```
def firstprimes(m):  
    (count,i,pl) = (0,1,[])  
    while (count < m):  
        if prime(i):  
            (count,pl) = (count+1,pl+[i])  
        i = i+1  
    return(pl)
```

Counting primes

- List all primes upto m
- List the first m primes
 - Multiple simultaneous assignment
- `for` vs `while`
 - Is the number of iterations known in advance?
 - Ensure progress to guarantee termination of `while`

```
def primesupto(m):  
    pl = []    # prime list  
    for i in range(1,m+1):  
        if prime(i):  
            pl.append(i)  
    return(pl)  
  
def firstprimes(m):  
    (count,i,pl) = (0,1,[])  
    while (count < m):  
        if prime(i):  
            (count,pl) = (count+1,pl+[i])  
        i = i+1  
    return(pl)
```

Computing primes

- Directly check if n has a factor between 2 and $n - 1$

```
def prime(n):  
    result = True  
    for i in range(2,n):  
        if (n%i) == 0:  
            result = False  
    return(result)
```

Computing primes

- Directly check if n has a factor between 2 and $n - 1$
- Terminate check after we find first factor
 - Breaking out of a loop

```
def prime(n):  
    result = True  
    for i in range(2,n):  
        if (n%i) == 0:  
            result = False  
    return(result)
```

```
def prime(n):  
    result = True  
    for i in range(2,n):  
        if (n%i) == 0:  
            result = False  
            break    # Abort loop  
    return(result)
```

Computing primes

- Directly check if n has a factor between 2 and $n - 1$
- Terminate check after we find first factor
 - Breaking out of a loop
- Alternatively, use `while`

```
def prime(n):  
    result = True  
    for i in range(2,n):  
        if (n%i) == 0:  
            result = False  
            break    # Abort loop  
    return(result)
```

```
def prime(n):  
    (result,i) = (True,2)  
    while (result and (i < n)):  
        if (n%i) == 0:  
            result = False  
        i = i+1  
    return(result)
```

Computing primes

- Directly check if n has a factor between 2 and $n - 1$
- Terminate check after we find first factor
 - Breaking out of a loop
- Alternatively, use `while`
- Speeding things up slightly
 - Factors occur in pairs
 - Sufficient to check factors upto \sqrt{n}
 - If n is prime, scan $2, \dots, \sqrt{n}$ instead of $2, \dots, n - 1$

```
import math
def prime(n):
    (result,i) = (True,2)
    while (result and (i < math.sqrt(n))):
        if (n%i) == 0:
            result = False
        i = i+1
    return(result)
```

Properties of primes

- There are infinitely many primes

Properties of primes

- There are infinitely many primes
- How are they distributed?

Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: $p, p + 2$

Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: $p, p + 2$
- Twin prime conjecture
There are infinitely many twin primes

Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: $p, p + 2$
- Twin prime conjecture
There are infinitely many twin primes
- Compute the differences between primes

Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: $p, p + 2$
- Twin prime conjecture
There are infinitely many twin primes
- Compute the differences between primes
- Use a dictionary
- Start checking from 3, since 2 is the smallest prime

Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: $p, p + 2$
- Twin prime conjecture
There are infinitely many twin primes
- Compute the differences between primes
- Use a dictionary
- Start checking from 3, since 2 is the smallest prime

```
def primediffs(n):  
    lastprime = 2  
    pd = {} # Dictionary for  
            # prime differences  
    for i in range(3,n+1):  
        if prime(i):  
            d = i - lastprime  
            lastprime = i  
            if d in pd.keys():  
                pd[d] = pd[d] + 1  
            else:  
                pd[d] = 1  
    return(pd)
```

Python Recap – III

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 1

Computing gcd

- Both versions of `gcd` take time proportional to $\min(m, n)$
- Can we do better?

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])
```

```
def gcd(m,n):  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            mrcf = i  
    return(mrcf)
```

Computing gcd

- Both versions of `gcd` take time proportional to $\min(m, n)$
- Can we do better?
- Suppose d divides m and n
 - $m = ad, n = bd$
 - $m - n = (a - b)d$
 - d also divides $m - n$

```
def gcd(m,n):  
    cf = [] # List of common factors  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            cf.append(i)  
    return(cf[-1])
```

```
def gcd(m,n):  
    for i in range(1,min(m,n)+1):  
        if (m%i) == 0 and (n%i) == 0:  
            mrcf = i  
    return(mrcf)
```

Computing gcd

- Both versions of `gcd` take time proportional to $\min(m, n)$

- Can we do better?

- Suppose d divides m and n

- $m = ad, n = bd$
 - $m - n = (a - b)d$
 - d also divides $m - n$

- Recursively defined function

- Base case: n divides m , answer is n
 - Otherwise, reduce $\text{gcd}(m, n)$ to $\text{gcd}(n, m - n)$

```
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else  
        return(gcd(b,a-b))
```

Computing gcd

- Unfortunately, this takes time proportional to $\max(m, n)$

```
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else  
        return(gcd(b,a-b))
```

Computing gcd

- Unfortunately, this takes time proportional to $\max(m, n)$
- Consider $\text{gcd}(2, 9999)$
 - $\rightarrow \text{gcd}(2, 9997)$
 - $\rightarrow \text{gcd}(2, 9995)$
 - \dots
 - $\rightarrow \text{gcd}(2, 3)$
 - $\rightarrow \text{gcd}(2, 1)$
 - $\rightarrow 1$

```
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else  
        return(gcd(b,a-b))
```

Computing gcd

- Unfortunately, this takes time proportional to $\max(m, n)$
- Consider $\text{gcd}(2, 9999)$
 - $\rightarrow \text{gcd}(2, 9997)$
 - $\rightarrow \text{gcd}(2, 9995)$
 - \dots
 - $\rightarrow \text{gcd}(2, 3)$
 - $\rightarrow \text{gcd}(2, 1)$
 - $\rightarrow 1$
- Approximately 5000 steps

```
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else  
        return(gcd(b,a-b))
```

Computing gcd

- Unfortunately, this takes time proportional to $\max(m, n)$
- Consider $\text{gcd}(2, 9999)$
 - $\rightarrow \text{gcd}(2, 9997)$
 - $\rightarrow \text{gcd}(2, 9995)$
 - \dots
 - $\rightarrow \text{gcd}(2, 3)$
 - $\rightarrow \text{gcd}(2, 1)$
 - $\rightarrow 1$
- Approximately 5000 steps
- Can we do better?

```
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else  
        return(gcd(b,a-b))
```

Euclid's algorithm

- Suppose n does not divide m

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad, n = bd$

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad, n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad, n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad, n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd
- Euclid's algorithm
 - If n divides m , $\text{gcd}(m, n) = n$
 - Otherwise, compute $\text{gcd}(n, m \bmod n)$

```
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else  
        return(gcd(b,a%b))
```

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad, n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd
- Euclid's algorithm
 - If n divides m , $\text{gcd}(m, n) = n$
 - Otherwise, compute $\text{gcd}(n, m \bmod n)$

```
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else  
        return(gcd(b,a%b))
```

- Can show that this takes time proportional to number of digits in $\max(m, n)$

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad, n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd
- Euclid's algorithm
 - If n divides m , $\gcd(m, n) = n$
 - Otherwise, compute $\gcd(n, m \bmod n)$

```
def gcd(m,n):  
    (a,b) = (max(m,n), min(m,n))  
    if a%b == 0:  
        return(b)  
    else  
        return(gcd(b,a%b))
```

- Can show that this takes time proportional to number of digits in $\max(m, n)$
- One of the first non-trivial algorithms

Exception handling

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 1

When things go wrong

- Our code could generate many types of errors
 - `y = x/z`, but `z` has value 0
 - `y = int(s)`, but string `s` does not represent a valid integer
 - `y = 5*x`, but `x` does not have a value
 - `y = l[i]`, but `i` is not a valid index for list `l`
 - Try to read from a file, but the file does not exist
 - Try to write to a file, but the disk is full

When things go wrong

- Our code could generate many types of errors
 - `y = x/z`, but `z` has value 0
 - `y = int(s)`, but string `s` does not represent a valid integer
 - `y = 5*x`, but `x` does not have a value
 - `y = l[i]`, but `i` is not a valid index for list `l`
 - Try to read from a file, but the file does not exist
 - Try to write to a file, but the disk is full
- Recovering gracefully
 - Try to anticipate errors
 - Provide a contingency plan
 - **Exception handling**

Types of errors

- Python flags the type of each error

Types of errors

- Python flags the type of each error
- Most common error is a syntax error
 - `SyntaxError: invalid syntax`
 - Not much you can do!

Types of errors

- Python flags the type of each error
- Most common error is a syntax error
 - `SyntaxError: invalid syntax`
 - Not much you can do!
- We are interested in errors when the code is running
 - Name used before value is defined
`NameError: name 'x' is not defined`
 - Division by zero in arithmetic expression
`ZeroDivisionError: division by zero`
 - Invalid list index
`IndexError: list assignment index out of range`

Terminology

- Raise an exception
 - Run time error → signal error type, with diagnostic information

```
NameError: name 'x' is not defined
```
- Handle an exception
 - Anticipate and take corrective action based on error type
- Unhandled exception aborts execution

Terminology

- Raise an exception
 - Run time error → signal error type, with diagnostic information
`NameError: name 'x' is not defined`
- Handle an exception
 - Anticipate and take corrective action based on error type
- Unhandled exception aborts execution

Handling exceptions

```
try:  
    ... ← Code where error may occur  
    ...  
  
except IndexError:  
    ... ← Handle IndexError  
  
except (NameError, KeyError):  
    ... ← Handle multiple exception types  
  
except:  
    ... ← Handle all other exceptions  
  
else:  
    ... ← Execute if try runs without errors
```

Using exceptions “positively”

- Collect scores in dictionary

```
scores = {"Shefali": [3, 22],  
          "Harmanpreet": [200, 3]}
```

- Update the dictionary

- Batter **b** already exists, append to list

```
scores[b].append(s)
```

- New batter, create a fresh entry

```
scores[b] = [s]
```

Using exceptions “positively”

- Collect scores in dictionary

```
scores = {"Shefali": [3, 22],  
          "Harmanpreet": [200, 3]}
```

- Update the dictionary

- Batter **b** already exists, append to list

```
scores[b].append(s)
```

- New batter, create a fresh entry

```
scores[b] = [s]
```

Traditional approach

```
if b in scores.keys():  
    scores[b].append(s)  
else:  
    scores[b] = [s]
```

Using exceptions “positively”

- Collect scores in dictionary

```
scores = {"Shefali": [3, 22],  
          "Harmanpreet": [200, 3]}
```

- Update the dictionary

- Batter **b** already exists, append to list

```
scores[b].append(s)
```

- New batter, create a fresh entry

```
scores[b] = [s]
```

Traditional approach

```
if b in scores.keys():  
    scores[b].append(s)  
else:  
    scores[b] = [s]
```

Using exceptions

```
try:  
    scores[b].append(s)  
except KeyError:  
    scores[b] = [s]
```

Flow of control

...

x = f(y,z)

Flow of control

...

x = f(y,z)

```
def f(a,b):  
    ...  
    g(a)
```

Flow of control

```
...  
x = f(y,z)  
  
def f(a,b):  
    ...  
    g(a)           def g(m):  
    ...  
    h(m)
```

Flow of control

...

x = f(y,z)

```
def f(a,b):
```

...

g(a)

```
def g(m):
```

...

h(m)

```
def h(s):
```

...

h(s)

Flow of control

...

x = f(y,z)

def f(a,b):

...

g(a)

def g(m):

...

h(m)

def h(s):

...

h(s)

IndexError not
handled in h()

Flow of control

...

x = f(y,z)

```
def f(a,b):
```

...

g(a)

```
def g(m):
```

...

h(m)

```
def h(s):
```

...

h(s)

IndexError

inherited from h()

IndexError not
handled in h()

Flow of control

...

x = f(y,z)

def f(a,b):

...

g(a)

IndexError
inherited from g()

def g(m):

...

h(m)

IndexError
inherited from h()
Not handled?

def h(s):

...

h(s)

IndexError not
handled in h()

Flow of control

...

x = f(y,z)

IndexError

inherited from f()

def f(a,b):

...

g(a)

IndexError

inherited from g()

Not handled?

def g(m):

...

h(m)

IndexError

inherited from h()

Not handled?

def h(s):

...

h(s)

IndexError not
handled in h()

Flow of control

...

x = f(y,z)

IndexError

inherited from f()

Not handled?

Abort!

def f(a,b):

...

g(a)

def g(m):

...

h(m)

def h(s):

...

h(s)

IndexError

inherited from g()

Not handled?

IndexError

inherited from h()

Not handled?

IndexError not

handled in h()

Classes and objects

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 1

Classes and objects

■ Abstract datatype

- Stores some information
- Designated functions to manipulate the information
- For instance, stack: last-in, first-out, `push()`, `pop()`

Classes and objects

- Abstract datatype
 - Stores some information
 - Designated functions to manipulate the information
 - For instance, stack: last-in, first-out, `push()`, `pop()`
- Separate the (private) implementation from the (public) specification

Classes and objects

- Abstract datatype
 - Stores some information
 - Designated functions to manipulate the information
 - For instance, stack: last-in, first-out, `push()`, `pop()`
- Separate the (private) implementation from the (public) specification
- Class
 - Template for a data type
 - How data is stored
 - How public functions manipulate data

Classes and objects

- Abstract datatype
 - Stores some information
 - Designated functions to manipulate the information
 - For instance, stack: last-in, first-out, `push()`, `pop()`
- Separate the (private) implementation from the (public) specification
- Class
 - Template for a data type
 - How data is stored
 - How public functions manipulate data
- Object
 - Concrete instance of template

Example: 2D points

- A point has coordinates (x, y)
 - `__init__()` initializes internal values `x, y`
 - First parameter is always `self`
 - Here, by default a point is at $(0, 0)$

```
class Point:  
    def __init__(self,a=0,b=0):  
        self.x = a  
        self.y = b
```

Example: 2D points

- A point has coordinates (x, y)
 - `__init__()` initializes internal values `x, y`
 - First parameter is always `self`
 - Here, by default a point is at $(0, 0)$
- Translation: shift a point by $(\Delta x, \Delta y)$
 - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$

```
class Point:  
    def __init__(self,a=0,b=0):  
        self.x = a  
        self.y = b  
  
    def translate(self,deltax,deltay):  
        self.x += deltax  
        self.y += deltay
```

Example: 2D points

- A point has coordinates (x, y)
 - `__init__()` initializes internal values `x, y`
 - First parameter is always `self`
 - Here, by default a point is at $(0, 0)$
- Translation: shift a point by $(\Delta x, \Delta y)$
 - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$
- Distance from the origin
 - $d = \sqrt{x^2 + y^2}$

```
class Point:  
    def __init__(self,a=0,b=0):  
        self.x = a  
        self.y = b  
  
    def translate(self,deltax,deltay):  
        self.x += deltax  
        self.y += deltay  
  
    def odistance(self):  
        import math  
        d = math.sqrt(self.x*self.x +  
                      self.y*self.y)  
        return(d)
```

Polar coordinates

- (r, θ) instead of (x, y)
 - $r = \sqrt{x^2 + y^2}$
 - $\theta = \tan^{-1}(y/x)$

```
import math
class Point:
    def __init__(self,a=0,b=0):
        self.r = math.sqrt(a*a + b*b)
        if a == 0:
            self.theta = math.pi/2
        else:
            self.theta = math.atan(b/a)
```

Polar coordinates

- (r, θ) instead of (x, y)
 - $r = \sqrt{x^2 + y^2}$
 - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r

```
import math
class Point:
    def __init__(self,a=0,b=0):
        self.r = math.sqrt(a*a + b*b)
        if a == 0:
            self.theta = math.pi/2
        else:
            self.theta = math.atan(b/a)

    def odistance(self):
        return(self.r)
```

Polar coordinates

- (r, θ) instead of (x, y)
 - $r = \sqrt{x^2 + y^2}$
 - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
 - Convert (r, θ) to (x, y)
 - $x = r \cos \theta, y = r \sin \theta$
 - Recompute r, θ from $(x + \Delta x, y + \Delta y)$

```
def translate(self,deltax,deltay):  
    x = self.r*math.cos(self.theta)  
    y = self.r*math.sin(self.theta)  
    x += deltax  
    y += deltay  
    self.r = math.sqrt(x*x + y*y)  
    if x == 0:  
        self.theta = math.pi/2  
    else:  
        self.theta = math.atan(y/x)
```

Polar coordinates

- (r, θ) instead of (x, y)
 - $r = \sqrt{x^2 + y^2}$
 - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
 - Convert (r, θ) to (x, y)
 - $x = r \cos \theta, y = r \sin \theta$
 - Recompute r, θ from $(x + \Delta x, y + \Delta y)$
- Interface has not changed
 - User need not be aware whether representation is (x, y) or (r, θ)

```
def translate(self,deltax,deltay):  
    x = self.r*math.cos(self.theta)  
    y = self.r*math.sin(self.theta)  
    x += deltax  
    y += deltay  
    self.r = math.sqrt(x*x + y*y)  
    if x == 0:  
        self.theta = math.pi/2  
    else:  
        self.theta = math.atan(y/x)
```

Special functions

- `__init__()` — constructor

Special functions

- `__init__()` — constructor
- `__str__()` — convert object to string
 - `str(o) == o.__str__()`
 - Implicitly invoked by `print()`

```
class Point:  
    ...  
  
    def __str__(self):  
        return(  
            '('+str(self.x)+', '  
            +str(self.y)+')'  
)
```

Special functions

- `__init__()` — constructor
- `__str__()` — convert object to string
 - `str(o) == o.__str__()`
 - Implicitly invoked by `print()`
- `__add__()`
 - Implicitly invoked by `+`

```
class Point:
```

```
...
```

```
def __str__(self):  
    return(  
        '('+str(self.x)+', '  
             +str(self.y)+')'  
)
```

```
def __add__(self,p):  
    return(Point(self.x + p.x,  
                self.y + p.y))
```

Special functions

- `__init__()` — constructor
- `__str__()` — convert object to string
 - `str(o) == o.__str__()`
 - Implicitly invoked by `print()`
- `__add__()`
 - Implicitly invoked by `+`
- Similarly
 - `__mult__()` invoked by `*`
 - `__lt__()` invoked by `<`
 - `__ge__()` invoked by `>=`
 - ...

```
class Point:  
    ...  
  
    def __str__(self):  
        return(  
            '('+str(self.x)+', '  
            +str(self.y)+')'  
)
```

```
def __add__(self,p):  
    return(Point(self.x + p.x,  
                self.y + p.y))
```

Timing our code

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 1

Timing our code

- How long does our code take to execute?
 - Depends from one language to another

Timing our code

- How long does our code take to execute?
 - Depends from one language to another
- Python has a library `time` with various useful functions

Timing our code

- How long does our code take to execute?
 - Depends from one language to another
- Python has a library `time` with various useful functions
- `perf_time()` is a performance counter
 - Absolute value of `perf_time()` is not meaningful
 - Compare two consecutive readings to get an interval
 - Default unit is seconds

Timing our code

- How long does our code take to execute?
 - Depends from one language to another
- Python has a library `time` with various useful functions
- `perf_time()` is a performance counter
 - Absolute value of `perf_time()` is not meaningful
 - Compare two consecutive readings to get an interval
 - Default unit is seconds

```
import time

start = time.perf_counter()
...
# Execute some code
...
end = time.perf_counter()

elapsed = end - start
```

A timer object

- Create a timer class

```
import time  
class Timer:
```

A timer object

- Create a timer class
- Two internal values
 - `_start_time`
 - `_elapsed_time`

```
import time
class Timer:

    def __init__(self):
        self._start_time = 0
        self._elapsed_time = 0
```

A timer object

- Create a timer class
- Two internal values
 - `_start_time`
 - `_elapsed_time`
- `start` starts the timer

```
import time
class Timer:

    def __init__(self):
        self._start_time = 0
        self._elapsed_time = 0

    def start(self):
        self._start_time = time.perf_counter()
```

A timer object

- Create a timer class
- Two internal values
 - `_start_time`
 - `_elapsed_time`
- `start` starts the timer
- `stop` records the elapsed time

```
import time
class Timer:

    def __init__(self):
        self._start_time = 0
        self._elapsed_time = 0

    def start(self):
        self._start_time = time.perf_counter()

    def stop(self):
        self._elapsed_time =
            time.perf_counter() - self._start_time

    def elapsed(self):
        return(self._elapsed_time)
```

A timer object

- Create a timer class
- Two internal values
 - `_start_time`
 - `_elapsed_time`
- `start` starts the timer
- `stop` records the elapsed time
- More sophisticated version in the actual code

```
import time
class Timer:

    def __init__(self):
        self._start_time = 0
        self._elapsed_time = 0

    def start(self):
        self._start_time = time.perf_counter()

    def stop(self):
        self._elapsed_time =
            time.perf_counter() - self._start_time

    def elapsed(self):
        return(self._elapsed_time)
```

A timer object

- Create a timer class
- Two internal values
 - `_start_time`
 - `_elapsed_time`
- `start` starts the timer
- `stop` records the elapsed time
- More sophisticated version in the actual code
- Python executes 10^7 operations per second

```
import time
class Timer:

    def __init__(self):
        self._start_time = 0
        self._elapsed_time = 0

    def start(self):
        self._start_time = time.perf_counter()

    def stop(self):
        self._elapsed_time =
            time.perf_counter() - self._start_time

    def elapsed(self):
        return(self._elapsed_time)
```

Why Efficiency Matters

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 1

A real world problem

- Every SIM card needs to be linked to an Aadhaar card

A real world problem

- Every SIM card needs to be linked to an Aadhaar card
- Validate the Aadhaar details for each SIM card

A real world problem

- Every SIM card needs to be linked to an Aadhaar card
- Validate the Aadhaar details for each SIM card
- Simple nested loop

for each SIM card S:

 for each Aadhaar number A:

 check if Aadhaar details of S
 match A

A real world problem

- Every SIM card needs to be linked to an Aadhaar card
- Validate the Aadhaar details for each SIM card
- Simple nested loop
- How long will this take?
 - M SIM cards, N Aadhaar cards
 - Nested loops iterate $M \cdot N$ times

for each SIM card S:

 for each Aadhaar number A:

 check if Aadhaar details of S
 match A

A real world problem

- Every SIM card needs to be linked to an Aadhaar card
- Validate the Aadhaar details for each SIM card
- Simple nested loop
- How long will this take?
 - M SIM cards, N Aadhaar cards
 - Nested loops iterate $M \cdot N$ times
- What are M and N
 - Almost everyone in India has an Aadhaar card: $N > 10^9$
 - Number of SIM cards registered is similar: $M > 10^9$

for each SIM card S:

 for each Aadhaar number A:

 check if Aadhaar details of S
 match A

A real world problem

- Assume $M = N = 10^9$

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second

```
for each SIM card S:  
    for each Aadhaar number A:  
        check if Aadhaar details of S  
        match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds

```
for each SIM card S:  
    for each Aadhaar number A:  
        check if Aadhaar details of S  
        match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds
 - $10^{11}/60 \approx 1.67 \times 10^9$ minutes

```
for each SIM card S:  
    for each Aadhaar number A:  
        check if Aadhaar details of S  
        match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds
 - $10^{11}/60 \approx 1.67 \times 10^9$ minutes
 - $(1.67 \times 10^9)/60 \approx 2.8 \times 10^7$ hours

```
for each SIM card S:  
    for each Aadhaar number A:  
        check if Aadhaar details of S  
        match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds
 - $10^{11}/60 \approx 1.67 \times 10^9$ minutes
 - $(1.67 \times 10^9)/60 \approx 2.8 \times 10^7$ hours
 - $(2.8 \times 10^7)/24 \approx 1.17 \times 10^6$ days

```
for each SIM card S:  
    for each Aadhaar number A:  
        check if Aadhaar details of S  
        match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds
 - $10^{11}/60 \approx 1.67 \times 10^9$ minutes
 - $(1.67 \times 10^9)/60 \approx 2.8 \times 10^7$ hours
 - $(2.8 \times 10^7)/24 \approx 1.17 \times 10^6$ days
 - $(1.17 \times 10^6)/365 \approx 3200$ years!

```
for each SIM card S:  
    for each Aadhaar number A:  
        check if Aadhaar details of S  
        match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds
 - $10^{11}/60 \approx 1.67 \times 10^9$ minutes
 - $(1.67 \times 10^9)/60 \approx 2.8 \times 10^7$ hours
 - $(2.8 \times 10^7)/24 \approx 1.17 \times 10^6$ days
 - $(1.17 \times 10^6)/365 \approx 3200$ years!
- How can we fix this?

```
for each SIM card S:  
    for each Aadhaar number A:  
        check if Aadhaar details of S  
        match A
```

Guess my birthday

- You propose a date

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
- What is the best strategy?

Guess my birthday

- You propose a date
- Interval of possibilities
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
- What is the best strategy?

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
 - June 30? *Earlier*

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
 - June 30? *Earlier*
 - March 31? *Later*

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
 - June 30? *Earlier*
 - March 31? *Later*
 - May 15? *Earlier*

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
 - June 30? *Earlier*
 - March 31? *Later*
 - May 15? *Earlier*
 - April 22? *Earlier*

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
 - June 30? *Earlier*
 - March 31? *Later*
 - May 15? *Earlier*
 - April 22? *Earlier*
 - April 11? *Later*

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
 - June 30? *Earlier*
 - March 31? *Later*
 - May 15? *Earlier*
 - April 22? *Earlier*
 - April 11? *Later*
 - April 16? *Earlier*

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
 - June 30? *Earlier*
 - March 31? *Later*
 - May 15? *Earlier*
 - April 22? *Earlier*
 - April 11? *Later*
 - April 16? *Earlier*
 - April 13? *Earlier*

Guess my birthday

- You propose a date
- I answer, *Yes, Earlier, Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
 - June 30? *Earlier*
 - March 31? *Later*
 - May 15? *Earlier*
 - April 22? *Earlier*
 - April 11? *Later*
 - April 16? *Earlier*
 - April 13? *Earlier*
 - April 12? *Yes*

Guess my birthday

- You propose a date
 - I answer, *Yes, Earlier, Later*
 - Suppose my birthday is 12 April
 - A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
 - What is the best strategy?
-
- Interval of possibilities
 - Query midpoint — halves the interval
 - June 30? *Earlier*
 - March 31? *Later*
 - May 15? *Earlier*
 - April 22? *Earlier*
 - April 11? *Later*
 - April 16? *Earlier*
 - April 13? *Earlier*
 - April 12? *Yes*
 - Interval shrinks from $365 \rightarrow 182 \rightarrow 91 \rightarrow 45 \rightarrow 22 \rightarrow 11 \rightarrow 5 \rightarrow 2 \rightarrow 1$

Guess my birthday

- You propose a date
 - I answer, *Yes, Earlier, Later*
 - Suppose my birthday is 12 April
 - A possible sequence of questions
 - September 12? *Earlier*
 - February 23? *Later*
 - July 2? *Earlier*
 - ...
 - What is the best strategy?
- Interval of possibilities
 - Query midpoint — halves the interval
 - June 30? *Earlier*
 - March 31? *Later*
 - May 15? *Earlier*
 - April 22? *Earlier*
 - April 11? *Later*
 - April 16? *Earlier*
 - April 13? *Earlier*
 - April 12? *Yes*
 - Interval shrinks from $365 \rightarrow 182 \rightarrow 91 \rightarrow 45 \rightarrow 22 \rightarrow 11 \rightarrow 5 \rightarrow 2 \rightarrow 1$
 - Under 10 questions

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card

for each SIM card S:
probe sorted Aadhaar list to
check Aadhaar details of S

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$

for each SIM card S:

probe sorted Aadhaar list to
check Aadhaar details of S

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6

for each SIM card S:
probe sorted Aadhaar list to
check Aadhaar details of S

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3

for each SIM card S:

probe sorted Aadhaar list to
check Aadhaar details of S

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1

for each SIM card S:
probe sorted Aadhaar list to
check Aadhaar details of S

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
 - Use the halving strategy to check each SIM card
 - Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
 - After 10 queries, interval shrinks to 10^6
 - After 20 queries, interval shrinks to 10^3
 - After 30 queries, interval shrinks to 1
 - Total time $\approx 10^9 \times 30$
- for each SIM card S:
probe sorted Aadhaar list to
check Aadhaar details of S

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

for each SIM card S:

probe sorted Aadhaar list to
check Aadhaar details of S

- 3000 seconds, or 50 minutes

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

for each SIM card S:

probe sorted Aadhaar list to
check Aadhaar details of S

- 3000 seconds, or 50 minutes
- From 3200 years to 50 minutes!

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

for each SIM card S:

probe sorted Aadhaar list to
check Aadhaar details of S

- 3000 seconds, or 50 minutes
- From 3200 years to 50 minutes!
- Of course, to achieve this we have to first sort the Aadhaar cards

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

for each SIM card S:

probe sorted Aadhaar list to
check Aadhaar details of S

- 3000 seconds, or 50 minutes
- From 3200 years to 50 minutes!
- Of course, to achieve this we have to first sort the Aadhaar cards
- Arranging the data results in a much more efficient solution

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

for each SIM card S:

probe sorted Aadhaar list to
check Aadhaar details of S

- 3000 seconds, or 50 minutes
- From 3200 years to 50 minutes!
- Of course, to achieve this we have to first sort the Aadhaar cards
- Arranging the data results in a much more efficient solution
- Both algorithms **and** data structures matter

Analysis of algorithms

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 2

Measuring performance

- Example of validating SIM cards against Aadhaar data
 - Naive approach takes thousands of years
 - Smarter solution takes a few minutes

Measuring performance

- Example of validating SIM cards against Aadhaar data
 - Naive approach takes thousands of years
 - Smarter solution takes a few minutes
- Two main resources of interest
 - Running time — how long the algorithm takes
 - Space — memory requirement

Measuring performance

- Example of validating SIM cards against Aadhaar data
 - Naive approach takes thousands of years
 - Smarter solution takes a few minutes
- Two main resources of interest
 - Running time — how long the algorithm takes
 - Space — memory requirement
- Time depends on processing power
 - Impossible to change for given hardware
 - Enhancing hardware has only a limited impact at a practical level

Measuring performance

- Example of validating SIM cards against Aadhaar data
 - Naive approach takes thousands of years
 - Smarter solution takes a few minutes
- Two main resources of interest
 - Running time — how long the algorithm takes
 - Space — memory requirement
- Time depends on processing power
 - Impossible to change for given hardware
 - Enhancing hardware has only a limited impact at a practical level
- Storage is limited by available memory
 - Easier to configure, augment

Measuring performance

- Example of validating SIM cards against Aadhaar data
 - Naive approach takes thousands of years
 - Smarter solution takes a few minutes
- Two main resources of interest
 - Running time — how long the algorithm takes
 - Space — memory requirement
- Time depends on processing power
 - Impossible to change for given hardware
 - Enhancing hardware has only a limited impact at a practical level
- Storage is limited by available memory
 - Easier to configure, augment
- Typically, we focus on time rather than space

Input size

- Running time depends on input size
 - Larger arrays will take longer to sort

Input size

- Running time depends on input size
 - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
 - Input size n
 - Running time $t(n)$

Input size

- Running time depends on input size
 - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
 - Input size n
 - Running time $t(n)$
- Different inputs of size n may take different amounts of time
 - We will return to this point later

Input size

- Running time depends on input size
 - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
 - Input size n
 - Running time $t(n)$
- Different inputs of size n may take different amounts of time
 - We will return to this point later

Example 1 SIM cards vs Aadhaar cards

- $n \approx 10^9$ — number of cards

Input size

- Running time depends on input size
 - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
 - Input size n
 - Running time $t(n)$
- Different inputs of size n may take different amounts of time
 - We will return to this point later

Example 1 SIM cards vs Aadhaar cards

- $n \approx 10^9$ — number of cards
- Naive algorithm: $t(n) \approx n^2$

Input size

- Running time depends on input size
 - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
 - Input size n
 - Running time $t(n)$
- Different inputs of size n may take different amounts of time
 - We will return to this point later

Example 1 SIM cards vs Aadhaar cards

- $n \approx 10^9$ — number of cards
- Naive algorithm: $t(n) \approx n^2$
- Clever algorithm: $t(n) \approx n \log_2 n$
 - $\log_2 n$ — number of times you need to divide n by 2 to reach 1
 - $\log_2(n) = k \Rightarrow n = 2^k$

Input size . . .

Example 2 Video game

- Several objects on screen

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
 - For each pair of objects, compute their distance
 - Report minimum distance across all pairs

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
 - For each pair of objects, compute their distance
 - Report minimum distance across all pairs
- There is a clever algorithm that takes some $n \log_2 n$

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
 - For each pair of objects, compute their distance
 - Report minimum distance across all pairs
- There is a clever algorithm that takes some $n \log_2 n$
- High resolution gaming console may have 4000×2000 pixels
 - 8×10^6 points — 8 million

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
 - For each pair of objects, compute their distance
 - Report minimum distance across all pairs
- There is a clever algorithm that takes some $n \log_2 n$
- High resolution gaming console may have 4000×2000 pixels
 - 8×10^6 points — 8 million
- Suppose we have $100,000 = 1 \times 10^5$ objects

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
 - For each pair of objects, compute their distance
 - Report minimum distance across all pairs
- There is a clever algorithm that takes some $n \log_2 n$
- High resolution gaming console may have 4000×2000 pixels
 - 8×10^6 points — 8 million
- Suppose we have $100,000 = 1 \times 10^5$ objects
- Naive algorithm takes 10^{10} steps
 - 1000 seconds, or 16.7 minutes in Python
 - Unacceptable response time!

Input size . . .

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
 - For each pair of objects, compute their distance
 - Report minimum distance across all pairs
- There is a clever algorithm that takes time $n \log_2 n$
- High resolution gaming console may have 4000×2000 pixels
 - 8×10^6 points — 8 million
- Suppose we have $100,000 = 1 \times 10^5$ objects
 - Naive algorithm takes 10^{10} steps
 - 1000 seconds, or 16.7 minutes in Python
 - Unacceptable response time!
 - $\log_2 100,000$ is under 20, so $n \log_2 n$ takes a fraction of a second

Orders of magnitude

- When comparing $t(n)$, focus on orders of magnitude
 - Ignore constant factors

Orders of magnitude

- When comparing $t(n)$, focus on orders of magnitude
 - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
 - For small values of n , $f(n) < g(n)$
 - After $n = 5000$, $f(n)$ overtakes $g(n)$

Orders of magnitude

- When comparing $t(n)$, focus on orders of magnitude
 - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
 - For small values of n , $f(n) < g(n)$
 - After $n = 5000$, $f(n)$ overtakes $g(n)$
- Asymptotic complexity
 - What happens in the limit, as n becomes large

Orders of magnitude

- When comparing $t(n)$, focus on orders of magnitude
 - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
 - For small values of n , $f(n) < g(n)$
 - After $n = 5000$, $f(n)$ overtakes $g(n)$
- Asymptotic complexity
 - What happens in the limit, as n becomes large
- Typical growth functions
 - Is $t(n)$ proportional to $\log n, \dots, n^2, n^3, \dots, 2^n$?
 - Note: $\log n$ means $\log_2 n$ by default
 - Logarithmic, polynomial, exponential, ...

Orders of magnitude

Input size	Values of $t(n)$						
	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7	10^{12}			
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}	10^{11}				

Measuring running time

- Analysis should be independent of the underlying hardware
 - Don't use actual time
 - Measure in terms of **basic operations**

Measuring running time

- Analysis should be independent of the underlying hardware
 - Don't use actual time
 - Measure in terms of **basic operations**
- Typical basic operations
 - Compare two values
 - Assign a value to a variable

Measuring running time

- Analysis should be independent of the underlying hardware
 - Don't use actual time
 - Measure in terms of **basic operations**
- Typical basic operations
 - Compare two values
 - Assign a value to a variable
- Exchange a pair of values?

$$\begin{array}{ll} (x,y) = (y,x) & t = x \\ & x = y \\ & y = t \end{array}$$

- If we ignore constants, focus on orders of magnitude, both are within a factor of 3
- Need not be very precise about defining basic operations

What is the input size

- Typically a natural parameter
 - Size of a list/array that we want to search or sort
 - Number of objects we want to rearrange
 - Number of vertices and number edges in a graph
 - We shall see why these are separate parameters

What is the input size

- Typically a natural parameter
 - Size of a list/array that we want to search or sort
 - Number of objects we want to rearrange
 - Number of vertices and number edges in a graph
 - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?

What is the input size

- Typically a natural parameter
 - Size of a list/array that we want to search or sort
 - Number of objects we want to rearrange
 - Number of vertices and number edges in a graph
 - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?
 - Magnitude of n is not the correct measure

What is the input size

- Typically a natural parameter
 - Size of a list/array that we want to search or sort
 - Number of objects we want to rearrange
 - Number of vertices and number edges in a graph
 - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?
 - Magnitude of n is not the correct measure
 - Arithmetic operations are performed digit by digit
 - Addition with carry, subtraction with borrow, multiplication, long division ...

What is the input size

- Typically a natural parameter
 - Size of a list/array that we want to search or sort
 - Number of objects we want to rearrange
 - Number of vertices and number edges in a graph
 - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?
 - Magnitude of n is not the correct measure
 - Arithmetic operations are performed digit by digit
 - Addition with carry, subtraction with borrow, multiplication, long division ...
 - Number of digits is a natural measure of input size
 - Same as $\log_b n$, when we write n in base b

Which inputs should we consider?

- Performance varies across input instances
 - By luck, the value we are searching for is the first element we examine in an array

Which inputs should we consider?

- Performance varies across input instances
 - By luck, the value we are searching for is the first element we examine in an array
- Ideally, want the “average” behaviour
 - Difficult to compute
 - Average over what? Are all inputs equally likely?
 - Need a probability distribution over inputs

Which inputs should we consider?

- Performance varies across input instances
 - By luck, the value we are searching for is the first element we examine in an array
- Ideally, want the “average” behaviour
 - Difficult to compute
 - Average over what? Are all inputs equally likely?
 - Need a probability distribution over inputs
- Instead, **worst case** input
 - Input that forces algorithm to take longest possible time
 - Search for a value that is not present in an unsorted list
 - Must scan all elements

Which inputs should we consider?

- Performance varies across input instances
 - By luck, the value we are searching for is the first element we examine in an array
- Ideally, want the “average” behaviour
 - Difficult to compute
 - Average over what? Are all inputs equally likely?
 - Need a probability distribution over inputs
- Instead, **worst case** input
 - Input that forces algorithm to take longest possible time
 - Search for a value that is not present in an unsorted list
 - Must scan all elements
 - Pessimistic — worst case may be rare

Which inputs should we consider?

- Performance varies across input instances
 - By luck, the value we are searching for is the first element we examine in an array
- Ideally, want the “average” behaviour
 - Difficult to compute
 - Average over what? Are all inputs equally likely?
 - Need a probability distribution over inputs
- Instead, **worst case** input
 - Input that forces algorithm to take longest possible time
 - Search for a value that is not present in an unsorted list
 - Must scan all elements
 - Pessimistic — worst case may be rare
 - Upper bound for worst case **guarantees** good performance

Summary

- Two important parameters when measuring algorithm performance
 - Running time, memory requirement (space)
 - We mainly focus on time

Summary

- Two important parameters when measuring algorithm performance
 - Running time, memory requirement (space)
 - We mainly focus on time
- Running time $t(n)$ is a function of input size n
 - Interested in orders of magnitude
 - Asymptotic complexity, as n becomes large

Summary

- Two important parameters when measuring algorithm performance
 - Running time, memory requirement (space)
 - We mainly focus on time
- Running time $t(n)$ is a function of input size n
 - Interested in orders of magnitude
 - Asymptotic complexity, as n becomes large
- From running time, we can estimate feasible input sizes

Summary

- Two important parameters when measuring algorithm performance
 - Running time, memory requirement (space)
 - We mainly focus on time
- Running time $t(n)$ is a function of input size n
 - Interested in orders of magnitude
 - Asymptotic complexity, as n becomes large
- From running time, we can estimate feasible input sizes
- We focus on worst case inputs
 - Pessimistic, but easier to calculate than average case
 - Upper bound on worst case gives us an overall guarantee on performance

Comparing orders of magnitude

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 2

Orders of magnitude

- When comparing $t(n)$, focus on orders of magnitude
 - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$

Orders of magnitude

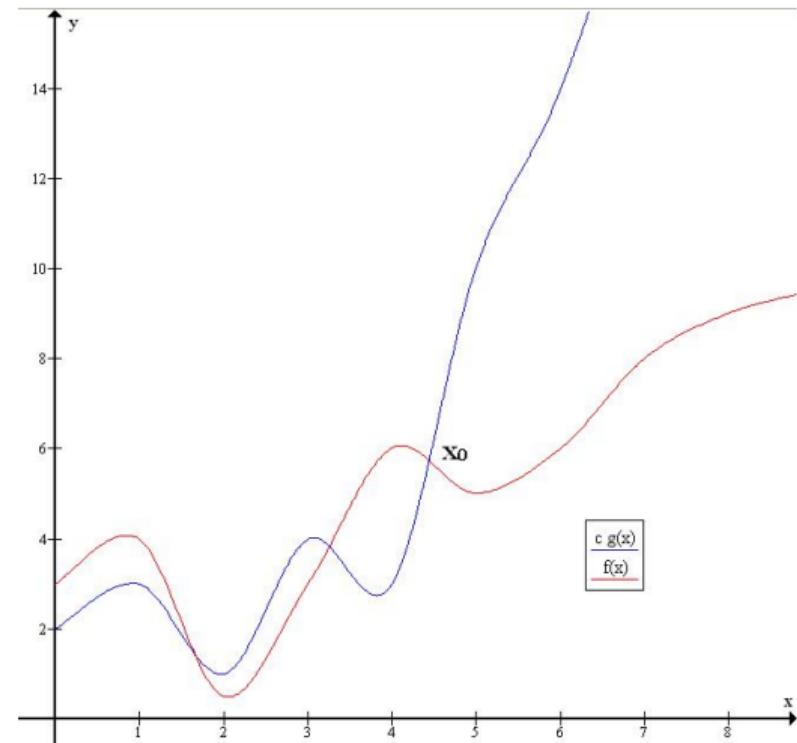
- When comparing $t(n)$, focus on orders of magnitude
 - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
- How do we compare functions with respect to orders of magnitude?

Upper bounds

- $f(x)$ is said to be $O(g(x))$ if we can find constants c and x_0 such that $c \cdot g(x)$ is an upper bound for $f(x)$ for x beyond x_0

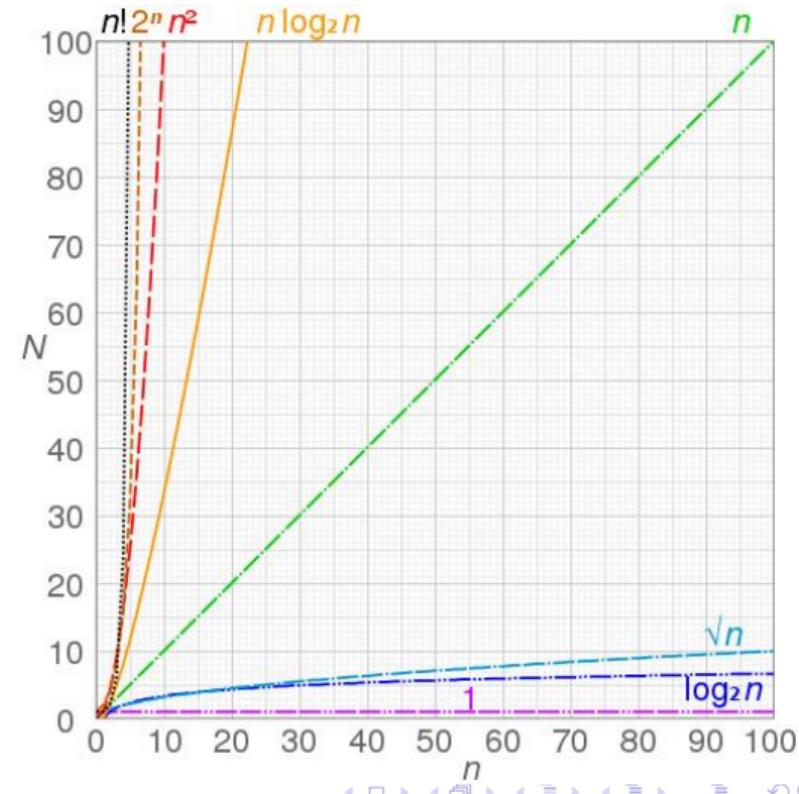
Upper bounds

- $f(x)$ is said to be $O(g(x))$ if we can find constants c and x_0 such that $c \cdot g(x)$ is an upper bound for $f(x)$ for x beyond x_0
- $f(x) \leq cg(x)$ for every $x \geq x_0$



Upper bounds

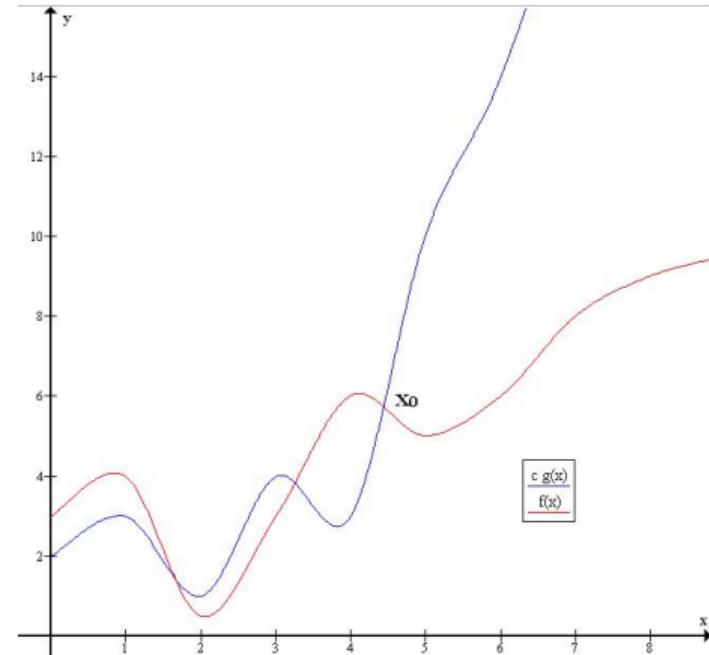
- $f(x)$ is said to be $O(g(x))$ if we can find constants c and x_0 such that $c \cdot g(x)$ is an upper bound for $f(x)$ for x beyond x_0
- $f(x) \leq cg(x)$ for every $x \geq x_0$
- Graphs of typical functions we have seen



Examples

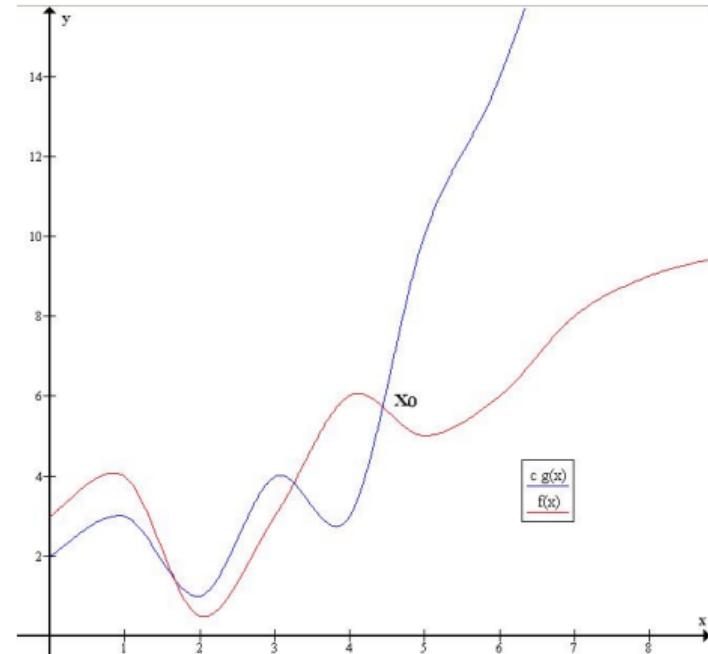
- $100n + 5$ is $O(n^2)$

- $100n + 5 \leq 100n + n = 101n$, for $n \geq 5$
- $101n \leq 101n^2$
- Choose $n_0 = 5$, $c = 101$



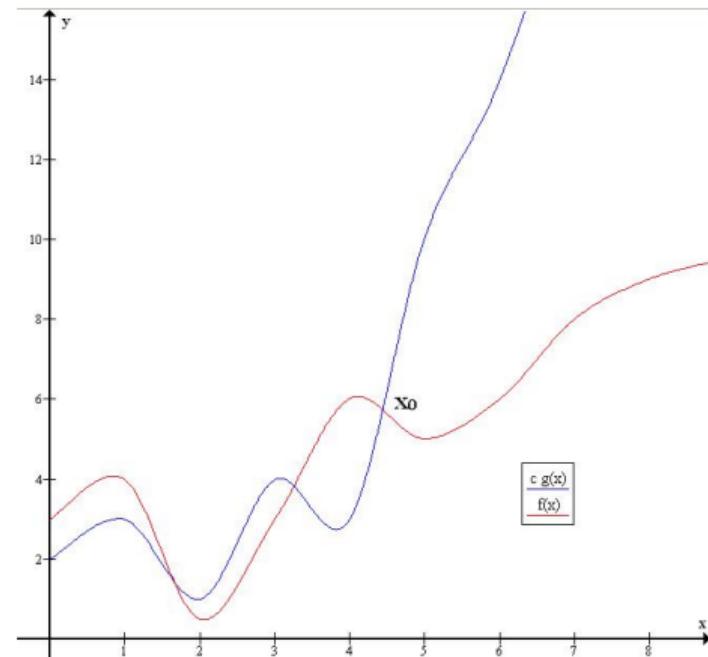
Examples

- $100n + 5$ is $O(n^2)$
 - $100n + 5 \leq 100n + n = 101n$, for $n \geq 5$
 - $101n \leq 101n^2$
 - Choose $n_0 = 5$, $c = 101$
- Alternatively
 - $100n + 5 \leq 100n + 5n = 105n$, for $n \geq 1$
 - $105n \leq 105n^2$
 - Choose $n_0 = 1$, $c = 105$



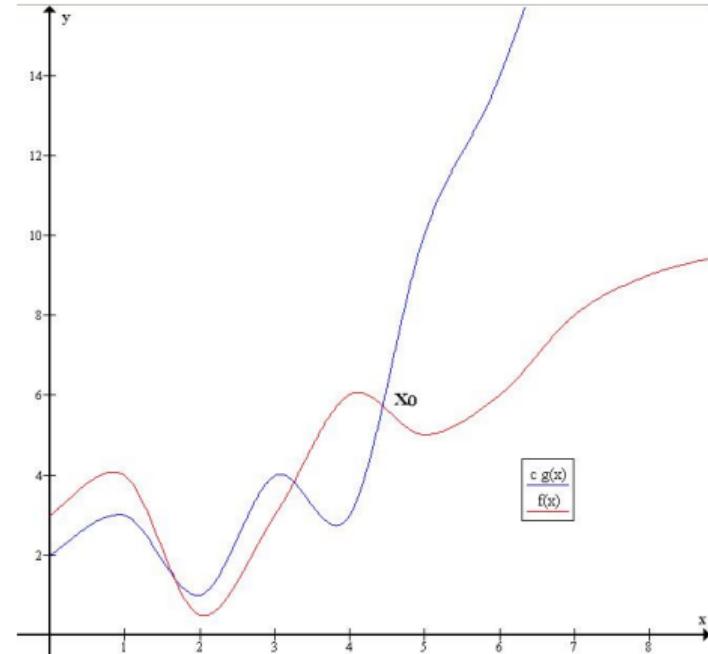
Examples

- $100n + 5$ is $O(n^2)$
 - $100n + 5 \leq 100n + n = 101n$, for $n \geq 5$
 - $101n \leq 101n^2$
 - Choose $n_0 = 5$, $c = 101$
- Alternatively
 - $100n + 5 \leq 100n + 5n = 105n$, for $n \geq 1$
 - $105n \leq 105n^2$
 - Choose $n_0 = 1$, $c = 105$
- Choice of n_0 , c not unique



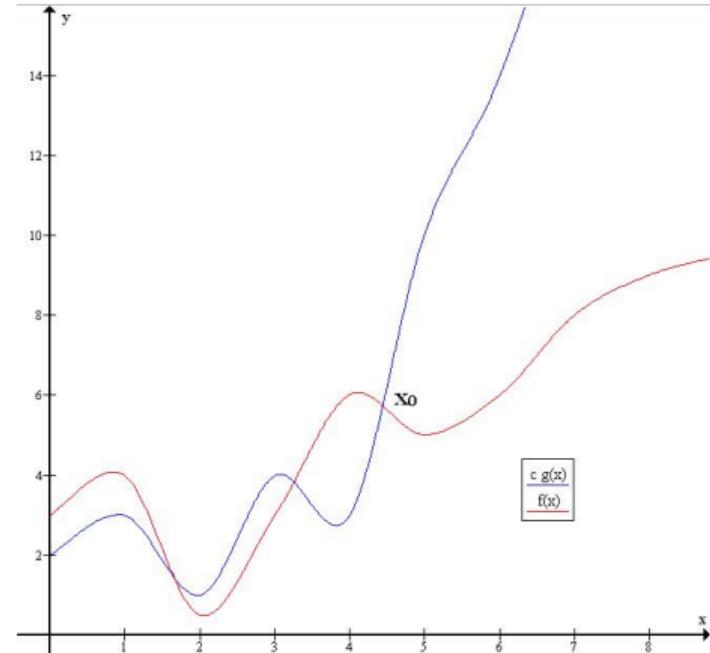
Examples . . .

- $100n^2 + 20n + 5$ is $O(n^2)$
 - $100n^2 + 20n + 5 \leq 100n^2 + 20n^2 + 5n^2$, for $n \geq 1$
 - $100n^2 + 20n + 5 \leq 125n^2$, for $n \geq 1$
 - Choose $n_0 = 1$, $c = 125$



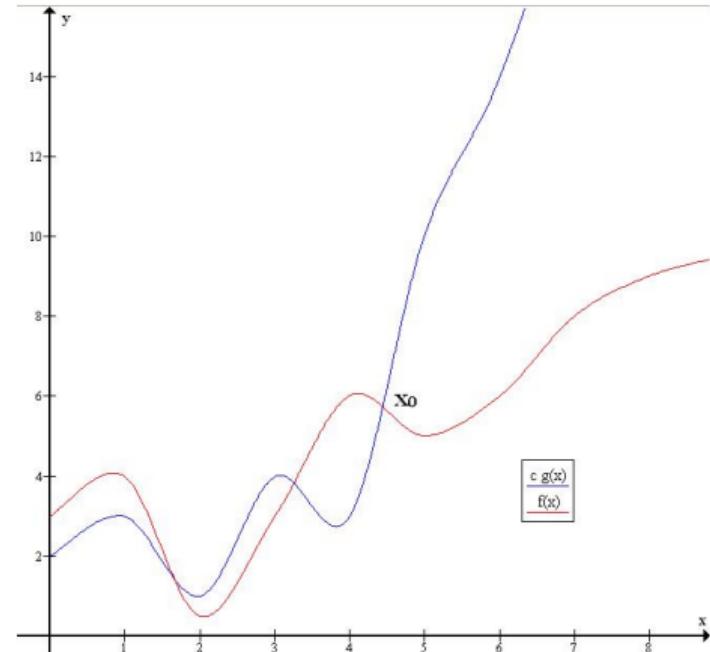
Examples . . .

- $100n^2 + 20n + 5$ is $O(n^2)$
 - $100n^2 + 20n + 5 \leq 100n^2 + 20n^2 + 5n^2$, for $n \geq 1$
 - $100n^2 + 20n + 5 \leq 125n^2$, for $n \geq 1$
 - Choose $n_0 = 1$, $c = 125$
- What matters is the highest term
 - $20n + 5$ is dominated by $100n^2$



Examples . . .

- $100n^2 + 20n + 5$ is $O(n^2)$
 - $100n^2 + 20n + 5 \leq 100n^2 + 20n^2 + 5n^2$, for $n \geq 1$
 - $100n^2 + 20n + 5 \leq 125n^2$, for $n \geq 1$
 - Choose $n_0 = 1$, $c = 125$
- What matters is the highest term
 - $20n + 5$ is dominated by $100n^2$
- n^3 is not $O(n^2)$
 - No matter what c we choose, cn^2 will be dominated by n^3 for $n \geq c$



Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then
 $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
 - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
 - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then
 $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
 - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
 - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
 - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then
 $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
 - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
 - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
 - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
 - For $n \geq n_3$,
$$\begin{aligned}f_1(n) + f_2(n) \\ \leq c_1 g_1(n) + c_2 g_2(n)\end{aligned}$$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then
 $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof

- $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
- $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
- Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
- For $n \geq n_3$,
$$\begin{aligned}f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\&\leq c_3(g_1(n) + g_2(n))\end{aligned}$$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then
 $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof

- $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
- $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
- Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
- For $n \geq n_3$,
$$\begin{aligned}f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\&\leq c_3(g_1(n) + g_2(n)) \\&\leq 2c_3(\max(g_1(n), g_2(n)))\end{aligned}$$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
 - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
 - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
 - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
 - For $n \geq n_3$,
$$\begin{aligned}f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\&\leq c_3(g_1(n) + g_2(n)) \\&\leq 2c_3(\max(g_1(n), g_2(n)))\end{aligned}$$
- Algorithm has two phases
 - Phase A takes time $O(g_A(n))$
 - Phase B takes time $O(g_B(n))$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
 - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
 - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
 - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
 - For $n \geq n_3$,
$$\begin{aligned}f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\&\leq c_3(g_1(n) + g_2(n)) \\&\leq 2c_3(\max(g_1(n), g_2(n)))\end{aligned}$$
- Algorithm has two phases
 - Phase A takes time $O(g_A(n))$
 - Phase B takes time $O(g_B(n))$
- Algorithm as a whole takes time $\max(O(g_A(n), g_B(n)))$

Useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
 - $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$
 - $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$
 - Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
 - For $n \geq n_3$,
$$\begin{aligned}f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\&\leq c_3(g_1(n) + g_2(n)) \\&\leq 2c_3(\max(g_1(n), g_2(n)))\end{aligned}$$
- Algorithm has two phases
 - Phase A takes time $O(g_A(n))$
 - Phase B takes time $O(g_B(n))$
- Algorithm as a whole takes time $\max(O(g_A(n), g_B(n)))$
- Least efficient phase is the upper bound for the whole algorithm

Lower bounds

- $f(x)$ is said to be $\Omega(g(x))$ if we can find constants c and x_0 such that $cg(x)$ is a lower bound for $f(x)$ for x beyond x_0
 - $f(x) \geq cg(x)$ for every $x \geq x_0$

Lower bounds

- $f(x)$ is said to be $\Omega(g(x))$ if we can find constants c and x_0 such that $cg(x)$ is a lower bound for $f(x)$ for x beyond x_0
 - $f(x) \geq cg(x)$ for every $x \geq x_0$
- n^3 is $\Omega(n^2)$
 - $n^3 > n^2$ for all n , so $n_0 = 1, c = 1$

Lower bounds

- $f(x)$ is said to be $\Omega(g(x))$ if we can find constants c and x_0 such that $cg(x)$ is a lower bound for $f(x)$ for x beyond x_0
 - $f(x) \geq cg(x)$ for every $x \geq x_0$
- n^3 is $\Omega(n^2)$
 - $n^3 > n^2$ for all n , so $n_0 = 1, c = 1$
- Typically we establish lower bounds for a problem rather than an individual algorithm
 - If we sort a list by comparing elements and swapping them, we require $\Omega(n \log n)$ comparisons
 - This is **independent** of the algorithm we use for sorting

Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
 - Find constants c_1, c_2, x_0 such that $c_1g(x) \leq f(x) \leq c_2g(x)$ for every $x \geq x_0$

Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
 - Find constants c_1, c_2, x_0 such that $c_1g(x) \leq f(x) \leq c_2g(x)$ for every $x \geq x_0$
- $n(n - 1)/2$ is $\Theta(n^2)$

Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
 - Find constants c_1, c_2, x_0 such that $c_1g(x) \leq f(x) \leq c_2g(x)$ for every $x \geq x_0$
- $n(n - 1)/2$ is $\Theta(n^2)$
 - Upper bound
 - $n(n - 1)/2 = n^2/2 - n/2 \leq n^2/2$ for all $n \geq 0$

Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
 - Find constants c_1, c_2, x_0 such that $c_1g(x) \leq f(x) \leq c_2g(x)$ for every $x \geq x_0$
- $n(n - 1)/2$ is $\Theta(n^2)$
 - Upper bound
 - $n(n - 1)/2 = n^2/2 - n/2 \leq n^2/2$ for all $n \geq 0$
 - Lower bound
 - $n(n - 1)/2 = n^2/2 - n/2 \geq n^2/2 - (n/2 \times n/2) \geq n^2/4$ for $n \geq 2$

Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
 - Find constants c_1, c_2, x_0 such that $c_1g(x) \leq f(x) \leq c_2g(x)$ for every $x \geq x_0$
- $n(n - 1)/2$ is $\Theta(n^2)$
 - Upper bound
 - $n(n - 1)/2 = n^2/2 - n/2 \leq n^2/2$ for all $n \geq 0$
 - Lower bound
 - $n(n - 1)/2 = n^2/2 - n/2 \geq n^2/2 - (n/2 \times n/2) \geq n^2/4$ for $n \geq 2$
 - Choose $n_0 = 2, c_1 = 1/4, c_2 = 1/2$

Summary

- $f(n)$ is $O(g(n))$ means $g(n)$ is an upper bound for $f(n)$
 - Useful to describe asymptotic worst case running time

Summary

- $f(n)$ is $O(g(n))$ means $g(n)$ is an upper bound for $f(n)$
 - Useful to describe asymptotic worst case running time
- $f(n)$ is $\Omega(g(n))$ means $g(n)$ is a lower bound for $f(n)$
 - Typically used for a problem as a whole, rather than an individual algorithm

Summary

- $f(n)$ is $O(g(n))$ means $g(n)$ is an upper bound for $f(n)$
 - Useful to describe asymptotic worst case running time
- $f(n)$ is $\Omega(g(n))$ means $g(n)$ is a lower bound for $f(n)$
 - Typically used for a problem as a whole, rather than an individual algorithm
- $f(n)$ is $\Theta(g(n))$: matching upper and lower bounds
 - We have found an optimal algorithm for a problem

Calculating complexity — Examples

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 2

Calculating complexity

- Iterative programs
- Recursive programs

Example 1

Find the maximum element in a list

- Input size is length of the list
- Single loop scans all elements
- Always takes n steps
- Overall time is $O(n)$

```
def maxElement(L):  
    maxval = L[0]  
    for i in range(len(L)):  
        if L[i] > maxval:  
            maxval = L[i]  
    return(maxval)
```

Example 2

Check whether a list contains duplicates

- Input size is length of the list
- Nested loop scans all pairs of elements
- A duplicate may be found in the very first iteration
- Worst case — no duplicates, both loops run fully
- Time is $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2$
- Overall time is $O(n^2)$

```
def noDuplicates(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[i] == L[j]:  
                return(False)  
    return(True)
```

Example 3

Matrix multiplication

- Matrix is represented as list of lists
 - $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
 - $[[1,2,3], [4,5,6]]$
- Input matrices have size $m \times n$, $n \times p$
- Output matrix is $m \times p$
- Three nested loops
- Overall time is $O(mnp) — O(n^3)$ if both are $n \times n$

```
def matrixMultiply(A,B):
    (m,n,p) = (len(A),len(B),len(B[0]))

    C = [[ 0 for i in range(p) ]
          for j in range(m) ]

    for i in range(m):
        for j in range(p):
            for k in range(n):
                C[i][j] = C[i][j] + A[i][k]*B[k][j]

    return(C)
```

Example 4

Number of bits in binary representation of n

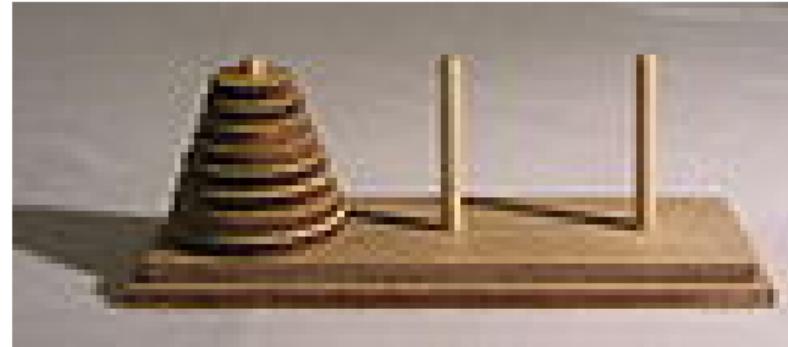
- $\log n$ steps for n to reach 1
- For number theoretic problems, input size is number of digits
- This algorithm is linear in input size

```
def numberOfBits(n):  
    count = 1  
  
    while n > 1:  
        count = count + 1  
        n = n // 2  
  
    return(count)
```

Example 5

Towers of Hanoi

- Three pegs A,B,C
- Move n disks from A to B, use C as transit peg
- Never put a larger disk on a smaller one



Example 5

Towers of Hanoi

- Three pegs A,B,C
- Move n disks from A to B, use C as transit peg
- Never put a larger disk on a smaller one



Recursive solution

- Move $n - 1$ disks from A to C, use B as transit peg
- Move largest disk from A to B
- Move $n - 1$ disks from C to B, use A as transit peg

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n - 1) + 1 + M(n - 1) = 2M(n - 1) + 1$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n - 1) + 1 + M(n - 1) = 2M(n - 1) + 1$

Unwind and solve

$$M(n) = 2M(n - 1) + 1$$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n - 1) + 1 + M(n - 1) = 2M(n - 1) + 1$

Unwind and solve

$$\begin{aligned}M(n) &= 2M(n - 1) + 1 \\&= 2(2M(n - 2) + 1) + 1 = 2^2M(n - 2) + (2 + 1)\end{aligned}$$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n - 1) + 1 + M(n - 1) = 2M(n - 1) + 1$

Unwind and solve

$$\begin{aligned}M(n) &= 2M(n - 1) + 1 \\&= 2(2M(n - 2) + 1) + 1 = 2^2M(n - 2) + (2 + 1) \\&= 2^2(2M(n - 3) + 1) + (2 + 1) = 2^3M(n - 3) + (4 + 2 + 1)\end{aligned}$$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n - 1) + 1 + M(n - 1) = 2M(n - 1) + 1$

Unwind and solve

$$\begin{aligned}M(n) &= 2M(n - 1) + 1 \\&= 2(2M(n - 2) + 1) + 1 = 2^2M(n - 2) + (2 + 1) \\&= 2^2(2M(n - 3) + 1) + (2 + 1) = 2^3M(n - 3) + (4 + 2 + 1) \\&\quad \dots \\&= 2^kM(n - k) + (2^k - 1))\end{aligned}$$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n - 1) + 1 + M(n - 1) = 2M(n - 1) + 1$

Unwind and solve

$$\begin{aligned}M(n) &= 2M(n - 1) + 1 \\&= 2(2M(n - 2) + 1) + 1 = 2^2M(n - 2) + (2 + 1) \\&= 2^2(2M(n - 3) + 1) + (2 + 1) = 2^3M(n - 3) + (4 + 2 + 1) \\&\quad \dots \\&= 2^kM(n - k) + (2^k - 1)) \\&\quad \dots \\&= 2^{n-1}M(1) + (2^{n-1} - 1)\end{aligned}$$

Example 5

Recurrence

- $M(n)$ — number of moves to transfer n disks
- $M(1) = 1$
- $M(n) = M(n - 1) + 1 + M(n - 1) = 2M(n - 1) + 1$

Unwind and solve

$$\begin{aligned}M(n) &= 2M(n - 1) + 1 \\&= 2(2M(n - 2) + 1) + 1 = 2^2M(n - 2) + (2 + 1) \\&= 2^2(2M(n - 3) + 1) + (2 + 1) = 2^3M(n - 3) + (4 + 2 + 1) \\&\quad \dots \\&= 2^kM(n - k) + (2^k - 1)) \\&\quad \dots \\&= 2^{n-1}M(1) + (2^{n-1} - 1) \\&= 2^{n-1} + 2^{n-1} - 1 = 2^n - 1\end{aligned}$$

Summary

- Iterative programs
 - Focus on loops

Summary

- Iterative programs
 - Focus on loops
- Recursive programs
 - Write and solve a recurrence

Summary

- Iterative programs
 - Focus on loops
- Recursive programs
 - Write and solve a recurrence
- Need to be clear about accounting for “basic” operations

Searching in a List

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 2

Search problem

- Is value `v` present in list `l`?

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list

```
def naivesearch(v,l):  
    for x in l:  
        if v == x:  
            return(True)  
    return(False)
```

Search problem

- Is value v present in list l ?
- Naive solution scans the list
- Input size n , the length of the list

```
def naivesearch(v,l):  
    for x in l:  
        if v == x:  
            return(True)  
    return(False)
```

Search problem

- Is value v present in list l ?
- Naive solution scans the list
- Input size n , the length of the list
- Worst case is when v is not present in l

```
def naivesearch(v,l):  
    for x in l:  
        if v == x:  
            return(True)  
    return(False)
```

Search problem

- Is value v present in list l ?
- Naive solution scans the list
- Input size n , the length of the list
- Worst case is when v is not present in l
- Worst case complexity is $O(n)$

```
def naivesearch(v,l):  
    for x in l:  
        if v == x:  
            return(True)  
    return(False)
```

Searching a sorted list

- What if `l` is sorted in ascending order?

Searching a sorted list

- What if `l` is sorted in ascending order?
- Compare `v` with the midpoint of `l`

Searching a sorted list

- What if `l` is sorted in ascending order?
- Compare `v` with the midpoint of `l`
 - If midpoint is `v`, the value is found
 - If `v` less than midpoint, search the first half
 - If `v` greater than midpoint, search the second half
 - Stop when the interval to search becomes empty

```
def binarysearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(binarysearch(v,l[:m]))  
    else:  
        return(binarysearch(v,l[m+1:])))
```

Searching a sorted list

- What if `l` is sorted in ascending order?
- Compare `v` with the midpoint of `l`
 - If midpoint is `v`, the value is found
 - If `v` less than midpoint, search the first half
 - If `v` greater than midpoint, search the second half
 - Stop when the interval to search becomes empty
- Binary search

```
def binarysearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(binarysearch(v,l[:m]))  
    else:  
        return(binarysearch(v,l[m+1:])))
```

Binary search

- How long does this take?

```
def binarysearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(binarysearch(v,l[:m]))  
    else:  
        return(binarysearch(v,l[m+1:])))
```

Binary search

- How long does this take?
 - Each call halves the interval to search
 - Stop when the interval become empty
- $\log n$ — number of times to divide n by 2 to reach 1
 - $1 // 2 = 0$, so next call reaches empty interval

```
def binarysearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(binarysearch(v,l[:m]))  
    else:  
        return(binarysearch(v,l[m+1:])))
```

Binary search

- How long does this take?
 - Each call halves the interval to search
 - Stop when the interval become empty
- $\log n$ — number of times to divide n by 2 to reach 1
 - $1 // 2 = 0$, so next call reaches empty interval
- $O(\log n)$ steps

```
def binarysearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(binarysearch(v,l[:m]))  
    else:  
        return(binarysearch(v,l[m+1:])))
```

Alternative calculation

- $T(n)$: the time to search a list of length n

- If $n = 0$, we exit, so $T(n) = 1$
- If $n > 0$, $T(n) = T(n // 2) + 1$

```
def bsearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(bsearch(v,l[:m]))  
    else:  
        return(bsearch(v,l[m+1:]))


```

Alternative calculation

- $T(n)$: the time to search a list of length n

- If $n = 0$, we exit, so $T(n) = 1$
 - If $n > 0$, $T(n) = T(n // 2) + 1$

- Recurrence for $T(n)$

- $T(0) = 1$
 - $T(n) = T(n // 2) + 1, n > 0$

```
def bsearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(bsearch(v,l[:m]))  
    else:  
        return(bsearch(v,l[m+1:]))


```

Alternative calculation

- $T(n)$: the time to search a list of length n

- If $n = 0$, we exit, so $T(n) = 1$
 - If $n > 0$, $T(n) = T(n // 2) + 1$

- Recurrence for $T(n)$

- $T(0) = 1$
 - $T(n) = T(n // 2) + 1, n > 0$

- Solve by “unwinding”

```
def bsearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(bsearch(v,l[:m]))  
    else:  
        return(bsearch(v,l[m+1:]))


```

Alternative calculation

- $T(n)$: the time to search a list of length n

- If $n = 0$, we exit, so $T(n) = 1$
 - If $n > 0$, $T(n) = T(n // 2) + 1$

- Recurrence for $T(n)$

- $T(0) = 1$
 - $T(n) = T(n // 2) + 1, n > 0$

- Solve by “unwinding”

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

```
def bsearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(bsearch(v,l[:m]))  
    else:  
        return(bsearch(v,l[m+1:])))
```

Alternative calculation

- $T(n)$: the time to search a list of length n

- If $n = 0$, we exit, so $T(n) = 1$
 - If $n > 0$, $T(n) = T(n // 2) + 1$

- Recurrence for $T(n)$

- $T(0) = 1$
 - $T(n) = T(n // 2) + 1, n > 0$

- Solve by “unwinding”

- $$\begin{aligned}T(n) &= T(n // 2) + 1 \\&= (T(n // 4) + 1) + 1\end{aligned}$$

```
def bsearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(bsearch(v,l[:m]))  
    else:  
        return(bsearch(v,l[m+1:]))


```

Alternative calculation

- $T(n)$: the time to search a list of length n
 - If $n = 0$, we exit, so $T(n) = 1$
 - If $n > 0$, $T(n) = T(n // 2) + 1$
- Recurrence for $T(n)$
 - $T(0) = 1$
 - $T(n) = T(n // 2) + 1, n > 0$
- Solve by “unwinding”
- $$\begin{aligned}T(n) &= T(n // 2) + 1 \\&= (T(n // 4) + 1) + 1 = T(n // 2^2) + \underbrace{1+1}_{2}\end{aligned}$$

```
def bsearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(bsearch(v,l[:m]))  
    else:  
        return(bsearch(v,l[m+1:]))


```

Alternative calculation

- $T(n)$: the time to search a list of length n

- If $n = 0$, we exit, so $T(n) = 1$
 - If $n > 0$, $T(n) = T(n // 2) + 1$

- Recurrence for $T(n)$

- $T(0) = 1$
 - $T(n) = T(n // 2) + 1, n > 0$

- Solve by “unwinding”

- $$\begin{aligned}T(n) &= T(n // 2) + 1 \\&= (T(n // 4) + 1) + 1 = T(n // 2^2) + \underbrace{1 + 1}_2 \\&= \dots \\&= T(n // 2^k) + \underbrace{1 + \dots + 1}_k\end{aligned}$$

```
def bsearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(bsearch(v,l[:m]))  
    else:  
        return(bsearch(v,l[m+1:])))
```

Alternative calculation

- $T(n)$: the time to search a list of length n

- If $n = 0$, we exit, so $T(n) = 1$
 - If $n > 0$, $T(n) = T(n // 2) + 1$

- Recurrence for $T(n)$

- $T(0) = 1$
 - $T(n) = T(n // 2) + 1, n > 0$

- Solve by “unwinding”

$$\begin{aligned} T(n) &= T(n // 2) + 1 \\ &= (T(n // 4) + 1) + 1 = T(n // 2^2) + \underbrace{1 + 1}_2 \\ &= \dots \\ &= T(n // 2^k) + \underbrace{1 + \dots + 1}_k \\ &= T(1) + k, \text{ for } k = \log n \end{aligned}$$

```
def bsearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(bsearch(v,l[:m]))  
    else:  
        return(bsearch(v,l[m+1:]))

    < > << >> <<< >>> <<<< >>>> <<<<< >>>>>
```

Alternative calculation

- $T(n)$: the time to search a list of length n

- If $n = 0$, we exit, so $T(n) = 1$
 - If $n > 0$, $T(n) = T(n // 2) + 1$

- Recurrence for $T(n)$

- $T(0) = 1$
 - $T(n) = T(n // 2) + 1, n > 0$

- Solve by “unwinding”

$$\begin{aligned} T(n) &= T(n // 2) + 1 \\ &= (T(n // 4) + 1) + 1 = T(n // 2^2) + \underbrace{1 + 1}_2 \\ &= \dots \\ &= T(n // 2^k) + \underbrace{1 + \dots + 1}_k \\ &= T(1) + k, \text{ for } k = \log n \\ &= (T(0) + 1) + \log n = 2 + \log n \end{aligned}$$

```
def bsearch(v,l):  
    if l == []:  
        return(False)  
  
    m = len(l)//2  
  
    if v == l[m]:  
        return(True)  
  
    if v < l[m]:  
        return(bsearch(v,l[:m]))  
    else:  
        return(bsearch(v,l[m+1:]))

    < > << >> <<< >>> <<<< >>>> <<<<< >>>>>
```

Summary

- Search in an unsorted list takes time $O(n)$
 - Need to scan the entire list
 - Worst case is when the value is not present in the list

Summary

- Search in an unsorted list takes time $O(n)$
 - Need to scan the entire list
 - Worst case is when the value is not present in the list
- For a sorted list, binary search takes time $O(\log n)$
 - Halve the interval to search each time

Summary

- Search in an unsorted list takes time $O(n)$
 - Need to scan the entire list
 - Worst case is when the value is not present in the list
- For a sorted list, binary search takes time $O(\log n)$
 - Halve the interval to search each time
- In a sorted list, we can determine that v is absent by examining just $\log n$ values!

Selection Sort

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 2

Sorting a list

- Sorting a list makes many other computations easier
 - Binary search
 - Finding the median
 - Checking for duplicates
 - Building a frequency table of values

Sorting a list

- Sorting a list makes many other computations easier
 - Binary search
 - Finding the median
 - Checking for duplicates
 - Building a frequency table of values
- How do we sort a list?

Sorting a list

- Sorting a list makes many other computations easier
 - Binary search
 - Finding the median
 - Checking for duplicates
 - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
 - Instructor has a pile of evaluated exam papers
 - Papers in random order of marks
 - Your task is to arrange the papers in descending order of marks

Sorting a list

- Sorting a list makes many other computations easier
 - Binary search
 - Finding the median
 - Checking for duplicates
 - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
 - Instructor has a pile of evaluated exam papers
 - Papers in random order of marks
 - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks

Sorting a list

- Sorting a list makes many other computations easier
 - Binary search
 - Finding the median
 - Checking for duplicates
 - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
 - Instructor has a pile of evaluated exam papers
 - Papers in random order of marks
 - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile

Sorting a list

- Sorting a list makes many other computations easier
 - Binary search
 - Finding the median
 - Checking for duplicates
 - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
 - Instructor has a pile of evaluated exam papers
 - Papers in random order of marks
 - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile
- Repeat with the remaining papers
 - Add the paper with next minimum marks to the second pile each time

Sorting a list

- Sorting a list makes many other computations easier
 - Binary search
 - Finding the median
 - Checking for duplicates
 - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
 - Instructor has a pile of evaluated exam papers
 - Papers in random order of marks
 - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile
- Repeat with the remaining papers
 - Add the paper with next minimum marks to the second pile each time
- Eventually, the new pile is sorted in descending order

Sorting a list

74 32 89 55 21 64

Sorting a list

74

32

89

55

~~21~~

64

21

Sorting a list

74

32

89

55

21

64

21

32

Sorting a list

74 32 89 55 21 64

21 32 55

Sorting a list

74 32 89 55 21 **64**

21 32 55 64

Sorting a list

74

32

89

55

21

64

21

32

55

64

74

Sorting a list

74

32

89

55

21

64

21

32

55

64

74

89

Selection sort

- Select the next element in sorted order

Selection sort

- Select the next element in sorted order
- Append it to the final sorted list

Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
 - Swap the minimum element into the first position
 - Swap the second minimum element into the second position
 - ...

Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
 - Swap the minimum element into the first position
 - Swap the second minimum element into the second position
 - ...
- Eventually the list is rearranged in place in ascending order

Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
 - Swap the minimum element into the first position
 - Swap the second minimum element into the second position
 - ...
- Eventually the list is rearranged in place in ascending order

```
def SelectionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        mpos = i  
        # mpos: position of minimum in L[i:]  
        for j in range(i+1,n):  
            if L[j] < L[mpos]:  
                mpos = j  
        # L[mpos] : smallest value in L[i:]  
        # Exchange L[mpos] and L[i]  
        (L[i],L[mpos]) = (L[mpos],L[i])  
        # Now L[:i+1] is sorted  
    return(L)
```

Analysis of selection sort

- Correctness follows from the invariant

```
def SelectionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        mpos = i
        # mpos: position of minimum in L[i:]
        for j in range(i+1,n):
            if L[j] < L[mpos]:
                mpos = j
        # L[mpos] : smallest value in L[i:]
        # Exchange L[mpos] and L[i]
        (L[i],L[mpos]) = (L[mpos],L[i])
        # Now L[:i+1] is sorted
    return(L)
```

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency

```
def SelectionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        mpos = i  
        # mpos: position of minimum in L[i:]  
        for j in range(i+1,n):  
            if L[j] < L[mpos]:  
                mpos = j  
        # L[mpos] : smallest value in L[i:]  
        # Exchange L[mpos] and L[i]  
        (L[i],L[mpos]) = (L[mpos],L[i])  
        # Now L[:i+1] is sorted  
    return(L)
```

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
 - Outer loop iterates n times

```
def SelectionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        mpos = i  
        # mpos: position of minimum in L[i:]  
        for j in range(i+1,n):  
            if L[j] < L[mpos]:  
                mpos = j  
        # L[mpos] : smallest value in L[i:]  
        # Exchange L[mpos] and L[i]  
        (L[i],L[mpos]) = (L[mpos],L[i])  
        # Now L[:i+1] is sorted  
    return(L)
```

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
 - Outer loop iterates n times
 - Inner loop: $n - i$ steps to find minimum in $L[i:]$

```
def SelectionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        mpos = i  
        # mpos: position of minimum in L[i:]  
        for j in range(i+1,n):  
            if L[j] < L[mpos]:  
                mpos = j  
        # L[mpos] : smallest value in L[i:]  
        # Exchange L[mpos] and L[i]  
        (L[i],L[mpos]) = (L[mpos],L[i])  
        # Now L[:i+1] is sorted  
    return(L)
```

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
 - Outer loop iterates n times
 - Inner loop: $n - i$ steps to find minimum in $L[i:]$
 - $T(n) = n + (n - 1) + \dots + 1$

```
def SelectionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        mpos = i  
        # mpos: position of minimum in L[i:]  
        for j in range(i+1,n):  
            if L[j] < L[mpos]:  
                mpos = j  
        # L[mpos] : smallest value in L[i:]  
        # Exchange L[mpos] and L[i]  
        (L[i],L[mpos]) = (L[mpos],L[i])  
        # Now L[:i+1] is sorted  
    return(L)
```

Analysis of selection sort

- Correctness follows from the invariant

- Efficiency

- Outer loop iterates n times
- Inner loop: $n - i$ steps to find minimum in $L[i:]$
- $T(n) = n + (n - 1) + \dots + 1$
- $T(n) = n(n + 1)/2$

```
def SelectionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        mpos = i  
        # mpos: position of minimum in L[i:]  
        for j in range(i+1,n):  
            if L[j] < L[mpos]:  
                mpos = j  
        # L[mpos] : smallest value in L[i:]  
        # Exchange L[mpos] and L[i]  
        (L[i],L[mpos]) = (L[mpos],L[i])  
        # Now L[:i+1] is sorted  
    return(L)
```

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
 - Outer loop iterates n times
 - Inner loop: $n - i$ steps to find minimum in $L[i:]$
 - $T(n) = n + (n - 1) + \dots + 1$
 - $T(n) = n(n + 1)/2$
- $T(n)$ is $O(n^2)$

```
def SelectionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        mpos = i  
        # mpos: position of minimum in L[i:]  
        for j in range(i+1,n):  
            if L[j] < L[mpos]:  
                mpos = j  
        # L[mpos] : smallest value in L[i:]  
        # Exchange L[mpos] and L[i]  
        (L[i],L[mpos]) = (L[mpos],L[i])  
        # Now L[:i+1] is sorted  
    return(L)
```

Summary

- Selection sort is an intuitive algorithm to sort a list

Summary

- Selection sort is an intuitive algorithm to sort a list
- Repeatedly find the minimum (or maximum) and append to sorted list

Summary

- Selection sort is an intuitive algorithm to sort a list
- Repeatedly find the minimum (or maximum) and append to sorted list
- Worst case complexity is $O(n^2)$
 - Every input takes this much time
 - No advantage even if list is arranged carefully before sorting

Insertion Sort

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 2

Sorting a list

- You are the TA for a course
 - Instructor has a pile of evaluated exam papers
 - Papers in random order of marks
 - Your task is to arrange the papers in descending order of marks

Sorting a list

- You are the TA for a course

Strategy 2

- Instructor has a pile of evaluated exam papers
- Papers in random order of marks
- Your task is to arrange the papers in descending order of marks

Sorting a list

- You are the TA for a course
 - Instructor has a pile of evaluated exam papers
 - Papers in random order of marks
 - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile

Sorting a list

- You are the TA for a course
 - Instructor has a pile of evaluated exam papers
 - Papers in random order of marks
 - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile
- Second paper
 - Lower marks than first paper? Place below first paper in new pile
 - Higher marks than first paper? Place above first paper in new pile

Sorting a list

- You are the TA for a course
 - Instructor has a pile of evaluated exam papers
 - Papers in random order of marks
 - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile
- Second paper
 - Lower marks than first paper? Place below first paper in new pile
 - Higher marks than first paper? Place above first paper in new pile
- Third paper
 - **Insert** into correct position with respect to first two

Sorting a list

- You are the TA for a course
 - Instructor has a pile of evaluated exam papers
 - Papers in random order of marks
 - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile
- Second paper
 - Lower marks than first paper? Place below first paper in new pile
 - Higher marks than first paper? Place above first paper in new pile
- Third paper
 - **Insert** into correct position with respect to first two
- Do this for the remaining papers
 - **Insert** each one into correct position in the second pile

Sorting a list

74 32 89 55 21 64

Sorting a list

74

32

89

55

21

64

74

Sorting a list

74

~~32~~

89

55

21

64

32

74

Sorting a list

74 32 89 55 21 64

32 74 89

Sorting a list

74 32 89 55 21 64

32 55 74 89

Sorting a list

74

32

89

55

21

64

21

32

55

74

89

Sorting a list

74

32

89

55

21

64

21

32

55

64

74

89

Insertion sort

- Start building a new sorted list

Insertion sort

- Start building a new sorted list
- Pick next element and **insert** it into the sorted list

Insertion sort

- Start building a new sorted list
- Pick next element and **insert** it into the sorted list
- An iterative formulation
 - Assume $L[:i]$ is sorted
 - Insert $L[i]$ in $L[:i]$

Insertion sort

- Start building a new sorted list
- Pick next element and **insert** it into the sorted list
- An iterative formulation
 - Assume $L[:i]$ is sorted
 - Insert $L[i]$ in $L[:i]$

```
def InsertionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        # Move L[i] to correct position in L
        j = i
        while(j > 0 and L[j] < L[j-1]):
            (L[j],L[j-1]) = (L[j-1],L[j])
            j = j-1
        # Now L[:i+1] is sorted
    return(L)
```

Insertion sort

- Start building a new sorted list
- Pick next element and **insert** it into the sorted list
- An iterative formulation
 - Assume $L[:i]$ is sorted
 - Insert $L[i]$ in $L[:i]$
- A recursive formulation
 - Inductively sort $L[:i]$
 - Insert $L[i]$ in $L[:i]$

```
def InsertionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        # Move L[i] to correct position in L  
        j = i  
        while(j > 0 and L[j] < L[j-1]):  
            (L[j],L[j-1]) = (L[j-1],L[j])  
            j = j-1  
        # Now L[:i+1] is sorted  
    return(L)
```

Insertion sort

- Start building a new sorted list
- Pick next element and **insert** it into the sorted list
- An iterative formulation
 - Assume $L[:i]$ is sorted
 - Insert $L[i]$ in $L[:i]$
- A recursive formulation
 - Inductively sort $L[:i]$
 - Insert $L[i]$ in $L[:i]$

```
def Insert(L,v):  
    n = len(L)  
    if n == 0:  
        return([v])  
    if v >= L[-1]:  
        return(L+[v])  
    else:  
        return(Insert(L[:-1],v)+L[-1:])
```

```
def ISort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    L = Insert(ISort(L[:-1]),L[-1])  
    return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant

```
def InsertionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        # Move L[i] to correct position in L
        j = i
        while(L[j] < L[j-1]):
            (L[j],L[j-1]) = (L[j-1],L[j])
            j = j-1
        # Now L[:i+1] is sorted
    return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency

```
def InsertionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        # Move L[i] to correct position in L
        j = i
        while(L[j] < L[j-1]):
            (L[j],L[j-1]) = (L[j-1],L[j])
            j = j-1
        # Now L[:i+1] is sorted
    return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
 - Outer loop iterates n times

```
def InsertionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        # Move L[i] to correct position in L
        j = i
        while(L[j] < L[j-1]):
            (L[j],L[j-1]) = (L[j-1],L[j])
            j = j-1
        # Now L[:i+1] is sorted
    return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
 - Outer loop iterates n times
 - Inner loop: i steps to insert $L[i]$ in $L[:i]$

```
def InsertionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        # Move L[i] to correct position in L  
        j = i  
        while(L[j] < L[j-1]):  
            (L[j],L[j-1]) = (L[j-1],L[j])  
            j = j-1  
        # Now L[:i+1] is sorted  
    return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
 - Outer loop iterates n times
 - Inner loop: i steps to insert $L[i]$ in $L[:i]$
 - $T(n) = 0 + 1 + \dots + (n - 1)$

```
def InsertionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        # Move L[i] to correct position in L  
        j = i  
        while(L[j] < L[j-1]):  
            (L[j],L[j-1]) = (L[j-1],L[j])  
            j = j-1  
        # Now L[:i+1] is sorted  
    return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
 - Outer loop iterates n times
 - Inner loop: i steps to insert $L[i]$ in $L[:i]$
 - $T(n) = 0 + 1 + \dots + (n - 1)$
 - $T(n) = n(n - 1)/2$

```
def InsertionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        # Move L[i] to correct position in L  
        j = i  
        while(L[j] < L[j-1]):  
            (L[j],L[j-1]) = (L[j-1],L[j])  
            j = j-1  
        # Now L[:i+1] is sorted  
    return(L)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
 - Outer loop iterates n times
 - Inner loop: i steps to insert $L[i]$ in $L[:i]$
 - $T(n) = 0 + 1 + \dots + (n - 1)$
 - $T(n) = n(n - 1)/2$
- $T(n)$ is $O(n^2)$

```
def InsertionSort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    for i in range(n):  
        # Assume L[:i] is sorted  
        # Move L[i] to correct position in L  
        j = i  
        while(L[j] < L[j-1]):  
            (L[j],L[j-1]) = (L[j-1],L[j])  
            j = j-1  
        # Now L[:i+1] is sorted  
    return(L)
```

Analysis of recursive insertion sort

- For input of size n , let

- $TI(n)$ be the time taken by `Insert`
- $TS(n)$ be the time taken by `ISort`

```
def Insert(L,v):  
    n = len(L)  
    if n == 0:  
        return([v])  
    if v >= L[-1]:  
        return(L+[v])  
    else:  
        return(Insert(L[:-1],v)+L[-1:])
```

```
def ISort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    L = Insert(ISort(L[:-1]),L[-1])  
    return(L)
```

Analysis of recursive insertion sort

- For input of size n , let
 - $TI(n)$ be the time taken by `Insert`
 - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
 - $TI(0) = 1$
 - $TI(n) = TI(n - 1) + 1$

```
def Insert(L,v):  
    n = len(L)  
    if n == 0:  
        return([v])  
    if v >= L[-1]:  
        return(L+[v])  
    else:  
        return(Insert(L[:-1],v)+L[-1:])
```

```
def ISort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    L = Insert(ISort(L[:-1]),L[-1])  
    return(L)
```

Analysis of recursive insertion sort

- For input of size n , let
 - $TI(n)$ be the time taken by `Insert`
 - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
 - $TI(0) = 1$
 - $TI(n) = TI(n - 1) + 1$
 - Unwind to get $TI(n) = n$

```
def Insert(L,v):  
    n = len(L)  
    if n == 0:  
        return([v])  
    if v >= L[-1]:  
        return(L+[v])  
    else:  
        return(Insert(L[:-1],v)+L[-1:])  
  
def ISort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    L = Insert(ISort(L[:-1]),L[-1])  
    return(L)
```

Analysis of recursive insertion sort

- For input of size n , let
 - $TI(n)$ be the time taken by `Insert`
 - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
 - $TI(0) = 1$
 - $TI(n) = TI(n - 1) + 1$
 - Unwind to get $TI(n) = n$
- Set up a recurrence for $TS(n)$
 - $TS(0) = 1$
 - $TS(n) = TS(n - 1) + TI(n - 1)$

```
def Insert(L,v):  
    n = len(L)  
    if n == 0:  
        return([v])  
    if v >= L[-1]:  
        return(L+[v])  
    else:  
        return(Insert(L[:-1],v)+L[-1:])  
  
def ISort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    L = Insert(ISort(L[:-1]),L[-1])  
    return(L)
```

Analysis of recursive insertion sort

- For input of size n , let
 - $TI(n)$ be the time taken by `Insert`
 - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
 - $TI(0) = 1$
 - $TI(n) = TI(n - 1) + 1$
 - Unwind to get $TI(n) = n$
- Set up a recurrence for $TS(n)$
 - $TS(0) = 1$
 - $TS(n) = TS(n - 1) + TI(n - 1)$
- Unwind to get $1 + 2 + \dots + n - 1$

```
def Insert(L,v):  
    n = len(L)  
    if n == 0:  
        return([v])  
    if v >= L[-1]:  
        return(L+[v])  
    else:  
        return(Insert(L[:-1],v)+L[-1:])  
  
def ISort(L):  
    n = len(L)  
    if n < 1:  
        return(L)  
    L = Insert(ISort(L[:-1]),L[-1])  
    return(L)
```

Summary

- Insertion sort is another intuitive algorithm to sort a list

Summary

- Insertion sort is another intuitive algorithm to sort a list
- Create a new sorted list
- Repeatedly insert elements into the sorted list

Summary

- Insertion sort is another intuitive algorithm to sort a list
- Create a new sorted list
- Repeatedly insert elements into the sorted list
- Worst case complexity is $O(n^2)$
 - Unlike selection sort, not all cases take time n^2
 - If list is already sorted, `Insert` stops in 1 step
 - Overall time can be close to $O(n)$

Merge Sort

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 2

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Strategy 3

- Divide the list into two halves

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Strategy 3

- Divide the list into two halves
- Separately sort the left and right half

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Strategy 3

- Divide the list into two halves
- Separately sort the left and right half
- Combine the two sorted halves to get a fully sorted list

Combining two sorted lists

- Combine two sorted lists A and B into a single sorted list C

Combining two sorted lists

- Combine two sorted lists **A** and **B** into a single sorted list **C**
 - Compare first elements of **A** and **B**

Combining two sorted lists

- Combine two sorted lists **A** and **B** into a single sorted list **C**
 - Compare first elements of **A** and **B**
 - Move the smaller of the two to **C**

Combining two sorted lists

- Combine two sorted lists **A** and **B** into a single sorted list **C**
 - Compare first elements of **A** and **B**
 - Move the smaller of the two to **C**
 - Repeat till you exhaust **A** and **B**

Combining two sorted lists

- Combine two sorted lists **A** and **B** into a single sorted list **C**
 - Compare first elements of **A** and **B**
 - Move the smaller of the two to **C**
 - Repeat till you exhaust **A** and **B**

32 74 89
21 55 64

Combining two sorted lists

- Combine two sorted lists **A** and **B** into a single sorted list **C**
 - Compare first elements of **A** and **B**
 - Move the smaller of the two to **C**
 - Repeat till you exhaust **A** and **B**

32 74 89

21 55 64

21

Combining two sorted lists

- Combine two sorted lists **A** and **B** into a single sorted list **C**
 - Compare first elements of **A** and **B**
 - Move the smaller of the two to **C**
 - Repeat till you exhaust **A** and **B**

32 74 89

21 55 64

21 32

Combining two sorted lists

- Combine two sorted lists **A** and **B** into a single sorted list **C**
 - Compare first elements of **A** and **B**
 - Move the smaller of the two to **C**
 - Repeat till you exhaust **A** and **B**
- | | | | |
|--|----|----|----|
| | 32 | 74 | 89 |
| | 21 | 55 | 64 |
| | 21 | 32 | 55 |

Combining two sorted lists

- Combine two sorted lists **A** and **B** into a single sorted list **C**
 - Compare first elements of **A** and **B**
 - Move the smaller of the two to **C**
 - Repeat till you exhaust **A** and **B**
- | | | | |
|----|----|----|----|
| 32 | 74 | 89 | |
| 21 | 55 | 64 | |
| 21 | 32 | 55 | 64 |

Combining two sorted lists

- Combine two sorted lists **A** and **B** into a single sorted list **C**
 - Compare first elements of **A** and **B**
 - Move the smaller of the two to **C**
 - Repeat till you exhaust **A** and **B**
- | | | | | | |
|--|----|----|----|----|----|
| | 32 | 74 | 89 | | |
| | 21 | 55 | 64 | | |
| | 21 | 32 | 55 | 64 | 74 |

Combining two sorted lists

- Combine two sorted lists **A** and **B** into a single sorted list **C**

32 74 89

21 55 64

- Compare first elements of **A** and **B**
- Move the smaller of the two to **C**
- Repeat till you exhaust **A** and **B**

21 32 55 64 74 89

Combining two sorted lists

- Combine two sorted lists **A** and **B** into a single sorted list **C**
 - Compare first elements of **A** and **B**
 - Move the smaller of the two to **C**
 - Repeat till you exhaust **A** and **B**
- **Merging A and B**

32 74 89

21 55 64

21 32 55 64 74 89

Merge sort

- Let n be the length of L

Merge sort

- Let n be the length of L
- Sort $A[:n//2]$

Merge sort

- Let n be the length of L
- Sort $A[:n//2]$
- Sort $A[n//2:]$

Merge sort

- Let n be the length of L
- Sort $A[:n//2]$
- Sort $A[n//2:]$
- Merge the sorted halves into B

Merge sort

- Let n be the length of L
- Sort $A[:n//2]$
- Sort $A[n//2:]$
- Merge the sorted halves into B
- How do we sort $A[:n//2]$ and $A[n//2:]$?

Merge sort

- Let n be the length of L
- Sort $A[:n//2]$
- Sort $A[n//2:]$
- Merge the sorted halves into B
- How do we sort $A[:n//2]$ and $A[n//2:]$?
 - Recursively, same strategy!

Merge sort

- Let n be the length of L
- Sort $A[:n//2]$
- Sort $A[n//2:]$
- Merge the sorted halves into B
- How do we sort $A[:n//2]$ and $A[n//2:]$?
 - Recursively, same strategy!

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- How do we sort

$A[:n//2]$ and
 $A[n//2:]$?

- Recursively, same strategy!

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- How do we sort

$A[:n//2]$ and
 $A[n//2:]$?

- Recursively, same strategy!

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- How do we sort

$A[:n//2]$ and
 $A[n//2:]$?

- Recursively, same strategy!

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- How do we sort

$A[:n//2]$ and
 $A[n//2:]$?

- Recursively, same strategy!

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- How do we sort

$A[:n//2]$ and
 $A[n//2:]$?

- Recursively, same strategy!

32	43	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- How do we sort

$A[:n//2]$ and
 $A[n//2:]$?

- Recursively, same strategy!

32	43	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- How do we sort

$A[:n//2]$ and
 $A[n//2:]$?

- Recursively, same strategy!

32	43	22	78	57	63	91	13
----	----	----	----	----	----	----	----

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- How do we sort

$A[:n//2]$ and
 $A[n//2:]$?

- Recursively, same strategy!

32	43	22	78	57	63	13	91
----	----	----	----	----	----	----	----

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- How do we sort

$A[:n//2]$ and
 $A[n//2:]$?

- Recursively, same strategy!

32	43	22	78	57	63	13	91
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

22	32	43	78	63	57	91	13
----	----	----	----	----	----	----	----

- How do we sort

$A[:n//2]$ and
 $A[n//2:]$?

- Recursively, same strategy!

32	43	22	78	57	63	13	91
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

22	32	43	78	13	57	63	91
----	----	----	----	----	----	----	----

- How do we sort $A[:n//2]$ and $A[n//2:]$?

- Recursively, same strategy!

32	43	22	78	57	63	13	91
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

22	32	43	78	13	57	63	91
----	----	----	----	----	----	----	----

- How do we sort $A[:n//2]$ and $A[n//2:]$?

- Recursively, same strategy!

Merge sort

- Let n be the length of L

- Sort $A[:n//2]$

13	22	32	43	57	63	78	91
----	----	----	----	----	----	----	----

- Sort $A[n//2:]$

- Merge the sorted halves into B

22	32	43	78	13	57	63	91
----	----	----	----	----	----	----	----

- How do we sort $A[:n//2]$ and $A[n//2:]$?

- Recursively, same strategy!

Merge sort

- Let n be the length of L
- Sort $A[:n//2]$
- Sort $A[n//2:]$
- Merge the sorted halves into B
- How do we sort $A[:n//2]$ and $A[n//2:]$?
 - Recursively, same strategy!

13	22	32	43	57	63	78	91
----	----	----	----	----	----	----	----

Merge sort

- Let n be the length of L
- Sort $A[:n//2]$
- Sort $A[n//2:]$
- Merge the sorted halves into B
- How do we sort $A[:n//2]$ and $A[n//2:]$?
 - Recursively, same strategy!

Merge sort

- Let n be the length of L
- Sort $A[:n//2]$
- Sort $A[n//2:]$
- Merge the sorted halves into B
- How do we sort $A[:n//2]$ and $A[n//2:]$?
 - Recursively, same strategy!

Divide and Conquer

- Break up the problem into disjoint parts
- Solve each part separately
- Combine the solutions efficiently

Merging sorted lists

- Combine two sorted lists **A** and **B** into **C**

Merging sorted lists

- Combine two sorted lists **A** and **B** into **C**
 - If **A** is empty, copy **B** into **C**

Merging sorted lists

- Combine two sorted lists **A** and **B** into **C**
 - If **A** is empty, copy **B** into **C**
 - If **B** is empty, copy **A** into **C**

Merging sorted lists

- Combine two sorted lists **A** and **B** into **C**
 - If **A** is empty, copy **B** into **C**
 - If **B** is empty, copy **A** into **C**
 - Otherwise, compare first elements of **A** and **B**
 - Move the smaller of the two to **C**

Merging sorted lists

- Combine two sorted lists **A** and **B** into **C**
 - If **A** is empty, copy **B** into **C**
 - If **B** is empty, copy **A** into **C**
 - Otherwise, compare first elements of **A** and **B**
 - Move the smaller of the two to **C**
 - Repeat till all elements of **A** and **B** have been moved

Merging sorted lists

- Combine two sorted lists A and B into C
 - If A is empty, copy B into C
 - If B is empty, copy A into C
 - Otherwise, compare first elements of A and B
 - Move the smaller of the two to C
 - Repeat till all elements of A and B have been moved

```
def merge(A,B):  
    (m,n) = (len(A),len(B))  
    (C,i,j,k) = ([],0,0,0)  
    while k < m+n:  
        if i == m:  
            C.extend(B[j:])  
            k = k + (n-j)  
        elif j == n:  
            C.extend(A[i:])  
            k = k + (m-i)  
        elif A[i] < B[j]:  
            C.append(A[i])  
            (i,k) = (i+1,k+1)  
        else:  
            C.append(B[j])  
            (j,k) = (j+1,k+1)  
    return(C)
```

Merge sort

- To sort A into B, both of length n

Merge sort

- To sort A into B, both of length n
- If $n \leq 1$, nothing to be done

Merge sort

- To sort A into B, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise

Merge sort

- To sort A into B, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
 - Sort $A[:n//2]$ into L

Merge sort

- To sort A into B, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
 - Sort A[: $n//2$] into L
 - Sort A[$n//2:$] into R

Merge sort

- To sort A into B, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
 - Sort A[: $n//2$] into L
 - Sort A[$n//2:$] into R
 - Merge L and R into B

Merge sort

- To sort A into B, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
 - Sort $A[:n//2]$ into L
 - Sort $A[n//2:]$ into R
 - Merge L and R into B

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```

Summary

- Merge sort using divide and conquer to sort a list

Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves

Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves
- Sort each half

Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves
- Sort each half
- Merge the sorted halves

Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves
- Sort each half
- Merge the sorted halves
- Next, we have to check that the complexity is less than $O(n^2)$

Analysis of Merge Sort

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 2

Merge sort

- To sort A into B , both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
 - Sort $A[:n//2]$ into L
 - Sort $A[n//2:]$ into R
 - Merge L and R into B

Merging two sorted lists A and B into C

- If A is empty, copy B into C
- If B is empty, copy A into C
- Otherwise, compare first elements of A and B
 - Move the smaller of the two to C
- Repeat till all elements of A and B have been moved

Analysing merge

- Merge A of length m , B of length n

```
def merge(A,B):
    (m,n) = (len(A),len(B))
    (C,i,j,k) = ([],0,0,0)
    while k < m+n:
        if i == m:
            C.extend(B[j:])
            k = k + (n-j)
        elif j == n:
            C.extend(A[i:])
            k = k + (m-i)
        elif A[i] < B[j]:
            C.append(A[i])
            (i,k) = (i+1,k+1)
        else:
            C.append(B[j])
            (j,k) = (j+1,k+1)
    return(C)
```

Analysing merge

- Merge A of length m , B of length n
- Output list C has length $m + n$

```
def merge(A,B):
    (m,n) = (len(A),len(B))
    (C,i,j,k) = ([],0,0,0)
    while k < m+n:
        if i == m:
            C.extend(B[j:])
            k = k + (n-j)
        elif j == n:
            C.extend(A[i:])
            k = k + (m-i)
        elif A[i] < B[j]:
            C.append(A[i])
            (i,k) = (i+1,k+1)
        else:
            C.append(B[j])
            (j,k) = (j+1,k+1)
    return(C)
```

Analysing merge

- Merge A of length m , B of length n
- Output list C has length $m + n$
- In each iteration we add (at least) one element to C

```
def merge(A,B):  
    (m,n) = (len(A),len(B))  
    (C,i,j,k) = ([],0,0,0)  
    while k < m+n:  
        if i == m:  
            C.extend(B[j:])  
            k = k + (n-j)  
        elif j == n:  
            C.extend(A[i:])  
            k = k + (m-i)  
        elif A[i] < B[j]:  
            C.append(A[i])  
            (i,k) = (i+1,k+1)  
        else:  
            C.append(B[j])  
            (j,k) = (j+1,k+1)  
    return(C)
```

Analysing merge

- Merge A of length m , B of length n
- Output list C has length $m + n$
- In each iteration we add (at least) one element to C
- Hence `merge` takes time $O(m + n)$

```
def merge(A,B):  
    (m,n) = (len(A),len(B))  
    (C,i,j,k) = ([],0,0,0)  
    while k < m+n:  
        if i == m:  
            C.extend(B[j:])  
            k = k + (n-j)  
        elif j == n:  
            C.extend(A[i:])  
            k = k + (m-i)  
        elif A[i] < B[j]:  
            C.append(A[i])  
            (i,k) = (i+1,k+1)  
        else:  
            C.append(B[j])  
            (j,k) = (j+1,k+1)  
    return(C)
```

Analysing merge

- Merge A of length m , B of length n
- Output list C has length $m + n$
- In each iteration we add (at least) one element to C
- Hence `merge` takes time $O(m + n)$
- Recall that $m + n \leq 2(\max(m, n))$

```
def merge(A,B):  
    (m,n) = (len(A),len(B))  
    (C,i,j,k) = ([],0,0,0)  
    while k < m+n:  
        if i == m:  
            C.extend(B[j:])  
            k = k + (n-j)  
        elif j == n:  
            C.extend(A[i:])  
            k = k + (m-i)  
        elif A[i] < B[j]:  
            C.append(A[i])  
            (i,k) = (i+1,k+1)  
        else:  
            C.append(B[j])  
            (j,k) = (j+1,k+1)  
    return(C)
```

Analysing merge

- Merge A of length m , B of length n
- Output list C has length $m + n$
- In each iteration we add (at least) one element to C
- Hence `merge` takes time $O(m + n)$
- Recall that $m + n \leq 2(\max(m, n))$
- If $m \approx n$, `merge` take time $O(n)$

```
def merge(A,B):  
    (m,n) = (len(A),len(B))  
    (C,i,j,k) = ([],0,0,0)  
    while k < m+n:  
        if i == m:  
            C.extend(B[j:])  
            k = k + (n-j)  
        elif j == n:  
            C.extend(A[i:])  
            k = k + (m-i)  
        elif A[i] < B[j]:  
            C.append(A[i])  
            (i,k) = (i+1,k+1)  
        else:  
            C.append(B[j])  
            (j,k) = (j+1,k+1)  
    return(C)
```

Analysing mergesort

- Let $T(n)$ be the time taken for input of size n
 - For simplicity, assume $n = 2^k$ for some k

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```

Analysing mergesort

- Let $T(n)$ be the time taken for input of size n
 - For simplicity, assume $n = 2^k$ for some k
- Recurrence
 - $T(0) = T(1) = 1$
 - $T(n) = 2T(n/2) + n$
 - Solve two subproblems of size $n/2$
 - Merge the solutions in time $n/2 + n/2 = n$

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```

Analysing mergesort

- Let $T(n)$ be the time taken for input of size n
 - For simplicity, assume $n = 2^k$ for some k
- Recurrence
 - $T(0) = T(1) = 1$
 - $T(n) = 2T(n/2) + n$
 - Solve two subproblems of size $n/2$
 - Merge the solutions in time $n/2 + n/2 = n$
- Unwind the recurrence to solve

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```

Analysing mergesort

■ Recurrence

- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```

Analysing mergesort

■ Recurrence

- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$
- $T(n) = 2T(n/2) + n$

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```

Analysing mergesort

■ Recurrence

- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$
- $$T(n) = 2T(n/2) + n = 2[2T(n/4) + n/2] + n$$

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```

Analysing mergesort

■ Recurrence

- $T(0) = T(1) = 1$
 - $T(n) = 2T(n/2) + n$
- $T(n) = 2T(n/2) + n$
- $$= 2[2T(n/4) + n/2] + n = 2^2T(n/2^2) + 2n$$

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```

Analysing mergesort

■ Recurrence

- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$

- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n = 2^2 T(n/2^2) + 2n \\ &= 2^2 [2T(n/2^3) + n/2^2] + 2n = 2^3 T(n/2^3) + 3n \end{aligned}$$

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```

Analysing mergesort

■ Recurrence

- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$

- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n = 2^2 T(n/2^2) + 2n \\ &= 2^2 [2T(n/2^3) + n/2^2] + 2n = 2^3 T(n/2^3) + 3n \\ &\quad \vdots \\ &= 2^k T(n/2^k) + kn \end{aligned}$$

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```

Analysing mergesort

■ Recurrence

- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$

- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n = 2^2 T(n/2^2) + 2n \\ &= 2^2 [2T(n/2^3) + n/2^2] + 2n = 2^3 T(n/2^3) + 3n \\ &\quad \vdots \\ &= 2^k T(n/2^k) + kn \end{aligned}$$

- When $k = \log n$, $T(n/2^k) = T(1) = 1$

```
def mergesort(A):
    n = len(A)

    if n <= 1:
        return(A)

    L = mergesort(A[:n//2])
    R = mergesort(A[n//2:])

    B = merge(L,R)

    return(B)
```

Analysing mergesort

■ Recurrence

- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$

- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n = 2^2 T(n/2^2) + 2n \\ &= 2^2 [2T(n/2^3) + n/2^2] + 2n = 2^3 T(n/2^3) + 3n \\ &\quad \vdots \\ &= 2^k T(n/2^k) + kn \end{aligned}$$

- When $k = \log n$, $T(n/2^k) = T(1) = 1$
- $T(n) = 2^{\log n} T(1) + (\log n)n = n + n \log n$

```
def mergesort(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A)  
  
    L = mergesort(A[:n//2])  
    R = mergesort(A[n//2:])  
  
    B = merge(L,R)  
  
    return(B)
```

Analysing mergesort

■ Recurrence

- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$

- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n = 2^2 T(n/2^2) + 2n \\ &= 2^2 [2T(n/2^3) + n/2^2] + 2n = 2^3 T(n/2^3) + 3n \\ &\quad \vdots \\ &= 2^k T(n/2^k) + kn \end{aligned}$$

- When $k = \log n$, $T(n/2^k) = T(1) = 1$
- $T(n) = 2^{\log n} T(1) + (\log n)n = n + n \log n$
- Hence $T(n)$ is $O(n \log n)$

```
def mergesort(A):
    n = len(A)

    if n <= 1:
        return(A)

    L = mergesort(A[:n//2])
    R = mergesort(A[n//2:])

    B = merge(L,R)

    return(B)
```

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
 - Union of two sorted lists — discard duplicates, if $A[i] == B[j]$ move just one copy to C and increment both i and j

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
 - Union of two sorted lists — discard duplicates, if $A[i] == B[j]$ move just one copy to C and increment both i and j
 - Intersection of two sorted lists — when $A[i] == B[j]$, move one copy to C , otherwise discard the smaller of $A[i], B[j]$

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
 - Union of two sorted lists — discard duplicates, if $A[i] == B[j]$ move just one copy to C and increment both i and j
 - Intersection of two sorted lists — when $A[i] == B[j]$, move one copy to C , otherwise discard the smaller of $A[i], B[j]$
 - List difference — elements in A but not in B

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
 - Union of two sorted lists — discard duplicates, if $A[i] == B[j]$ move just one copy to C and increment both i and j
 - Intersection of two sorted lists — when $A[i] == B[j]$, move one copy to C , otherwise discard the smaller of $A[i], B[j]$
 - List difference — elements in A but not in B
- Merge needs to create a new list to hold the merged elements
 - No obvious way to efficiently merge two lists in place
 - Extra storage can be costly

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
 - Union of two sorted lists — discard duplicates, if $A[i] == B[j]$ move just one copy to C and increment both i and j
 - Intersection of two sorted lists — when $A[i] == B[j]$, move one copy to C , otherwise discard the smaller of $A[i], B[j]$
 - List difference — elements in A but not in B
- Merge needs to create a new list to hold the merged elements
 - No obvious way to efficiently merge two lists in place
 - Extra storage can be costly
- Inherently recursive
 - Recursive calls and returns are expensive

Quicksort

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 3

Shortcomings of merge sort

- Merge needs to create a new list to hold the merged elements
 - No obvious way to efficiently merge two lists in place
 - Extra storage can be costly
- Inherently recursive
 - Recursive calls and returns are expensive

Shortcomings of merge sort

- Merge needs to create a new list to hold the merged elements
 - No obvious way to efficiently merge two lists in place
 - Extra storage can be costly
- Inherently recursive
 - Recursive calls and returns are expensive
- Merging happens because elements in the left half need to move to the right half and vice versa
 - Consider an input of the form [0,2,4,6,1,3,5,9]

Shortcomings of merge sort

- Merge needs to create a new list to hold the merged elements
 - No obvious way to efficiently merge two lists in place
 - Extra storage can be costly
- Inherently recursive
 - Recursive calls and returns are expensive
- Merging happens because elements in the left half need to move to the right half and vice versa
 - Consider an input of the form [0,2,4,6,1,3,5,9]
- Can we divide the list so that everything on the left is smaller than everything on the right?
 - No need to merge!

Divide and conquer without merging

- Suppose the median of L is m

Divide and conquer without merging

- Suppose the median of L is m
- Move all values $\leq m$ to left half of L
 - Right half has values $> m$

Divide and conquer without merging

- Suppose the median of L is m
- Move all values $\leq m$ to left half of L
 - Right half has values $> m$
- Recursively sort left and right halves
 - L is now sorted, no merge!

Divide and conquer without merging

- Suppose the median of L is m
- Move all values $\leq m$ to left half of L
 - Right half has values $> m$
- Recursively sort left and right halves
 - L is now sorted, no merge!
- Recurrence: $T(n) = 2T(n/2) + n$
 - Rearrange in a single pass, time $O(n)$

Divide and conquer without merging

- Suppose the median of L is m
- Move all values $\leq m$ to left half of L
 - Right half has values $> m$
- Recursively sort left and right halves
 - L is now sorted, no merge!
- Recurrence: $T(n) = 2T(n/2) + n$
 - Rearrange in a single pass, time $O(n)$
- So $T(n)$ is $O(n \log n)$

Divide and conquer without merging

- Suppose the median of L is m
- How do we find the median?
- Move all values $\leq m$ to left half of L
 - Right half has values $> m$
- Recursively sort left and right halves
 - L is now sorted, no merge!
- Recurrence: $T(n) = 2T(n/2) + n$
 - Rearrange in a single pass, time $O(n)$
- So $T(n)$ is $O(n \log n)$

Divide and conquer without merging

- Suppose the median of L is m
 - Move all values $\leq m$ to left half of L
 - Right half has values $> m$
 - Recursively sort left and right halves
 - L is now sorted, no merge!
 - Recurrence: $T(n) = 2T(n/2) + n$
 - Rearrange in a single pass, time $O(n)$
 - So $T(n)$ is $O(n \log n)$
- How do we find the median?
 - Sort and pick up the middle element

Divide and conquer without merging

- Suppose the median of L is m
 - Move all values $\leq m$ to left half of L
 - Right half has values $> m$
 - Recursively sort left and right halves
 - L is now sorted, no merge!
 - Recurrence: $T(n) = 2T(n/2) + n$
 - Rearrange in a single pass, time $O(n)$
 - So $T(n)$ is $O(n \log n)$
- How do we find the median?
 - Sort and pick up the middle element
 - But our aim is to sort the list!

Divide and conquer without merging

- Suppose the median of L is m
 - Move all values $\leq m$ to left half of L
 - Right half has values $> m$
 - Recursively sort left and right halves
 - L is now sorted, no merge!
 - Recurrence: $T(n) = 2T(n/2) + n$
 - Rearrange in a single pass, time $O(n)$
 - So $T(n)$ is $O(n \log n)$
-
- How do we find the median?
 - Sort and pick up the middle element
 - But our aim is to sort the list!
 - Instead pick some value in L — pivot
 - Split L with respect to the pivot element

Quicksort [C.A.R. Hoare]

- Choose a pivot element
 - Typically the first element in the array

Quicksort [C.A.R. Hoare]

- Choose a pivot element
 - Typically the first element in the array
- Partition L into lower and upper parts with respect to the pivot

Quicksort [C.A.R. Hoare]

- Choose a pivot element
 - Typically the first element in the array
- Partition L into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition

Quicksort [C.A.R. Hoare]

- Choose a pivot element
 - Typically the first element in the array
- Partition L into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

Quicksort [C.A.R. Hoare]

- Choose a pivot element
 - Typically the first element in the array
- Partition L into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

High level view of quicksort

- Input list

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

Quicksort [C.A.R. Hoare]

- Choose a pivot element
 - Typically the first element in the array
- Partition L into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

High level view of quicksort

- Input list

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Identify pivot

Quicksort [C.A.R. Hoare]

- Choose a pivot element
 - Typically the first element in the array
- Partition L into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

High level view of quicksort

- Input list

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Identify pivot
- Mark lower elements and upper elements

Quicksort [C.A.R. Hoare]

- Choose a pivot element
 - Typically the first element in the array
- Partition L into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

High level view of quicksort

- Input list

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Identify pivot
- Mark lower elements and upper elements
- Rearrange the elements as lower–pivot–upper

32	22	13	43	78	63	57	91
----	----	----	----	----	----	----	----

Quicksort [C.A.R. Hoare]

- Choose a pivot element
 - Typically the first element in the array
- Partition L into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

High level view of quicksort

- Input list

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Identify **pivot**
- Mark **lower elements** and **upper elements**

- Rearrange the elements as lower–pivot–upper

32	22	13	43	78	63	57	91
----	----	----	----	----	----	----	----

- Recursively sort the lower and upper partitions

Partitioning

- Scan the list from left to right

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified

Partitioning

- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element

Partitioning

- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element

Partitioning

- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.

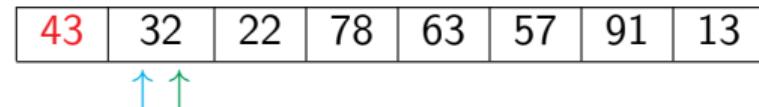
Partitioning

- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

Partitioning

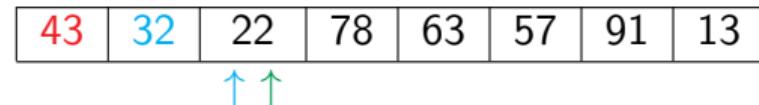
- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Partitioning

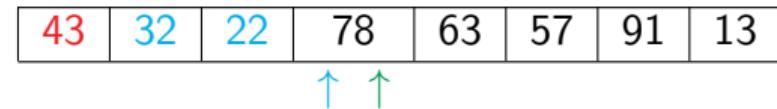
- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Partitioning

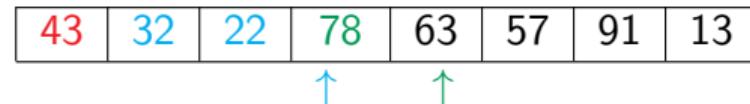
- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Partitioning

- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Partitioning

- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Partitioning

- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Partitioning

- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Partitioning

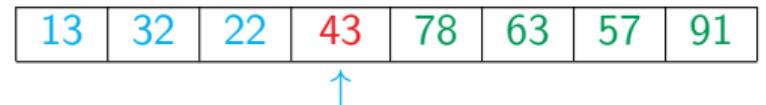
- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Partitioning

- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, Unclassified
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments
- After partitioning, exchange the pivot with the last element of the **Lower** segment

Quicksort code

- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**
- Classify the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Summary

- Quicksort uses divide and conquer, like merge sort

Summary

- Quicksort uses divide and conquer, like merge sort
- By partitioning the list carefully, we avoid a merge step
 - This allows an in place sort

Summary

- Quicksort uses divide and conquer, like merge sort
- By partitioning the list carefully, we avoid a merge step
 - This allows an in place sort
- We can also provide an iterative implementation to avoid the cost of recursive calls

Summary

- Quicksort uses divide and conquer, like merge sort
- By partitioning the list carefully, we avoid a merge step
 - This allows an in place sort
- We can also provide an iterative implementation to avoid the cost of recursive calls
- The partitioning strategy we described is not the only one used in the literature
 - Can build the lower and upper segments from opposite ends and meet in the middle

Summary

- Quicksort uses divide and conquer, like merge sort
- By partitioning the list carefully, we avoid a merge step
 - This allows an in place sort
- We can also provide an iterative implementation to avoid the cost of recursive calls
- The partitioning strategy we described is not the only one used in the literature
 - Can build the lower and upper segments from opposite ends and meet in the middle
- Need to analyse the complexity of quick sort

Analysis of Quicksort

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 3

Quicksort

- Choose a pivot element
- Partition L into lower and upper segments with respect to the pivot
- Move the pivot between the lower and upper segments
- Recursively sort the two partitions

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Analysis

- Partitioning with respect to the pivot takes time $O(n)$

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Analysis

- Partitioning with respect to the pivot takes time $O(n)$
- If the pivot is the median
 - $T(n) = 2T(n/2) + n$
 - $T(n)$ is $O(n \log n)$

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Analysis

- Partitioning with respect to the pivot takes time $O(n)$
- If the pivot is the median
 - $T(n) = 2T(n/2) + n$
 - $T(n)$ is $O(n \log n)$
- Worst case? Pivot is maximum or minimum
 - Partitions are of size 0, $n - 1$
 - $T(n) = T(n - 1) + n$
 - $T(n) = n + (n - 1) + \dots + 1$
 - $T(n)$ is $O(n^2)$

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Analysis

- Partitioning with respect to the pivot takes time $O(n)$
- If the pivot is the median
 - $T(n) = 2T(n/2) + n$
 - $T(n)$ is $O(n \log n)$
- Worst case? Pivot is maximum or minimum
 - Partitions are of size 0, $n - 1$
 - $T(n) = T(n - 1) + n$
 - $T(n) = n + (n - 1) + \dots + 1$
 - $T(n)$ is $O(n^2)$
- Already sorted array: worst case!

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Analysis . . .

- However, average case is $O(n \log n)$

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Analysis . . .

- However, average case is $O(n \log n)$
- Sorting is a rare situation where we can compute this
 - Values don't matter, only relative order is important
 - Analyze behaviour over permutations of $\{1, 2, \dots, n\}$
 - Each input permutation equally likely

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Analysis . . .

- However, average case is $O(n \log n)$
- Sorting is a rare situation where we can compute this
 - Values don't matter, only relative order is important
 - Analyze behaviour over permutations of $\{1, 2, \dots, n\}$
 - Each input permutation equally likely
- Expected running time is $O(n \log n)$

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Randomization

- Any fixed choice of pivot allows us to construct worst case input

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Randomization

- Any fixed choice of pivot allows us to construct worst case input
- Instead, choose pivot position **randomly** at each step

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Randomization

- Any fixed choice of pivot allows us to construct worst case input
- Instead, choose pivot position **randomly** at each step
- Expected running time is again $O(n \log n)$

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Iterative quicksort

- Recursive calls work on disjoint segments
 - No recombination of results is required

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Iterative quicksort

- Recursive calls work on disjoint segments
 - No recombination of results is required
- Can explicitly keep track of left and right endpoints of each segment to be sorted

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Quicksort in practice

- In practice, quicksort is very fast

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Quicksort in practice

- In practice, quicksort is very fast
- Very often the default algorithm used for in-built sort functions
 - Sorting a column in a spreadsheet
 - Library sort function in a programming language

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[1],L[lower-1]) = (L[lower-1],L[1])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```

Summary

- The worst case complexity of quicksort is $O(n^2)$

Summary

- The worst case complexity of quicksort is $O(n^2)$
- However, the average case is $O(n \log n)$

Summary

- The worst case complexity of quicksort is $O(n^2)$
- However, the average case is $O(n \log n)$
- Randomly choosing the pivot is a good strategy to beat worst case inputs

Summary

- The worst case complexity of quicksort is $O(n^2)$
- However, the average case is $O(n \log n)$
- Randomly choosing the pivot is a good strategy to beat worst case inputs
- Quicksort works in-place and can be implemented iteratively

Summary

- The worst case complexity of quicksort is $O(n^2)$
- However, the average case is $O(n \log n)$
- Randomly choosing the pivot is a good strategy to beat worst case inputs
- Quicksort works in-place and can be implemented iteratively
- Very fast in practice, and often used for built-in sorting functions
 - Good example of a situation when the worst case upper bound is pessimistic

Sorting: Concluding Remarks

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 3

Stable sorting

- Often list values are tuples
 - Rows from a table, with multiple columns / attributes
 - A list of students, each student entry has a roll number, name, marks, ...

Stable sorting

- Often list values are tuples
 - Rows from a table, with multiple columns / attributes
 - A list of students, each student entry has a roll number, name, marks, ...
- Suppose students have already been sorted by roll number

Stable sorting

- Often list values are tuples
 - Rows from a table, with multiple columns / attributes
 - A list of students, each student entry has a roll number, name, marks, ...
- Suppose students have already been sorted by roll number
- If we now sort by name, will all students with the same name remain in sorted order with respect to roll number?

Stable sorting

- Often list values are tuples
 - Rows from a table, with multiple columns / attributes
 - A list of students, each student entry has a roll number, name, marks, ...
- Suppose students have already been sorted by roll number
- If we now sort by name, will all students with the same name remain in sorted order with respect to roll number?
- **Stability** of sorting is crucial in many applications

Stable sorting

- Often list values are tuples
 - Rows from a table, with multiple columns / attributes
 - A list of students, each student entry has a roll number, name, marks, ...
- Suppose students have already been sorted by roll number
- If we now sort by name, will all students with the same name remain in sorted order with respect to roll number?
- **Stability** of sorting is crucial in many applications
- Sorting on column *B* should not disturb sorting on column *A*

Stable sorting

- The quicksort implementation we described is not stable
 - Swapping values while partitioning can disturb existing sorted order

Stable sorting

- The quicksort implementation we described is not stable
 - Swapping values while partitioning can disturb existing sorted order
- Merge sort is stable if we merge carefully
 - Do not allow elements from the right to overtake elements on the left
 - While merging, prefer the left list while breaking ties

Other criteria

- Minimizing data movement
 - Imagine each element is a heavy carton
 - Reduce the effort of moving values around

Best sorting algorithm?

- Quicksort is often the algorithm of choice, despite $O(n^2)$ worst case

Best sorting algorithm?

- Quicksort is often the algorithm of choice, despite $O(n^2)$ worst case
- Merge sort is typically used for “external” sorting
 - Database tables that are too large to store in memory all at once
 - Retrieve in parts from the disk and write back

Best sorting algorithm?

- Quicksort is often the algorithm of choice, despite $O(n^2)$ worst case
- Merge sort is typically used for “external” sorting
 - Database tables that are too large to store in memory all at once
 - Retrieve in parts from the disk and write back
- Other $O(n \log n)$ algorithms exist — heapsort

Best sorting algorithm?

- Quicksort is often the algorithm of choice, despite $O(n^2)$ worst case
- Merge sort is typically used for “external” sorting
 - Database tables that are too large to store in memory all at once
 - Retrieve in parts from the disk and write back
- Other $O(n \log n)$ algorithms exist — heapsort
- Sometimes hybrid strategies are used
 - Use divide and conquer for large n
 - Switch to insertion sort when n becomes small (e.g., $n < 16$)

Lists and Arrays

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 3

Sequences

- Two basic ways of storing a sequence of values
 - Lists
 - Arrays
- What's the difference?

Sequences

- Two basic ways of storing a sequence of values
 - Lists
 - Arrays
- What's the difference?
- Lists
 - Flexible length
 - Easy to modify the structure
 - Values are scattered in memory

Sequences

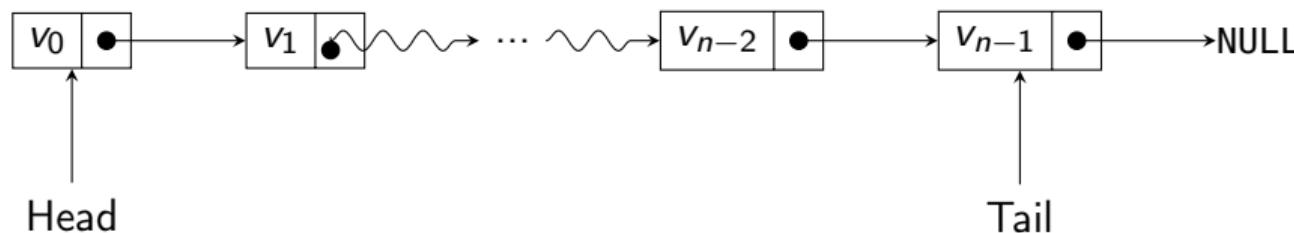
- Two basic ways of storing a sequence of values
 - Lists
 - Arrays
- What's the difference?
 - Lists
 - Flexible length
 - Easy to modify the structure
 - Values are scattered in memory
 - Arrays
 - Fixed size
 - Allocate a contiguous block of memory
 - Supports **random access**

Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - “Linked” list

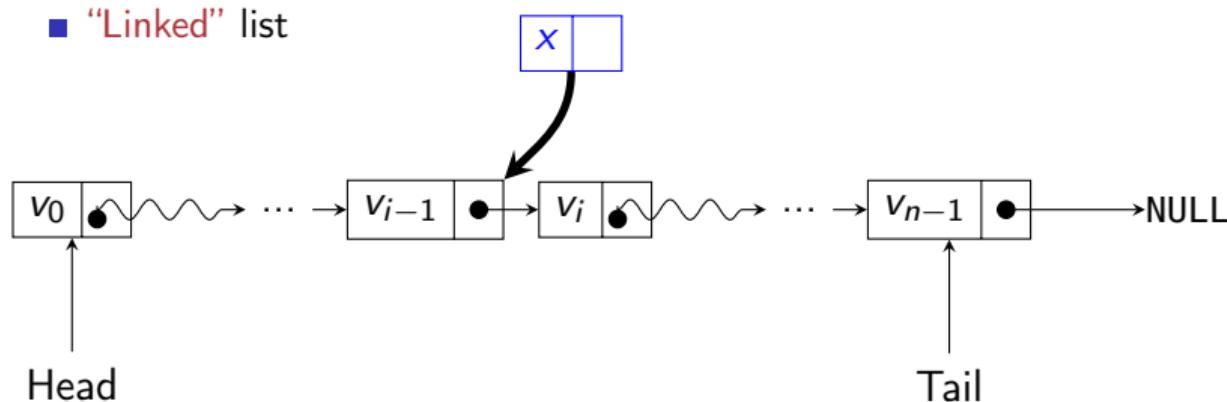
Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - “Linked” list



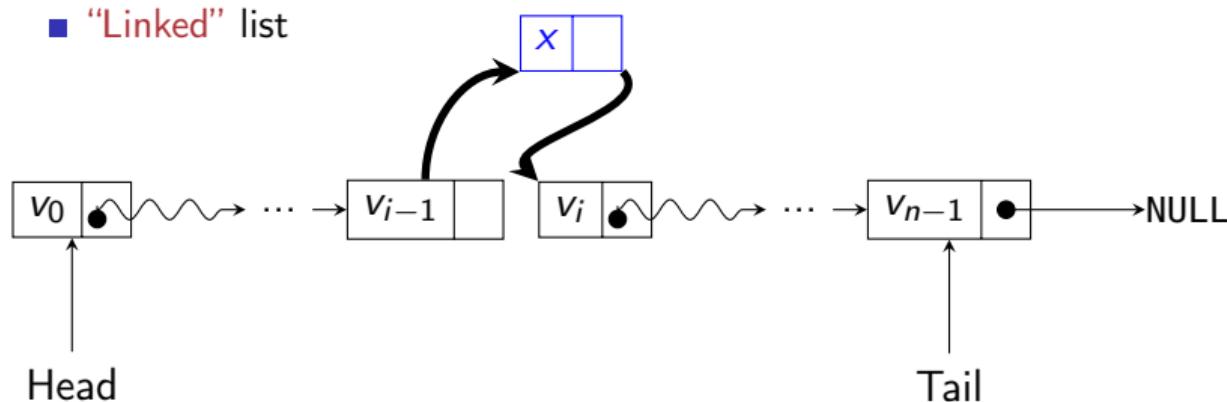
Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - “Linked” list
- Easy to modify
 - Inserting and deletion is easy via local “plumbing”
 - Flexible size



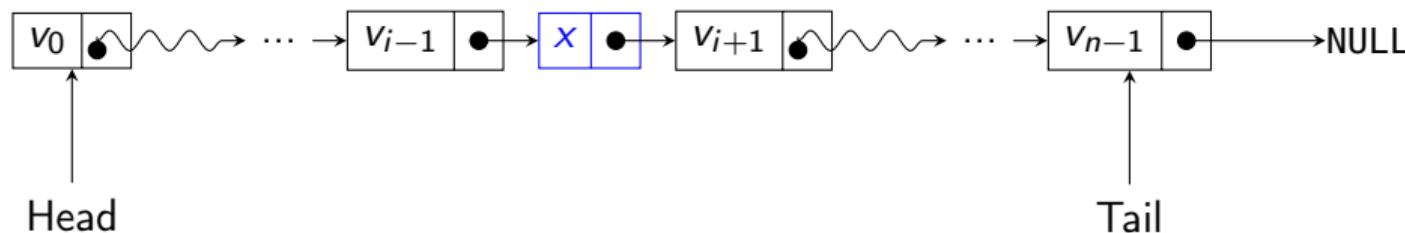
Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - “Linked” list
- Easy to modify
 - Inserting and deletion is easy via local “plumbing”
 - Flexible size



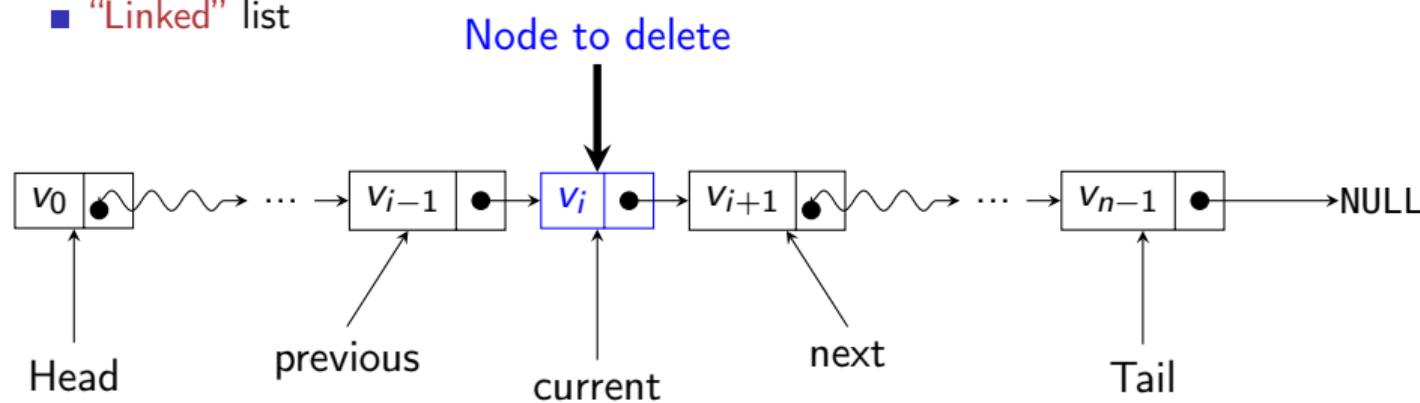
Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - “Linked” list
- Easy to modify
 - Inserting and deletion is easy via local “plumbing”
 - Flexible size



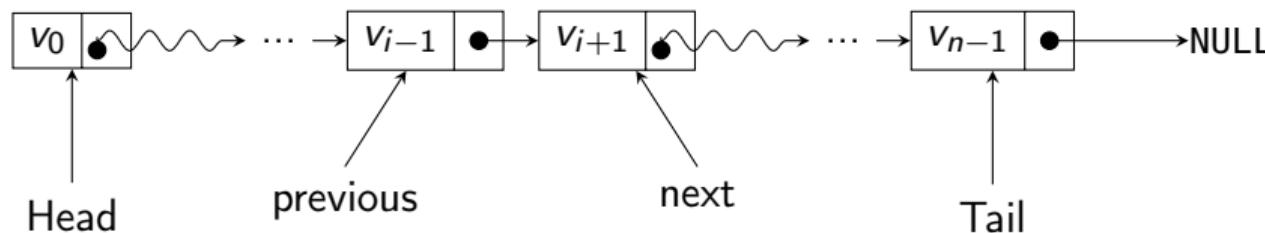
Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - “Linked” list
- Easy to modify
 - Inserting and deletion is easy via local “plumbing”
 - Flexible size



Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - “Linked” list
- Easy to modify
 - Inserting and deletion is easy via local “plumbing”
 - Flexible size



Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - “Linked” list
- Easy to modify
 - Inserting and deletion is easy via local “plumbing”
 - Flexible size
- Need to follow links to access $A[i]$
 - Takes time $O(i)$

Arrays

- Fixed size, declared in advance
- Allocate a contiguous block of memory
 - n times the storage for a single value

Index	Value
$A[0]$	v_0
$A[1]$	v_1
\vdots	\vdots
$A[i]$	v_i
\vdots	\vdots
$A[n-1]$	v_{n-1}

Arrays

- Fixed size, declared in advance
- Allocate a contiguous block of memory
 - n times the storage for a single value
- “Random” access
 - Compute offset to $A[i]$ from $A[0]$
 - Accessing $A[i]$ takes constant time, independent of i

Index	Value
$A[0]$	v_0
$A[1]$	v_1
\vdots	\vdots
$A[i]$	v_i
\vdots	\vdots
$A[n-1]$	v_{n-1}

Arrays

- Fixed size, declared in advance
- Allocate a contiguous block of memory
 - n times the storage for a single value
- “Random” access
 - Compute offset to $A[i]$ from $A[0]$
 - Accessing $A[i]$ takes constant time, independent of i
- Inserting and deleting elements is expensive
 - Expanding and contracting requires moving $O(n)$ elements in the worst case

Index	Value
$A[0]$	v_0
$A[1]$	v_1
:	:
$A[i]$	v_i
:	:
$A[n-1]$	v_{n-1}

Operations

- Exchange $A[i]$ and $A[j]$
 - Constant time for arrays
 - $O(n)$ for lists
- Delete $A[i]$, insert v after $A[i]$
 - Constant time for lists if we are already at $A[i]$
 - $O(n)$ for arrays
- Need to keep implementation in mind when analyzing data structures
 - For instance, can we use binary search to insert in a sorted sequence?
 - Either search is slow, or insertion is slow, still $O(n)$

Summary

- Sequences can be stored as lists or arrays

Summary

- Sequences can be stored as lists or arrays
- Lists are flexible but accessing an element is $O(n)$

Summary

- Sequences can be stored as lists or arrays
- Lists are flexible but accessing an element is $O(n)$
- Arrays support random access but are difficult to expand, contract

Summary

- Sequences can be stored as lists or arrays
- Lists are flexible but accessing an element is $O(n)$
- Arrays support random access but are difficult to expand, contract
- Algorithm analysis needs to take into account the underlying implementation

- Sequences can be stored as lists or arrays
- Lists are flexible but accessing an element is $O(n)$
- Arrays support random access but are difficult to expand, contract
- Algorithm analysis needs to take into account the underlying implementation
- How does it work in Python?
 - Is the built-in list type in Python really a “linked” list?
 - Numpy library provides arrays — are these faster than lists?

Designing a flexible list

Madhavan Mukund

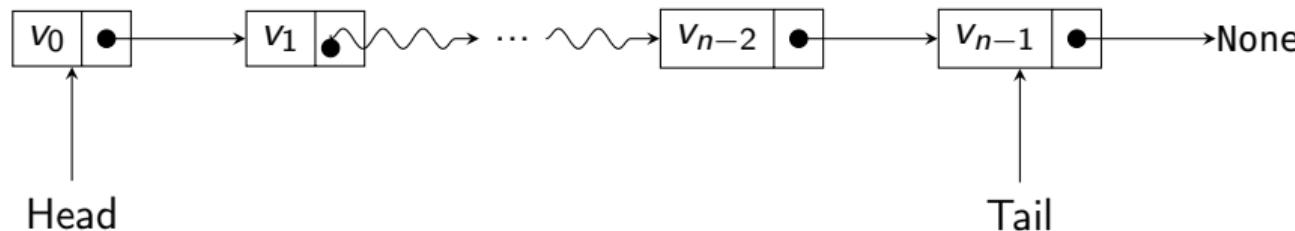
<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 3

Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - “Linked” list
- Easy to modify
 - Inserting and deletion is easy via local “plumbing”
 - Flexible size
- Need to follow links to access `A[i]`
 - Takes time $O(i)$



Implementing lists in Python

- Python class `Node`

```
class Node:  
    def __init__(self, v = None):  
        self.value = v  
        self.next = None  
        return  
  
    def isempty(self):  
        if self.value == None:  
            return(True)  
        else:  
            return(False)
```

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
 - `self.value` is the stored value
 - `self.next` points to next node

```
class Node:  
    def __init__(self, v = None):  
        self.value = v  
        self.next = None  
        return  
  
    def isempty(self):  
        if self.value == None:  
            return(True)  
        else:  
            return(False)
```

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
 - `self.value` is the stored value
 - `self.next` points to next node
- Empty list?
 - `self.value` is `None`

```
class Node:  
    def __init__(self, v = None):  
        self.value = v  
        self.next = None  
        return  
  
    def isempty(self):  
        if self.value == None:  
            return(True)  
        else:  
            return(False)
```

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
 - `self.value` is the stored value
 - `self.next` points to next node
- Empty list?
 - `self.value` is `None`
- Creating lists
 - `l1 = Node()` — empty list
 - `l2 = Node(5)` — singleton list

```
class Node:  
    def __init__(self, v = None):  
        self.value = v  
        self.next = None  
        return  
  
    def isempty(self):  
        if self.value == None:  
            return(True)  
        else:  
            return(False)
```

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
 - `self.value` is the stored value
 - `self.next` points to next node
- Empty list?
 - `self.value` is `None`
- Creating lists
 - `l1 = Node()` — empty list
 - `l2 = Node(5)` — singleton list
 - `l1isempty() == True`
 - `l2isempty() == False`

```
class Node:  
    def __init__(self, v = None):  
        self.value = v  
        self.next = None  
        return  
  
    def isempty(self):  
        if self.value == None:  
            return(True)  
        else:  
            return(False)
```

Appending to a list

- Add `v` to the end of list `l`
- If `l` is empty, update `l.value` from `None` to `v`
- If at last value, `l.next` is `None`
 - Point `next` at new node with value `v`
- Otherwise, recursively append to rest of list

```
def append(self,v):  
    # append, recursive  
    if self isempty():  
        self.value = v  
    elif self.next == None:  
        self.next = Node(v)  
    else:  
        self.next.append(v)  
    return
```

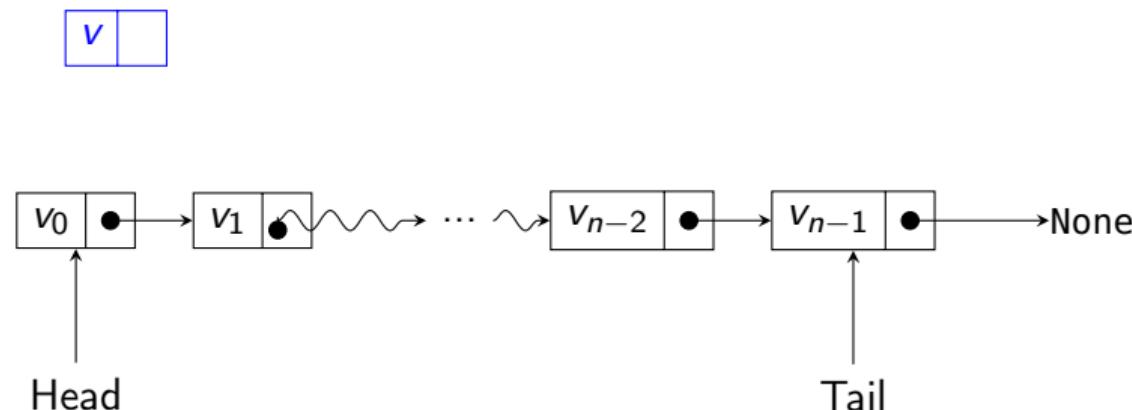
Appending to a list

- Add v to the end of list l
- If l is empty, update $l.value$ from `None` to v
- If at last value, $l.next$ is `None`
 - Point `next` at new node with value v
- Otherwise, recursively append to rest of list
- Iterative implementation
 - If empty, replace $l.value$ by v
 - Loop through $l.next$ to end of list
 - Add v at the end of the list

```
def append(self,v):  
    # append, iterative  
    if selfisempty():  
        self.value = v  
        return  
  
    temp = self  
    while temp.next != None:  
        temp = temp.next  
  
    temp.next = Node(v)  
    return
```

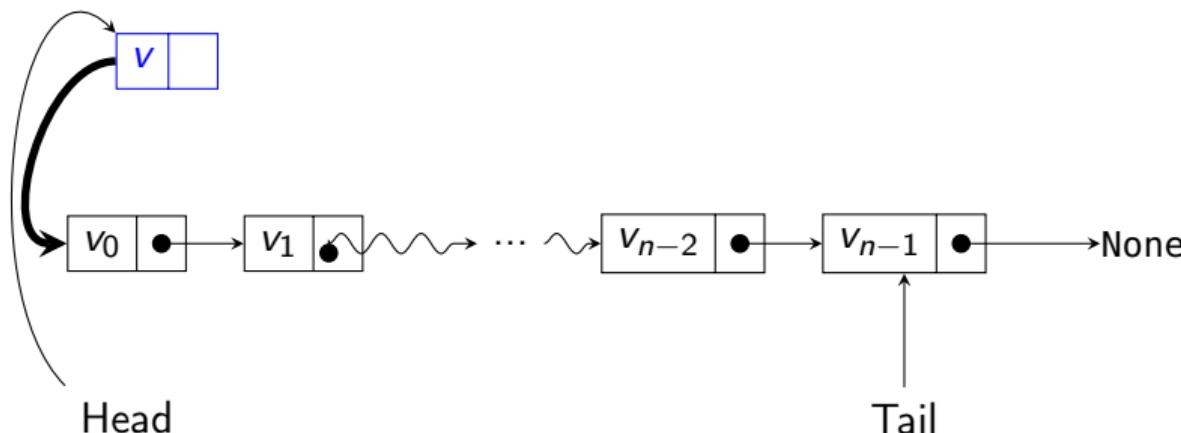
Insert at the start of the list

- Want to insert v at head
- Create a new node with v



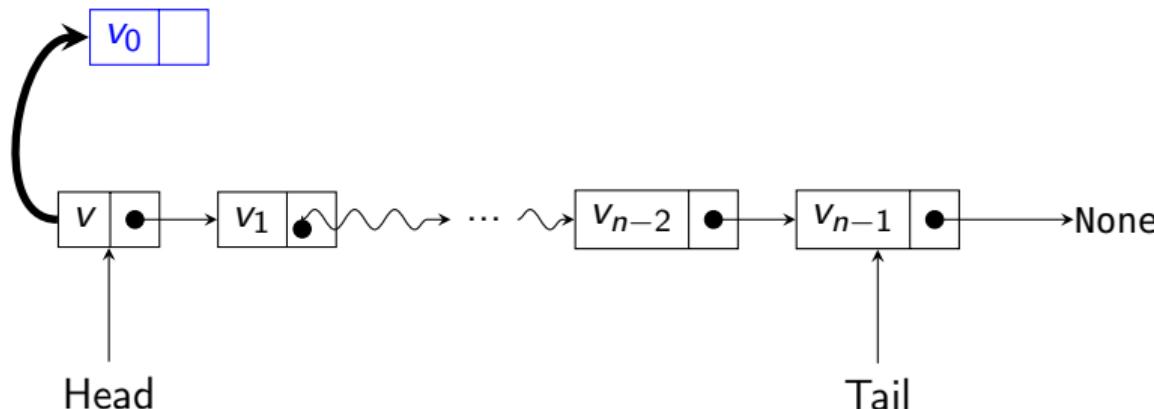
Insert at the start of the list

- Want to insert v at head
- Create a new node with v
- Cannot change where the head points!



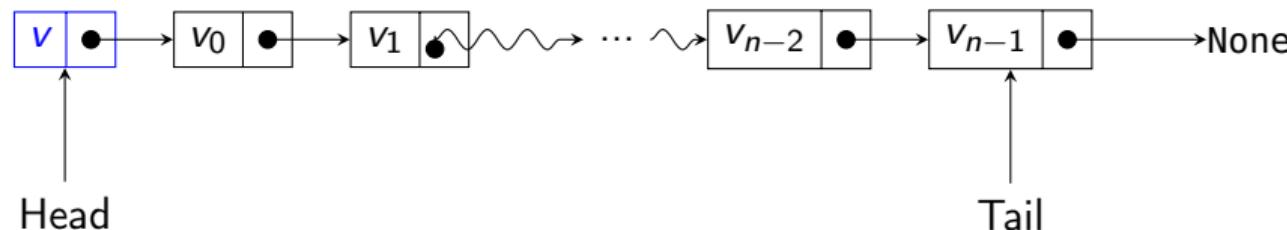
Insert at the start of the list

- Want to insert v at head
- Create a new node with v
- Cannot change where the head points!
- Exchange the values v_0, v



Insert at the start of the list

- Want to insert v at head
- Create a new node with v
- Cannot change where the head points!
- Exchange the values v_0, v
- Make new node point to `head.next`
- Make `head.next` point to new node



Appending to a list

- Create a new node with v
- Exchange the values v_0, v
- Make new node point to `head.next`
- Make `head.next` point to new node

```
def insert(self,v):  
    if selfisempty():  
        self.value = v  
        return  
  
    newnode = Node(v)  
  
    # Exchange values in self and newnode  
    (self.value, newnode.value) =  
        (newnode.value, self.value)  
  
    # Switch links  
    (self.next, newnode.next) =  
        (newnode, self.next)  
  
    return
```

Delete a value v

- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v , bypass it

Delete a value v

- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v , bypass it
- Cannot bypass the first node in the list
 - Instead, copy the second node value to head
 - Bypass second node

Delete a value v

- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v , bypass it
- Cannot bypass the first node in the list
 - Instead, copy the second node value to head
 - Bypass second node
- Recursive implementation

```
def delete(self,v):  
    # delete, recursive  
  
    if self.isempty():  
        return  
  
    if self.value == v:  
        self.value = None  
        if self.next != None:  
            self.value = self.next.value  
            self.next = self.next.next  
        return  
    else:  
        if self.next != None:  
            self.next.delete(v)  
        if self.next.value == None:  
            self.next = None  
  
    return
```

Delete a value v

- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v , bypass it
- Cannot bypass the first node in the list
 - Instead, copy the second node value to head
 - Bypass second node
- Recursive implementation
- Exercise: write an iterative version

```
def delete(self,v):  
    # delete, recursive  
  
    if self.isempty():  
        return  
  
    if self.value == v:  
        self.value = None  
        if self.next != None:  
            self.value = self.next.value  
            self.next = self.next.next  
        return  
    else:  
        if self.next != None:  
            self.next.delete(v)  
        if self.next.value == None:  
            self.next = None  
  
    return
```

Summary

- Use a linked list of nodes to implement a flexible list
- Append is easy
- Insert requires some care, cannot change where the head points to
- When deleting, look one step ahead to bypass the node to be deleted

Lists and Arrays in Python

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 3

Lists and arrays in Python

- Sequences can be stored as lists or arrays
- Lists are flexible but accessing an element is $O(n)$
- Arrays support random access but are difficult to expand, contract
- Algorithm analysis needs to take into account the underlying implementation
- How does it work in Python?
 - Is the built-in list type in Python really a “linked” list?
 - Numpy library provides arrays — are these faster than lists?

Lists in Python

- Python lists are **not** implemented as flexible linked lists

Lists in Python

- Python lists are **not** implemented as flexible linked lists
- Underlying interpretation maps the list to an array
 - Assign a fixed block when you create a list
 - Double the size if the list overflows the array

Lists in Python

- Python lists are **not** implemented as flexible linked lists
- Underlying interpretation maps the list to an array
 - Assign a fixed block when you create a list
 - Double the size if the list overflows the array
- Keep track of the last position of the list in the array
 - `l.append()` and `l.pop()` are constant time, amortised — $O(1)$
 - Insertion/deletion require time $O(n)$

Lists in Python

- Python lists are **not** implemented as flexible linked lists
- Underlying interpretation maps the list to an array
 - Assign a fixed block when you create a list
 - Double the size if the list overflows the array
- Keep track of the last position of the list in the array
 - `l.append()` and `l.pop()` are constant time, amortised — $O(1)$
 - Insertion/deletion require time $O(n)$
- Effectively, Python lists behave more like arrays than lists

Arrays vs lists in Python

- Arrays are useful for representing matrices

- In list notation, these are nested lists

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

`[[0,1], [1,0]]`

Arrays vs lists in Python

- Arrays are useful for representing matrices
- In list notation, these are nested lists
$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad [[0,1], [1,0]]$$
- Need to be careful when initializing a multidimensional list

```
zerolist = [0,0,0]
zeromatrix = [zerolist,zerolist,zerolist]
```

Arrays vs lists in Python

- Arrays are useful for representing matrices
- In list notation, these are nested lists
$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad [[0,1], [1,0]]$$
- Need to be careful when initializing a multidimensional list

```
zerolist = [0,0,0]
zeromatrix = [zerolist,zerolist,zerolist]

zeromatrix[1][1] = 1
print(zeromatrix)
```

Arrays vs lists in Python

- Arrays are useful for representing matrices
- In list notation, these are nested lists
$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad [[0,1], [1,0]]$$
- Need to be careful when initializing a multidimensional list

```
zerolist = [0,0,0]
zeromatrix = [zerolist,zerolist,zerolist]

zeromatrix[1][1] = 1
print(zeromatrix)

[[0, 1, 0], [0, 1, 0], [0, 1, 0]]
```

- Mutability **aliases** different values

Arrays vs lists in Python

- Arrays are useful for representing matrices

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

`[[0,1], [1,0]]`

- In list notation, these are nested lists
- Need to be careful when initializing a multidimensional list

```
zerolist = [0,0,0]
zeromatrix = [zerolist,zerolist,zerolist]
```

```
zeromatrix[1][1] = 1
print(zeromatrix)
```

```
[[0, 1, 0], [0, 1, 0], [0, 1, 0]]
```

- Mutability **aliases** different values
- Instead, use list comprehension

```
zeromatrix = [ [ 0 for i in range(3) ] for j in range(3) ]
```

Numpy arrays

- The Numpy library provides arrays as a basic type

```
import numpy as np  
zeromatrix = np.zeros(shape=(3,3))
```

Numpy arrays

- The Numpy library provides arrays as a basic type

```
import numpy as np  
zeromatrix = np.zeros(shape=(3,3))
```

- Can create an array from any sequence type

```
identitymatrix = np.array([[1,0],[0,1]])
```

Numpy arrays

- The Numpy library provides arrays as a basic type

```
import numpy as np  
zeromatrix = np.zeros(shape=(3,3))
```

- Can create an array from any sequence type

```
identitymatrix = np.array([[1,0],[0,1]])
```

- `arange` is the equivalent of `range` for lists

```
row2 = np.arange(5)
```

Numpy arrays

- The Numpy library provides arrays as a basic type

```
import numpy as np  
zeromatrix = np.zeros(shape=(3,3))
```

- Can create an array from any sequence type

```
identitymatrix = np.array([[1,0],[0,1]])
```

- `arange` is the equivalent of `range` for lists

```
row2 = np.arange(5)
```

- Can operate on a matrix as a whole

- `C = 3*A + B`
- `C = np.matmul(A,B)`
- Very useful for data science

Summary

- Python lists are not implemented as flexible linked structures
- Instead, allocate an array, and double space as needed
- Append is cheap, insert is expensive
- Arrays can be represented as multidimensional lists, but need to be careful about mutability, aliasing
- Numpy arrays are easier to use

Implementing dictionaries

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 3

Dictionary

- An array/list allows access through positional indices

Dictionary

- An array/list allows access through positional indices
- A dictionary allows access through arbitrary **keys**
 - A collection of key-value pairs
 - Random access — access time is the same for all keys

Dictionary

- An array/list allows access through positional indices
- A dictionary allows access through arbitrary **keys**
 - A collection of key-value pairs
 - Random access — access time is the same for all keys
- How is a dictionary implemented?

Implementing a dictionary

- The underlying storage is an array
 - Given an offset i , find $A[i]$ in constant time

Implementing a dictionary

- The underlying storage is an array
 - Given an offset i , find $A[i]$ in constant time
- Keys have to be mapped to $\{0, 1, \dots, n - 1\}$
 - Given a key k , convert it to an offset i

Implementing a dictionary

- The underlying storage is an array
 - Given an offset i , find $A[i]$ in constant time
- Keys have to be mapped to $\{0, 1, \dots, n - 1\}$
 - Given a key k , convert it to an offset i
- Hash function
 - $h : S \rightarrow X$ maps a set of values S to a small range of integers $X = \{0, 1, \dots, n - 1\}$

Implementing a dictionary

- The underlying storage is an array
 - Given an offset i , find $A[i]$ in constant time
- Keys have to be mapped to $\{0, 1, \dots, n - 1\}$
 - Given a key k , convert it to an offset i
- Hash function
 - $h : S \rightarrow X$ maps a set of values S to a small range of integers $X = \{0, 1, \dots, n - 1\}$
 - Typically $|X| \ll |S|$, so there will be **collisions**, $h(s) = h(s'), s \neq s'$
 - A good hash function will minimize collisions

Implementing a dictionary

- The underlying storage is an array
 - Given an offset i , find $A[i]$ in constant time
- Keys have to be mapped to $\{0, 1, \dots, n - 1\}$
 - Given a key k , convert it to an offset i
- Hash function
 - $h : S \rightarrow X$ maps a set of values S to a small range of integers $X = \{0, 1, \dots, n - 1\}$
 - Typically $|X| \ll |S|$, so there will be collisions, $h(s) = h(s'), s \neq s'$
 - A good hash function will minimize collisions
 - SHA-256 is an industry standard hashing function whose range is 256 bits
 - Use to hash large files — avoid uploading duplicates to cloud storage

Hash table

- An array A of size n combined with a hash function h

Hash table

- An array A of size n combined with a hash function h
- h maps keys to $\{0, 1, \dots, n - 1\}$

Hash table

- An array A of size n combined with a hash function h
- h maps keys to $\{0, 1, \dots, n - 1\}$
- Ideally, when we create an entry for key k , $A[h(k)]$ will be unused

Hash table

- An array A of size n combined with a hash function h
- h maps keys to $\{0, 1, \dots, n - 1\}$
- Ideally, when we create an entry for key k , $A[h(k)]$ will be unused
 - What if there is already a value at that location?
- Dealing with collisions

Hash table

- An array A of size n combined with a hash function h
- h maps keys to $\{0, 1, \dots, n - 1\}$
- Ideally, when we create an entry for key k , $A[h(k)]$ will be unused
 - What if there is already a value at that location?
- Dealing with collisions
 - Open addressing (closed hashing)
 - Probe a sequence of alternate slots in the same array

Hash table

- An array A of size n combined with a hash function h
- h maps keys to $\{0, 1, \dots, n - 1\}$
- Ideally, when we create an entry for key k , $A[h(k)]$ will be unused
 - What if there is already a value at that location?
- Dealing with collisions
 - Open addressing (closed hashing)
 - Probe a sequence of alternate slots in the same array
 - Open hashing
 - Each slot in the array points to a list of values
 - Insert into the list for the given slot

Hash table

- An array A of size n combined with a hash function h
- h maps keys to $\{0, 1, \dots, n - 1\}$
- Ideally, when we create an entry for key k , $A[h(k)]$ will be unused
 - What if there is already a value at that location?
- Dealing with collisions
 - Open addressing (closed hashing)
 - Probe a sequence of alternate slots in the same array
 - Open hashing
 - Each slot in the array points to a list of values
 - Insert into the list for the given slot
- Dictionary keys in Python must be immutable
 - If value changes, hash also changes!

- A dictionary is implemented as a hash table
 - An array plus a hash function
- Creating a good hash function is important (and hard!)
- Need a strategy to deal with collisions
 - Open addressing/closed hashing — probe for free space in the array
 - Open hashing — each slot in the hash table points to a list of key-value pairs
 - Many heuristics/optimizations possible for dea

Graphs

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

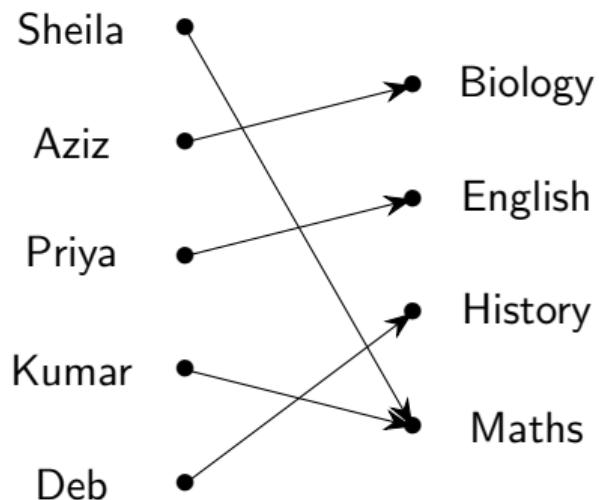
Week 4

Visualizing relations as graphs

■ Teachers and courses

- T , set of teachers in a college
- C , set of courses being offered
- $A \subseteq T \times C$ describes the allocation of teachers to courses
- $A = \{(t, c) \mid (t, c) \in T \times C, t \text{ teaches } c\}$

Teachers and courses



Visualizing relations as graphs

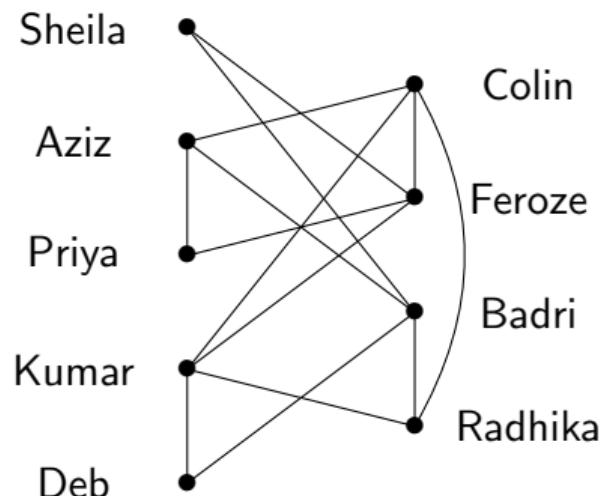
- Teachers and courses

- T , set of teachers in a college
- C , set of courses being offered
- $A \subseteq T \times C$ describes the allocation of teachers to courses
- $A = \{(t, c) \mid (t, c) \in T \times C, t \text{ teaches } c\}$

- Friendships

- P , a set of students
- $F \subseteq P \times P$ describes which pairs of students are friends
- $F = \{(p, q) \mid p, q \in P, p \neq q, p \text{ is a friend of } q\}$
- $(p, q) \in F$ iff $(q, p) \in F$

Friendship



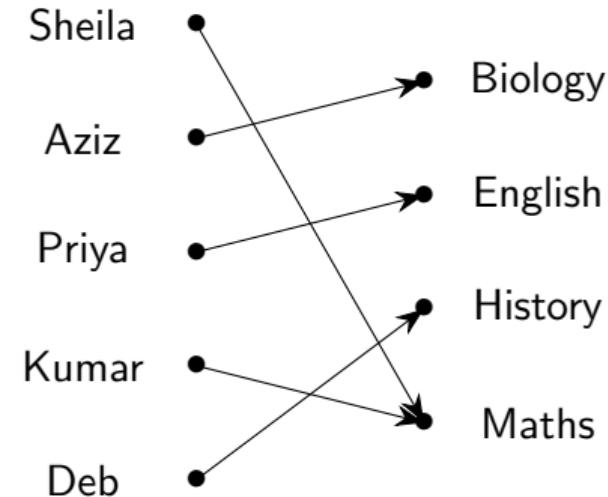
Graphs

- Graph: $G = (V, E)$
 - V is a set of **vertices** or **nodes**
 - One vertex, many vertices
 - E is a set of **edges**
 - $E \subseteq V \times V$ — binary relation

Graphs

- Graph: $G = (V, E)$
 - V is a set of vertices or nodes
 - One vertex, many vertices
 - E is a set of edges
 - $E \subseteq V \times V$ — binary relation
- Directed graph
 - $(v, v') \in E$ does not imply $(v', v) \in E$
 - The teacher-course graph is directed

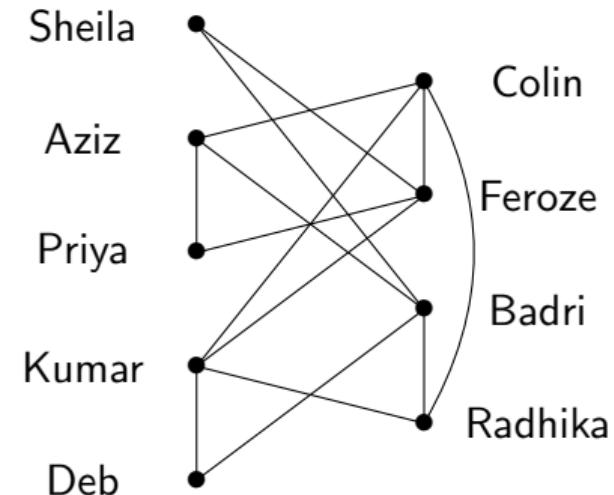
Teachers and courses



Graphs

- Graph: $G = (V, E)$
 - V is a set of vertices or nodes
 - One vertex, many vertices
 - E is a set of edges
 - $E \subseteq V \times V$ — binary relation
- Directed graph
 - $(v, v') \in E$ does not imply $(v', v) \in E$
 - The teacher-course graph is directed
- Undirected graph
 - $(v, v') \in E$ iff $(v', v) \in E$
 - Effectively (v, v') , (v', v) are the same edge
 - Friendship graph is undirected

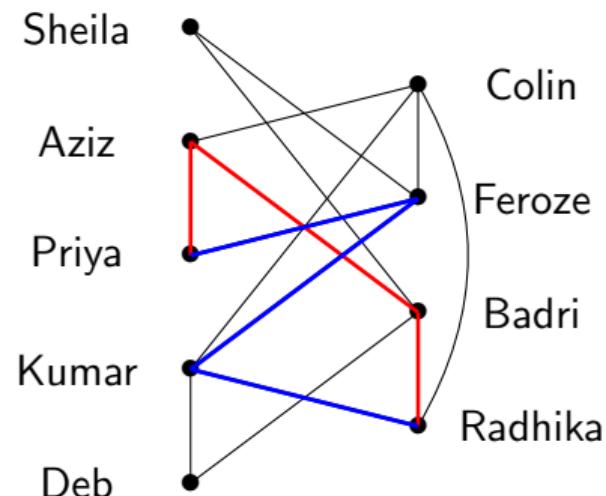
Friendship



Paths

- A **path** is a sequence of vertices v_1, v_2, \dots, v_k connected by edges
 - For $1 \leq i < k$, $(v_i, v_{i+1}) \in E$

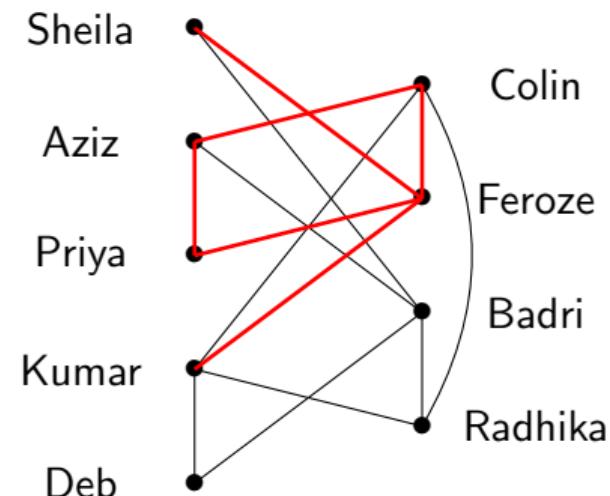
Friendship graph



Paths

- A **path** is a sequence of vertices v_1, v_2, \dots, v_k connected by edges
 - For $1 \leq i < k$, $(v_i, v_{i+1}) \in E$
- Normally, a path does not visit a vertex twice
- A sequence that re-visits a vertex is usually called a **walk**
 - Kumar — Feroze — Colin — Aziz — Priya — Feroze — Sheila

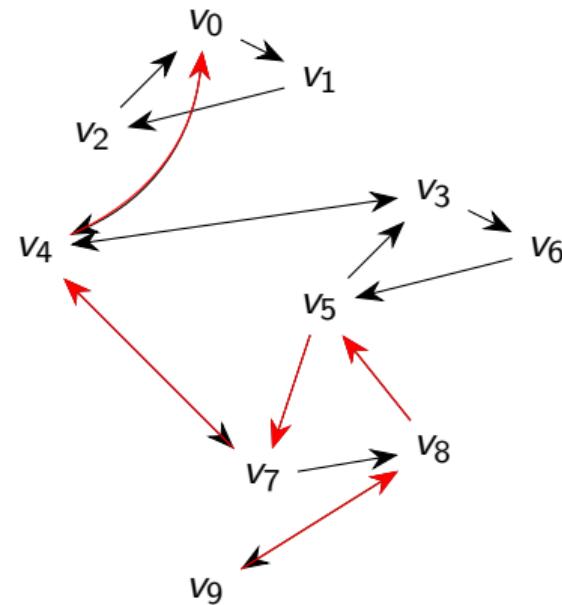
Friendship graph



Reachability

- Paths in directed graphs
- How can I fly from Madurai to Delhi?
 - Find a path from v_9 to v_0
- Vertex v is **reachable** from vertex u if there is a path from u to v

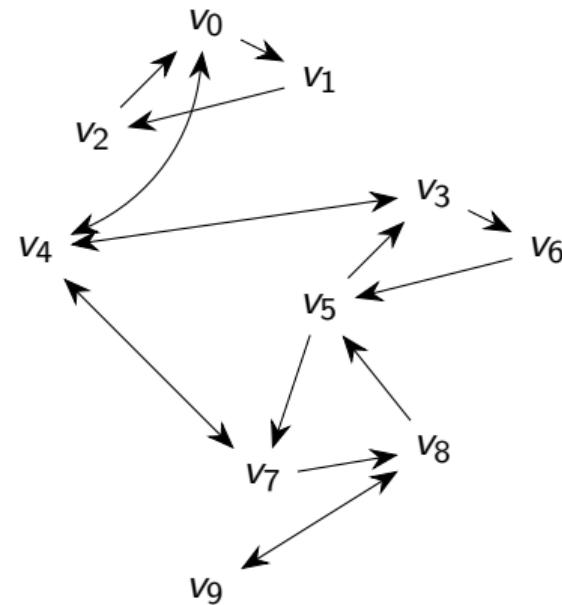
Airline routes



Reachability

- Paths in directed graphs
- How can I fly from Madurai to Delhi?
 - Find a path from v_9 to v_0
- Vertex v is **reachable** from vertex u if there is a path from u to v
- Typical questions
 - Is v reachable from u ?
 - What is the shortest path from u to v ?
 - What are the vertices reachable from u ?
 - Is the graph **connected**? Are all vertices reachable from each other?

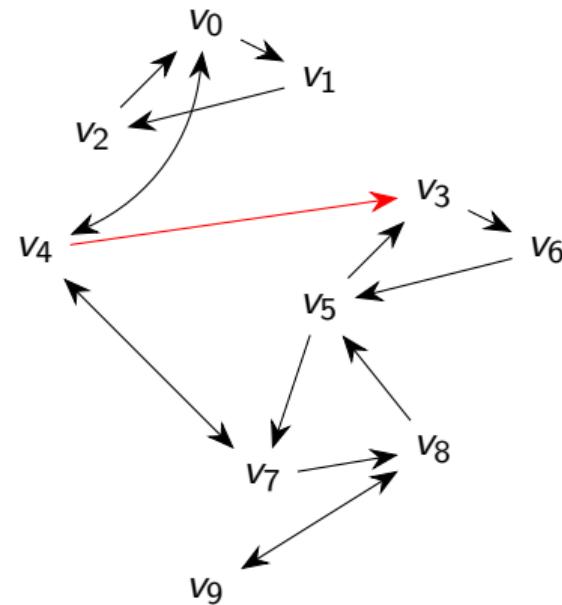
Airline routes



Reachability

- Paths in directed graphs
- How can I fly from Madurai to Delhi?
 - Find a path from v_9 to v_0
- Vertex v is **reachable** from vertex u if there is a path from u to v
- Typical questions
 - Is v reachable from u ?
 - What is the shortest path from u to v ?
 - What are the vertices reachable from u ?
 - Is the graph **connected**? Are all vertices reachable from each other?

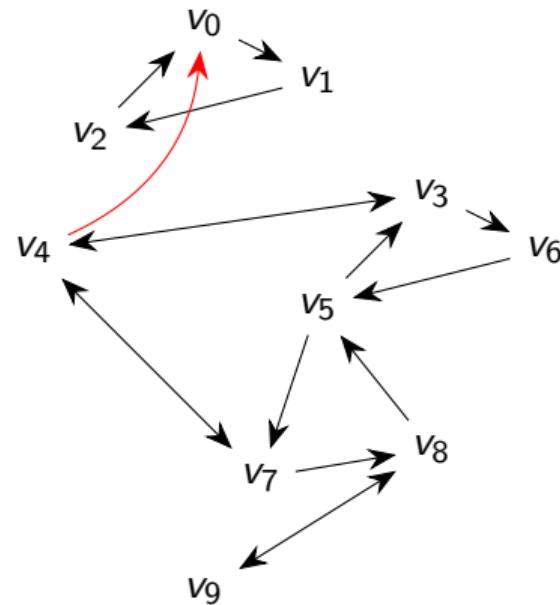
Airline routes



Reachability

- Paths in directed graphs
- How can I fly from Madurai to Delhi?
 - Find a path from v_9 to v_0
- Vertex v is **reachable** from vertex u if there is a path from u to v
- Typical questions
 - Is v reachable from u ?
 - What is the shortest path from u to v ?
 - What are the vertices reachable from u ?
 - Is the graph **connected**? Are all vertices reachable from each other?

Airline routes



Map colouring

- Assign each state a colour
 - States that share a border should be coloured differently
 - How many colours do we need?



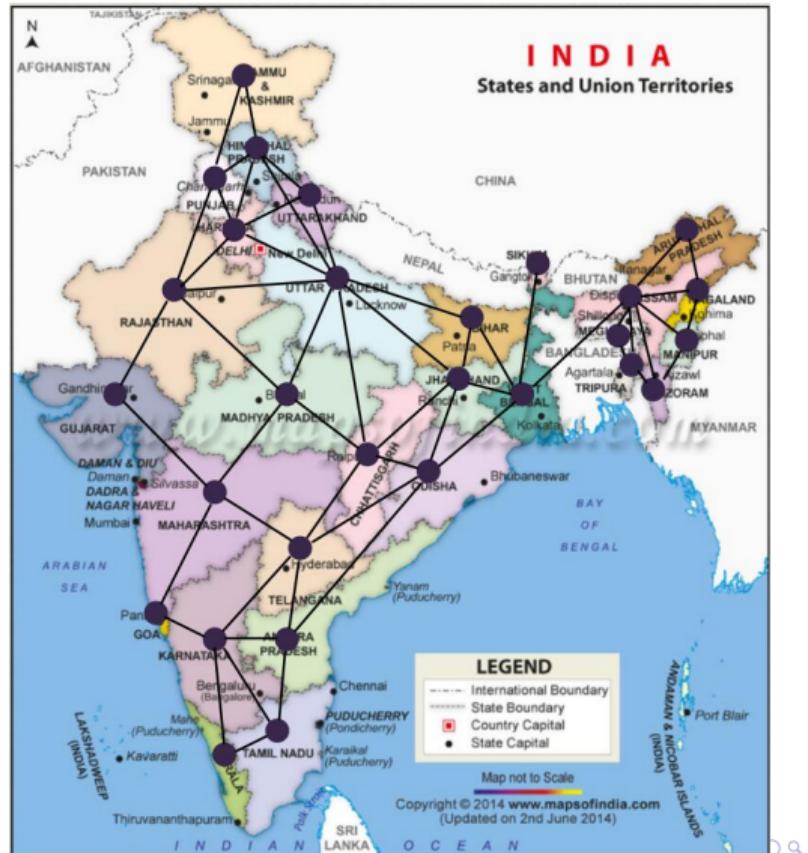
Map colouring

- Assign each state a colour
- States that share a border should be coloured differently
- How many colours do we need?
- Create a graph
 - Each state is a vertex



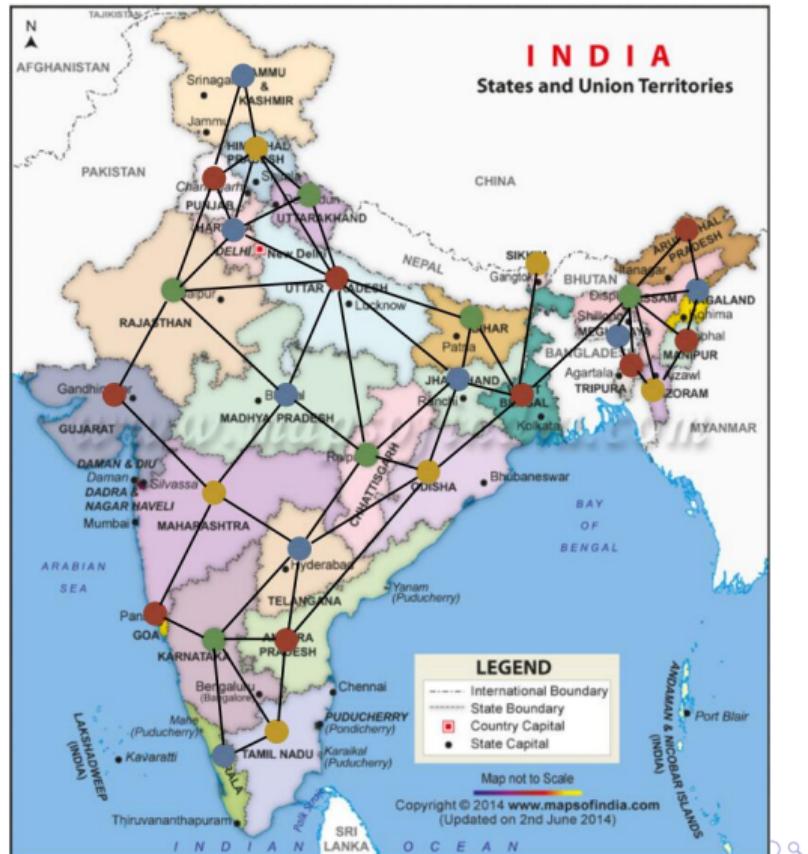
Map colouring

- Assign each state a colour
- States that share a border should be coloured differently
- How many colours do we need?
- Create a graph
 - Each state is a vertex
 - Connect states that share a border



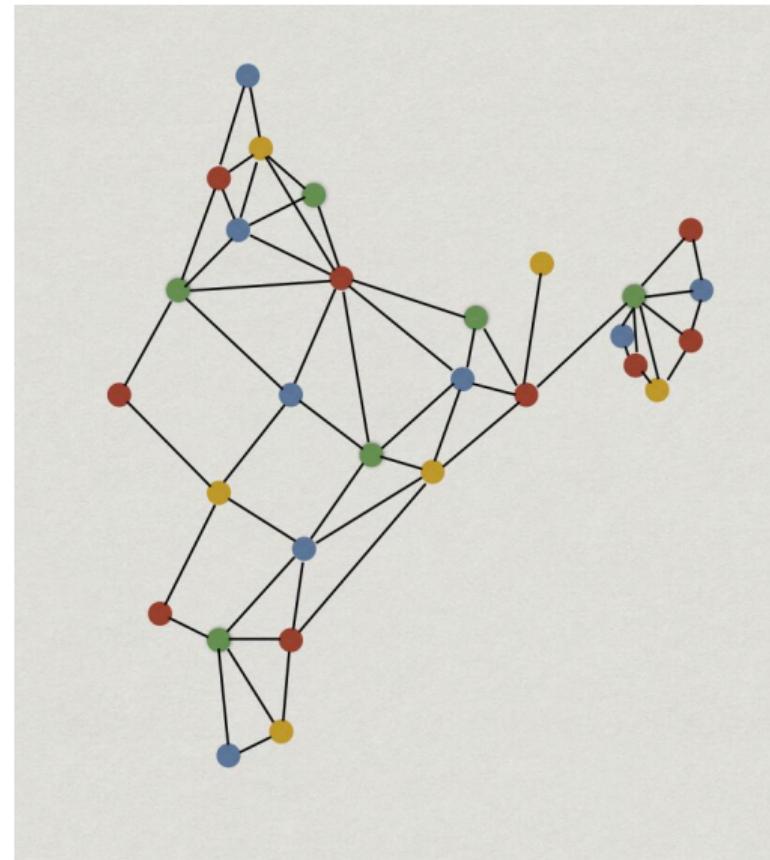
Map colouring

- Assign each state a colour
- States that share a border should be coloured differently
- How many colours do we need?
- Create a graph
 - Each state is a vertex
 - Connect states that share a border
- Assign colours to nodes so that endpoints of an edge have different colours



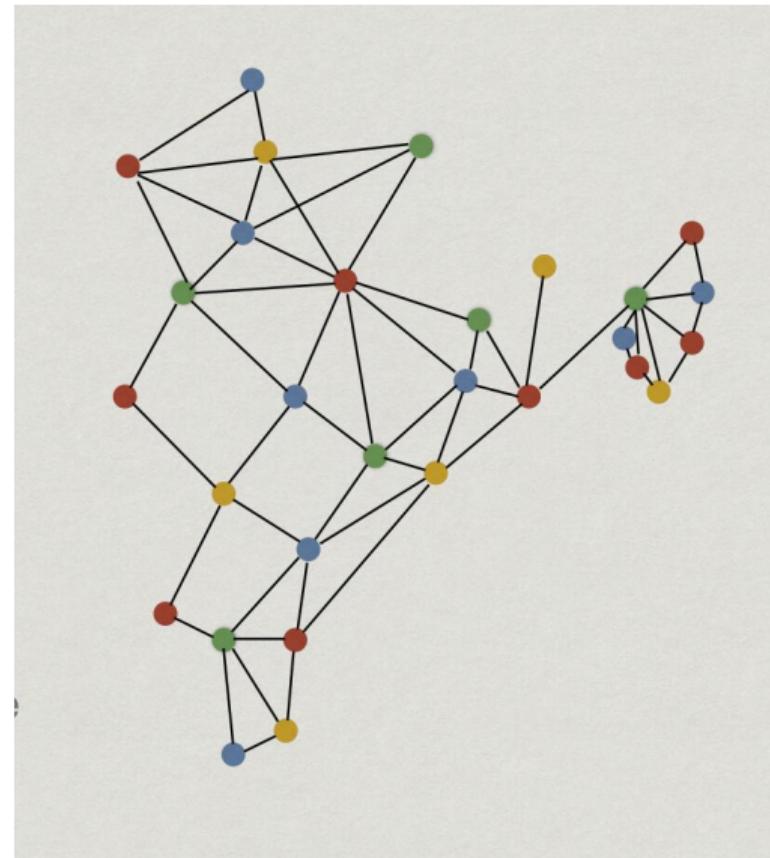
Map colouring

- Assign each state a colour
- States that share a border should be coloured differently
- How many colours do we need?
- Create a graph
 - Each state is a vertex
 - Connect states that share a border
- Assign colours to nodes so that endpoints of an edge have different colours
- Only need the underlying graph



Map colouring

- Assign each state a colour
- States that share a border should be coloured differently
- How many colours do we need?
- Create a graph
 - Each state is a vertex
 - Connect states that share a border
- Assign colours to nodes so that endpoints of an edge have different colours
- Only need the underlying graph
- Abstraction: if we distort the graph, problem is unchanged



Graph colouring

- Graph $G = (V, E)$, set of colours C
- Colouring is a function $c : V \rightarrow C$ such that $(u, v) \in E \Rightarrow c(u) \neq c(v)$
- Given $G = (V, E)$, what is the smallest set of colours need to colour G

Graph colouring

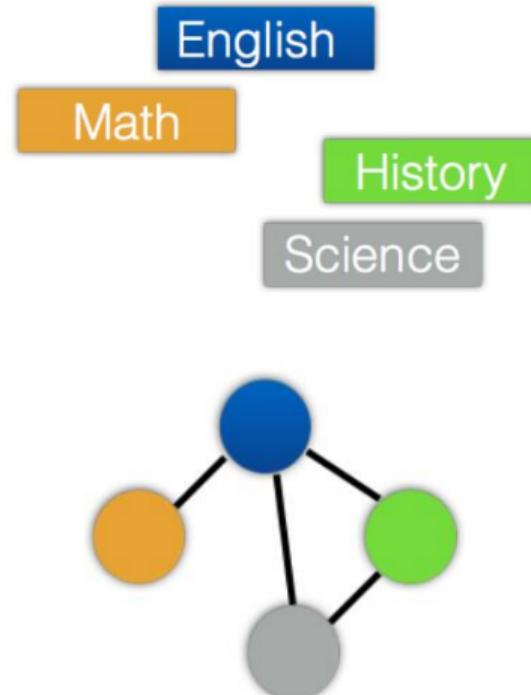
- Graph $G = (V, E)$, set of colours C
- Colouring is a function $c : V \rightarrow C$ such that $(u, v) \in E \Rightarrow c(u) \neq c(v)$
- Given $G = (V, E)$, what is the smallest set of colours need to colour G
 - **Four Colour Theorem** For planar graphs derived from geographical maps, 4 colours suffice

Graph colouring

- Graph $G = (V, E)$, set of colours C
- Colouring is a function $c : V \rightarrow C$ such that $(u, v) \in E \Rightarrow c(u) \neq c(v)$
- Given $G = (V, E)$, what is the smallest set of colours need to colour G
 - **Four Colour Theorem** For planar graphs derived from geographical maps, 4 colours suffice
 - Not all graphs are **planar**. General case?
Why do we care?

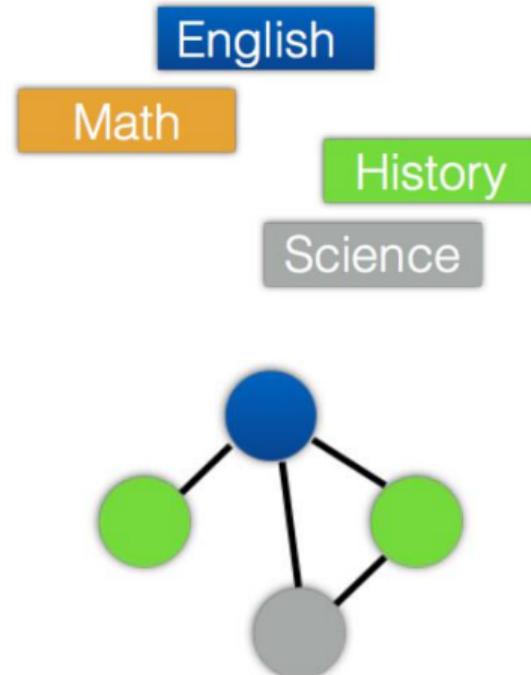
Graph colouring

- Graph $G = (V, E)$, set of colours C
- Colouring is a function $c : V \rightarrow C$ such that $(u, v) \in E \Rightarrow c(u) \neq c(v)$
- Given $G = (V, E)$, what is the smallest set of colours need to colour G
 - **Four Colour Theorem** For planar graphs derived from geographical maps, 4 colours suffice
 - Not all graphs are **planar**. General case? Why do we care?
- How many classrooms do we need?
 - Courses and timetable slots, edges represent overlapping slots
 - Colours are classrooms



Graph colouring

- Graph $G = (V, E)$, set of colours C
- Colouring is a function $c : V \rightarrow C$ such that $(u, v) \in E \Rightarrow c(u) \neq c(v)$
- Given $G = (V, E)$, what is the smallest set of colours need to colour G
 - **Four Colour Theorem** For planar graphs derived from geographical maps, 4 colours suffice
 - Not all graphs are **planar**. General case? Why do we care?
- How many classrooms do we need?
 - Courses and timetable slots, edges represent overlapping slots
 - Colours are classrooms

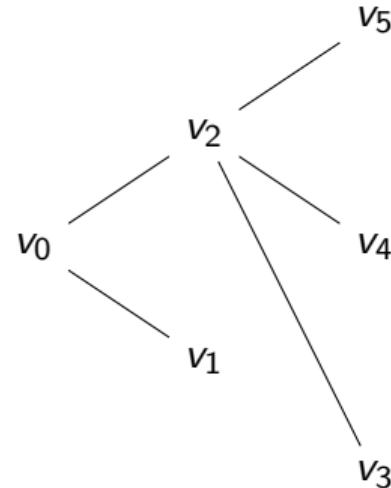


Vertex cover

- A hotel wants to install security cameras
 - All corridors are straight lines
 - Camera can monitor all corridors that meet at an intersection
- Minimum number of cameras needed?

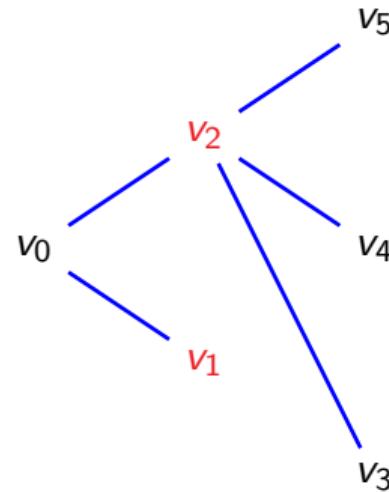
Vertex cover

- A hotel wants to install security cameras
 - All corridors are straight lines
 - Camera can monitor all corridors that meet at an intersection
- Minimum number of cameras needed?
- Represent the floor plan as a graph
 - V — intersections of corridors
 - E — corridor segments connecting intersections



Vertex cover

- A hotel wants to install security cameras
 - All corridors are straight lines
 - Camera can monitor all corridors that meet at an intersection
- Minimum number of cameras needed?
- Represent the floor plan as a graph
 - V — intersections of corridors
 - E — corridor segments connecting intersections
- Vertex cover
 - Marking v covers all edges from v
 - Mark smallest subset of V to cover all edges

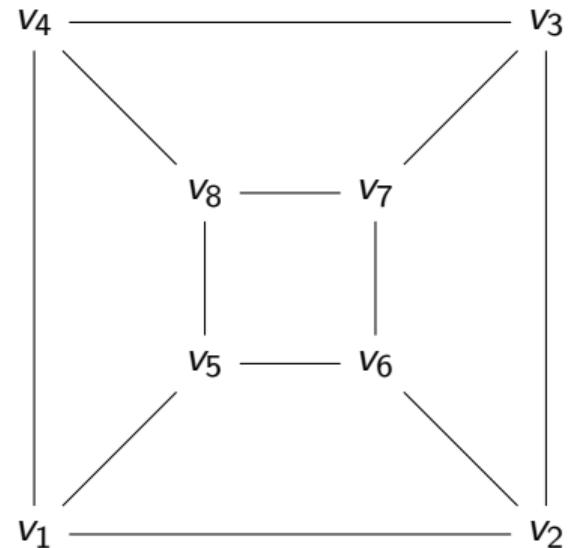


Independent set

- A dance school puts up group dances
 - Each dance has a set of dancers
 - Sets of dancers may overlap across dances
- Organizing a cultural programme
 - Each dancer performs at most once
 - Maximum number of dances possible?

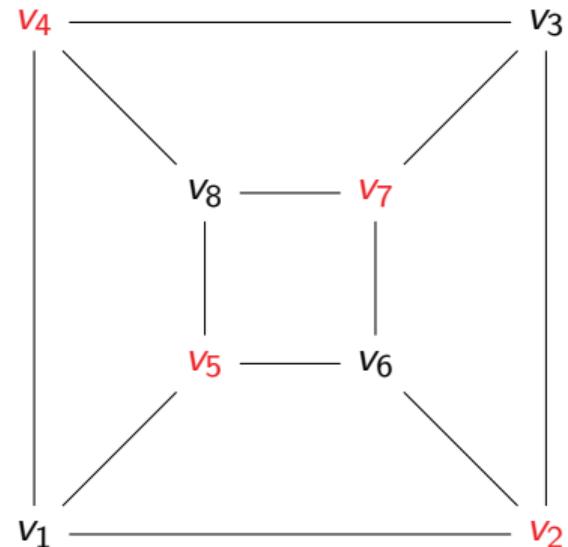
Independent set

- A dance school puts up group dances
 - Each dance has a set of dancers
 - Sets of dancers may overlap across dances
- Organizing a cultural programme
 - Each dancer performs at most once
 - Maximum number of dances possible?
- Represent the dances as a graph
 - V — dances
 - E — sets of dancers overlap



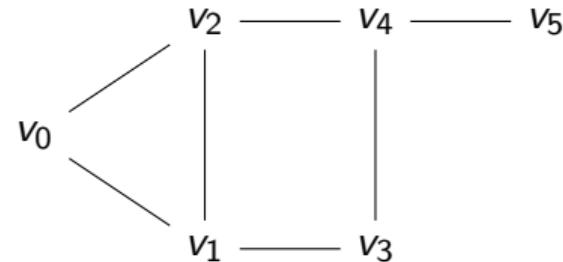
Independent set

- A dance school puts up group dances
 - Each dance has a set of dancers
 - Sets of dancers may overlap across dances
- Organizing a cultural programme
 - Each dancer performs at most once
 - Maximum number of dances possible?
- Represent the dances as a graph
 - V — dances
 - E — sets of dancers overlap
- Independent set
 - Subset of vertices such that no two are connected by an edge



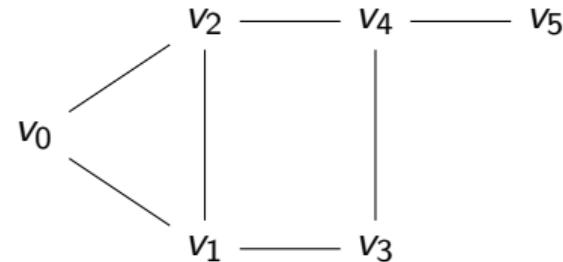
Matching

- Class project can be done by one or two people
 - If two people, they must be friends
- Assume we have a graph describing friendships



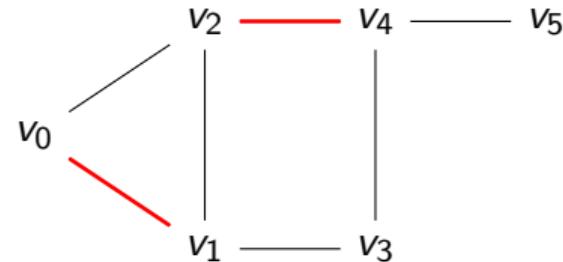
Matching

- Class project can be done by one or two people
 - If two people, they must be friends
- Assume we have a graph describing friendships
- Find a good allocation of groups
- Matching
 - $G = (V, E)$, an undirected graph
 - A matching is a subset $M \subseteq E$ of mutually disjoint edges



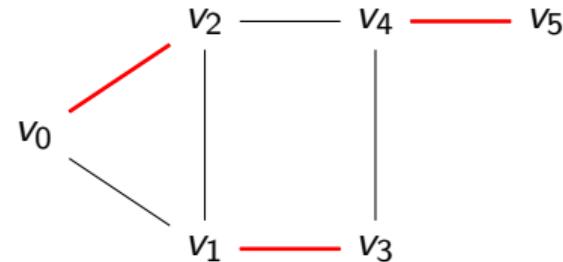
Matching

- Class project can be done by one or two people
 - If two people, they must be friends
- Assume we have a graph describing friendships
- Find a good allocation of groups
- Matching
 - $G = (V, E)$, an undirected graph
 - A matching is a subset $M \subseteq E$ of mutually disjoint edges
- Find a maximal matching in G



Matching

- Class project can be done by one or two people
 - If two people, they must be friends
- Assume we have a graph describing friendships
- Find a good allocation of groups
- Matching
 - $G = (V, E)$, an undirected graph
 - A matching is a subset $M \subseteq E$ of mutually disjoint edges
- Find a maximal matching in G
- Is there a perfect matching, covering all vertices?



Summary

- A graph represents relationships between entities
 - Entities are vertices/nodes
 - Relationships are edges
- A graph may be directed or undirected
 - A is a parent of B — directed
 - A is a friend of B — undirected

Summary

- A graph represents relationships between entities
 - Entities are vertices/nodes
 - Relationships are edges
- A graph may be directed or undirected
 - A is a parent of B — directed
 - A is a friend of B — undirected
- Paths are sequences of connected edges
- Reachability: is there a path from u to v ?

Summary

- A graph represents relationships between entities
 - Entities are vertices/nodes
 - Relationships are edges
- A graph may be directed or undirected
 - A is a parent of B — directed
 - A is a friend of B — undirected
- Paths are sequences of connected edges
- Reachability: is there a path from u to v ?
- Graphs are useful abstract representations for a wide range of problems
- Reachability and connectedness are not the only interesting problems we can solve on graphs
 - Graph colouring
 - Vertex cover
 - Independent set
 - Matching
 - ...

Representing Graphs

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

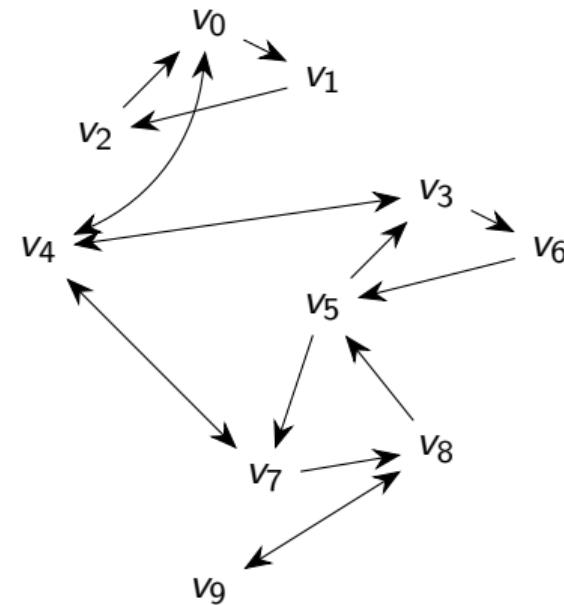
Programming, Data Structures and Algorithms using Python

Week 4

Working with graphs

- Graph $G = (V, E)$
 - V — set of vertices
 - $E \subseteq V \times V$ — set of edges
- A **path** is a sequence of vertices v_1, v_2, \dots, v_k connected by edges
 - For $1 \leq i < k$, $(v_i, v_{i+1}) \in E$
- Vertex v is **reachable** from vertex u if there is a path from u to v

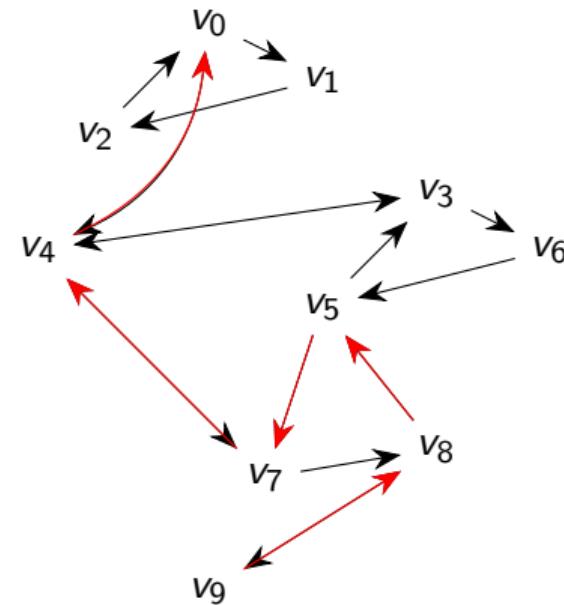
Airline routes



Working with graphs

- Graph $G = (V, E)$
 - V — set of vertices
 - $E \subseteq V \times V$ — set of edges
- A **path** is a sequence of vertices v_1, v_2, \dots, v_k connected by edges
 - For $1 \leq i < k$, $(v_i, v_{i+1}) \in E$
- Vertex v is **reachable** from vertex u if there is a path from u to v
- Looking at the picture of G , we can “see” that v_0 is reachable from v_9
- How do we represent this picture so that we can compute reachability?

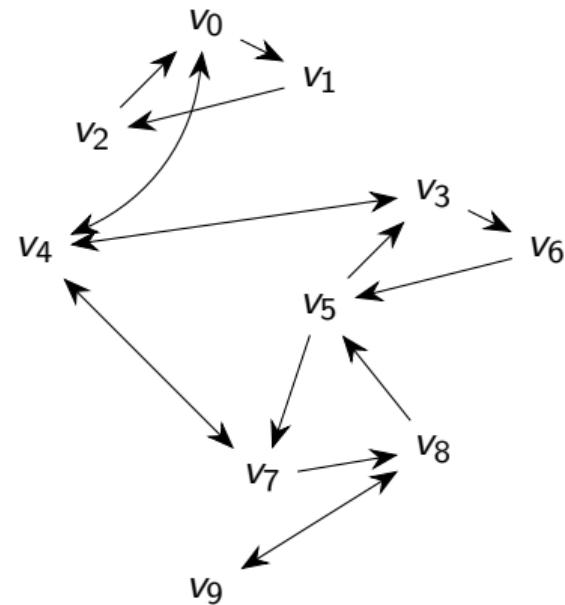
Airline routes



Adjacency matrix

- Let $|V| = n$
 - Assume $V = \{0, 1, \dots, n - 1\}$
 - Use a table to map actual vertex “names” to this set

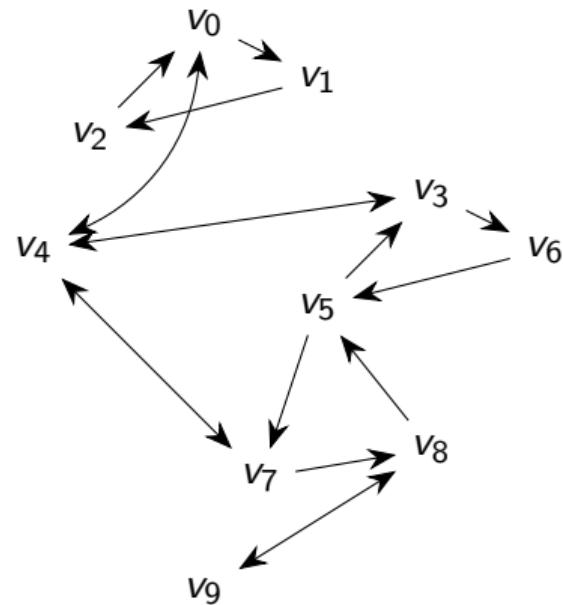
Airline routes



Adjacency matrix

- Let $|V| = n$
 - Assume $V = \{0, 1, \dots, n - 1\}$
 - Use a table to map actual vertex “names” to this set
- Edges are now pairs (i, j) , where $0 \leq i, j < n$
 - Usually assume $i \neq j$, no self loops

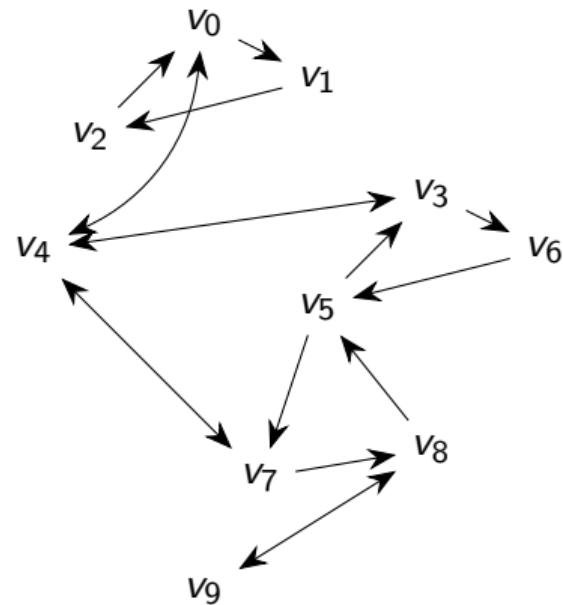
Airline routes



Adjacency matrix

- Let $|V| = n$
 - Assume $V = \{0, 1, \dots, n - 1\}$
 - Use a table to map actual vertex “names” to this set
- Edges are now pairs (i, j) , where $0 \leq i, j < n$
 - Usually assume $i \neq j$, no self loops
- Adjacency matrix
 - Rows and columns numbered $\{0, 1, \dots, n - 1\}$
 - $A[i, j] = 1$ if $(i, j) \in E$

Airline routes

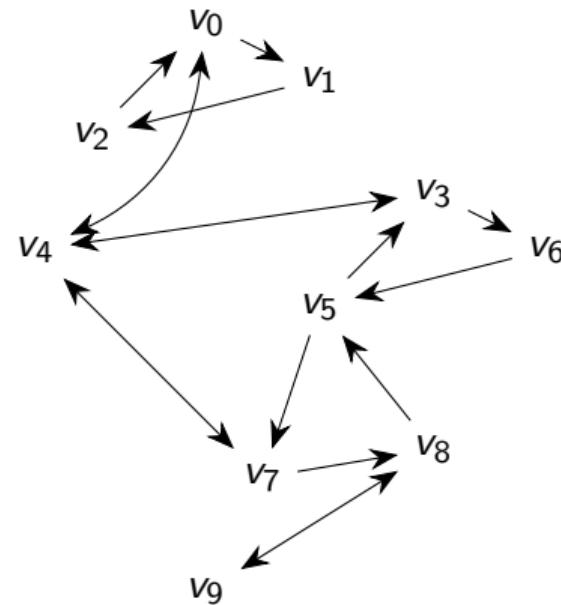


Adjacency matrix

- Adjacency matrix

- Rows and columns numbered $\{0, 1, \dots, n - 1\}$
- $A[i, j] = 1$ if $(i, j) \in E$

Airline routes



Adjacency matrix

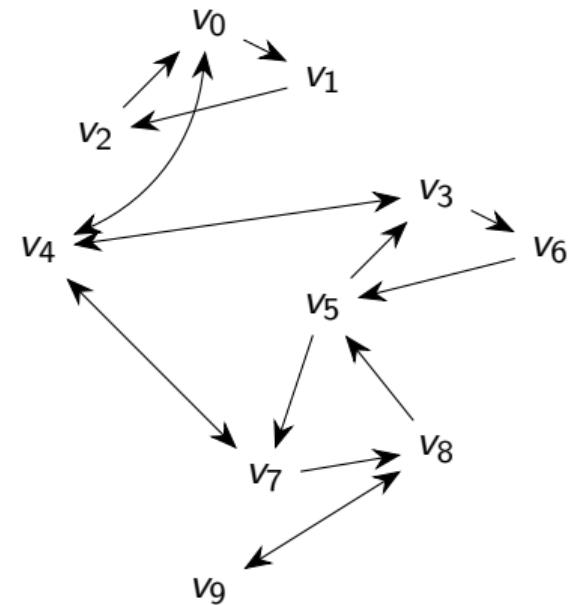
■ Adjacency matrix

- Rows and columns numbered $\{0, 1, \dots, n - 1\}$
- $A[i, j] = 1$ if $(i, j) \in E$

```
edges = [(0,1),(0,4),(1,2),(2,0),  
         (3,4),(3,6),(4,0),(4,3),  
         (4,7),(5,3),(5,7),  
         (6,5),(7,4),(7,8),  
         (8,5),(8,9),(9,8)]
```

```
import numpy as np  
A = np.zeros(shape=(10,10))  
  
for (i,j) in edges:  
    A[i,j] = 1
```

Airline routes



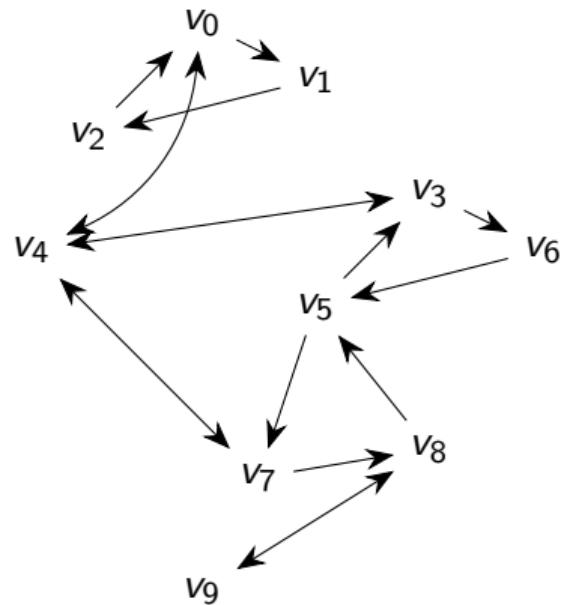
Adjacency matrix

■ Adjacency matrix

- Rows and columns numbered $\{0, 1, \dots, n - 1\}$
- $A[i, j] = 1$ if $(i, j) \in E$

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

Airline routes



Adjacency matrix

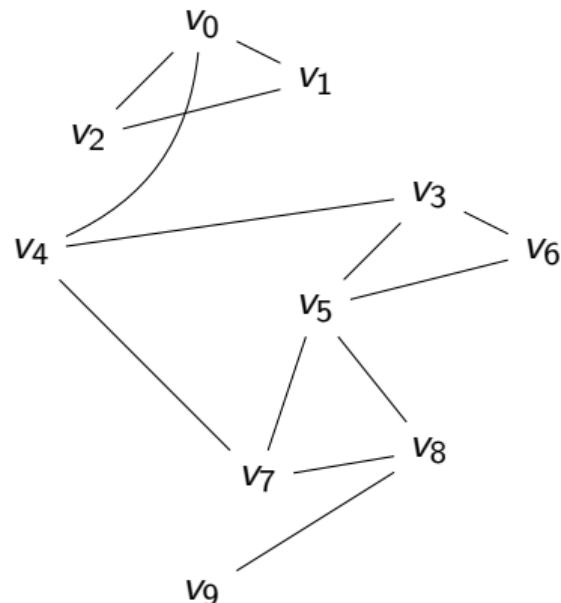
- Undirected graph

- $A[i,j] = 1$ iff $A[j,i] = 1$

- Symmetric across main diagonal

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Airline routes, all routes bidirectional



Computing with the adjacency matrix

- Neighbours of i — column j with entry 1
 - Scan row i to identify neighbours of i
 - Neighbours of 6 are [3, 5]

```
def neighbours(AMat,i):  
    nbrs = []  
    (rows,cols) = AMat.shape  
    for j in range(cols):  
        if AMat[i,j] == 1:  
            nbrs.append(j)  
    return(nbrs)  
  
neighbours(A,7)  
[4, 5, 8]
```

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Computing with the adjacency matrix

- Neighbours of i — column j with entry 1
 - Scan row i to identify neighbours of i
 - Neighbours of 6 are [3, 5]
- Directed graph

Directed airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

Computing with the adjacency matrix

- Neighbours of i — column j with entry 1
 - Scan row i to identify neighbours of i
 - Neighbours of 6 are [3, 5]
- Directed graph
 - Rows represent outgoing edges

Directed airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

Computing with the adjacency matrix

- Neighbours of i — column j with entry 1
 - Scan row i to identify neighbours of i
 - Neighbours of 6 are [3, 5]
- Directed graph
 - Rows represent outgoing edges
 - Columns represent incoming edges

Directed airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

Computing with the adjacency matrix

- Neighbours of i — column j with entry 1
 - Scan row i to identify neighbours of i
 - Neighbours of 6 are [3, 5]
- Directed graph
 - Rows represent outgoing edges
 - Columns represent incoming edges
- Degree of a vertex i
 - Number of edges incident on i
 $\text{degree}(6) = 2$

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Computing with the adjacency matrix

- Neighbours of i — column j with entry 1
 - Scan row i to identify neighbours of i
 - Neighbours of 6 are [3, 5]
- Directed graph
 - Rows represent outgoing edges
 - Columns represent incoming edges
- Degree of a vertex i
 - Number of edges incident on i
 $\text{degree}(6) = 2$
 - For directed graphs, outdegree and indegree
 $\text{indegree}(6) = 1$, $\text{outdegree}(6) = 1$

Directed airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable
- Systematically mark neighbours of marked vertices

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable
- Systematically mark neighbours of marked vertices

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable
- Systematically mark neighbours of marked vertices

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable
- Systematically mark neighbours of marked vertices

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable
- Systematically mark neighbours of marked vertices

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable
- Systematically mark neighbours of marked vertices

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable
- Systematically mark neighbours of marked vertices

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable
- Systematically mark neighbours of marked vertices
- Stop when 0 becomes marked

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Is Delhi (0) reachable from Madurai (9)?
- Mark 9 as reachable
- Mark each neighbour of 9 as reachable
- Systematically mark neighbours of marked vertices
- Stop when 0 becomes marked
- If marking process stops without target becoming marked, the target is unreachable

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked
- Need a strategy to systematically explore marked neighbours

Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Checking reachability

- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked
- Need a strategy to systematically explore marked neighbours
- Two primary strategies
 - Breadth first — propagate marks in “layers”
 - Depth first — explore a path till it dies out, then backtrack

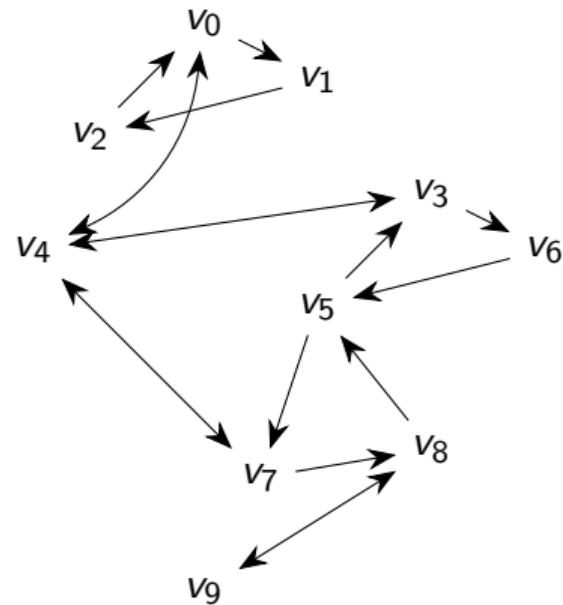
Undirected airline routes

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	1	1	1	0
6	0	0	0	1	0	1	0	0	0	0
7	0	0	0	0	1	1	0	0	1	0
8	0	0	0	0	0	1	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

Adjacency lists

- Adjacency matrix has many 0's
 - Size is n^2 , regardless of number of edges
 - Undirected graph: $|E| \leq n(n - 1)/2$
 - Directed graph: $|E| \leq n(n - 1)$
 - Typically $|E|$ much less than n^2

Airline routes



Adjacency lists

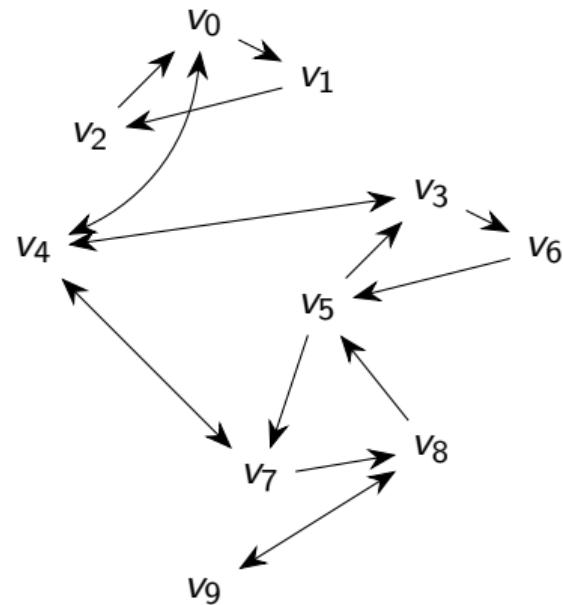
- Adjacency matrix has many 0's
 - Size is n^2 , regardless of number of edges
 - Undirected graph: $|E| \leq n(n - 1)/2$
 - Directed graph: $|E| \leq n(n - 1)$
 - Typically $|E|$ much less than n^2

- Adjacency list
 - List of neighbours for each vertex

0	[1,4]
1	[2]
2	[0]
3	[4,6]
4	[0,3,7]

5	[3,7]
6	[5]
7	[4,8]
8	[5,9]
9	[8]

Airline routes



Adjacency lists

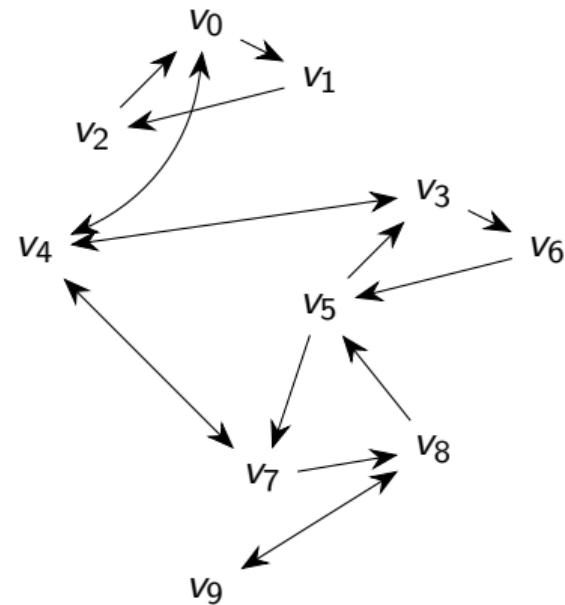
- Adjacency matrix has many 0's
 - Size is n^2 , regardless of number of edges
 - Undirected graph: $|E| \leq n(n - 1)/2$
 - Directed graph: $|E| \leq n(n - 1)$
 - Typically $|E|$ much less than n^2

■ Adjacency list

```
AList = []
for i in range(10):
    AList[i] = []
for (i,j) in edges:
    AList[i].append(j)

print(AList)
{0: [1, 4], 1: [2], 2: [0], 3: [4, 6], 4: [0, 3, 7],
5: [3, 7], 6: [5], 7: [4, 8], 8: [5, 9], 9: [8]}
```

Airline routes



Comparing representations

- Adjacency list typically requires less space

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

0	[1,4]
1	[2]
2	[0]
3	[4,6]
4	[0,3,7]

5	[3,7]
6	[5]
7	[4,8]
8	[5,9]
9	[8]

Comparing representations

- Adjacency list typically requires less space
- Is j a neighbour of i ?
 - Check if $A[i, j] = 1$ in adjacency matrix
 - Scan all neighbours of i in adjacency list

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

0	[1,4]
1	[2]
2	[0]
3	[4,6]
4	[0,3,7]

5	[3,7]
6	[5]
7	[4,8]
8	[5,9]
9	[8]

Comparing representations

- Adjacency list typically requires less space
- Is j a neighbour of i ?
 - Check if $A[i, j] = 1$ in adjacency matrix
 - Scan all neighbours of i in adjacency list
- Which are the neighbours of i ?
 - Scan all n entries in row i in adjacency matrix
 - Takes time proportional to (out)degree of i in adjacency list

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

0	[1,4]
1	[2]
2	[0]
3	[4,6]
4	[0,3,7]

5	[3,7]
6	[5]
7	[4,8]
8	[5,9]
9	[8]

Comparing representations

- Adjacency list typically requires less space
- Is j a neighbour of i ?
 - Check if $A[i, j] = 1$ in adjacency matrix
 - Scan all neighbours of i in adjacency list
- Which are the neighbours of i ?
 - Scan all n entries in row i in adjacency matrix
 - Takes time proportional to (out)degree of i in adjacency list
- Choose representation depending on requirement

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

0	[1,4]
1	[2]
2	[0]
3	[4,6]
4	[0,3,7]

5	[3,7]
6	[5]
7	[4,8]
8	[5,9]
9	[8]

Summary

- To operate on graphs, we need to represent them
- Adjacency matrix
 - $n \times n$ matrix, $A\text{Mat}[i,j] = 1$ iff $(i,j) \in E$
- Adjacency list
 - Dictionary of lists
 - For each vertex i , $A\text{List}[i]$ is the list of neighbours of i
- Can systematically explore a graph using these representations
 - For reachability, propagate marking to all reachable vertices

Breadth First Search

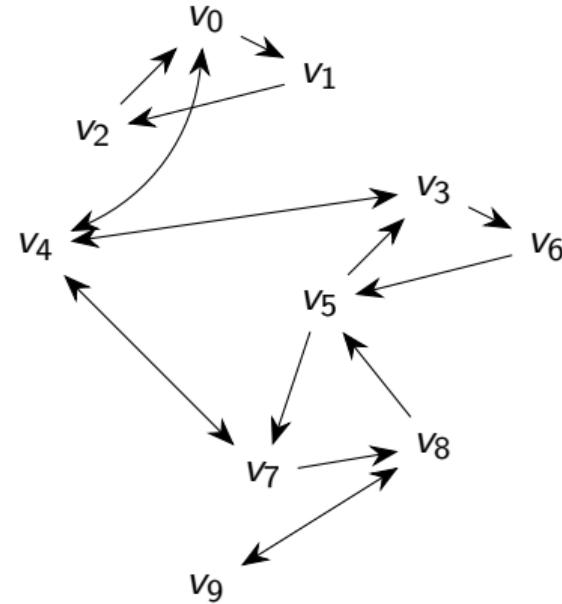
Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 4

Reachability in a graph

- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked



Reachability in a graph

- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked
- Choose an appropriate representation
 - Adjacency matrix
 - Adjacency list

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

0	{1,4}
1	{2}
2	{0}
3	{4,6}
4	{0,3,7}

5	{3,7}
6	{5}
7	{4,8}
8	{5,9}
9	{8}

Reachability in a graph

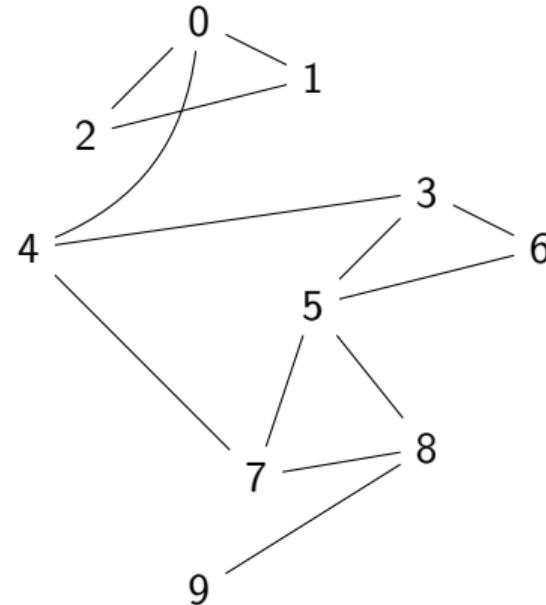
- Mark source vertex as reachable
- Systematically mark neighbours of marked vertices
- Stop when target becomes marked
- Choose an appropriate representation
 - Adjacency matrix
 - Adjacency list
- Strategies for systematic exploration
 - Breadth first — propagate marks in “layers”
 - Depth first — explore a path till it dies out, then backtrack

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	0	0	0
4	1	0	0	1	0	0	0	1	0	0
5	0	0	0	1	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	1	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	0	0	0	1	0

0	{1,4}
1	{2}
2	{0}
3	{4,6}
4	{0,3,7}
5	{3,7}
6	{5}
7	{4,8}
8	{5,9}
9	{8}

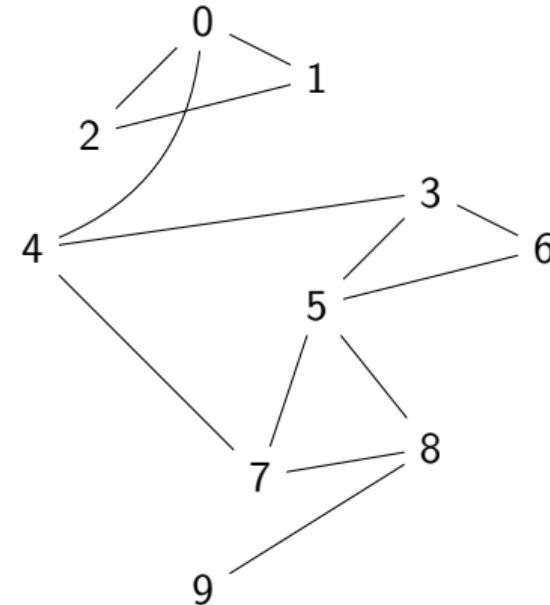
Breadth first search (BFS)

- Explore the graph level by level
 - First visit vertices one step away
 - Then two steps away
 - ...



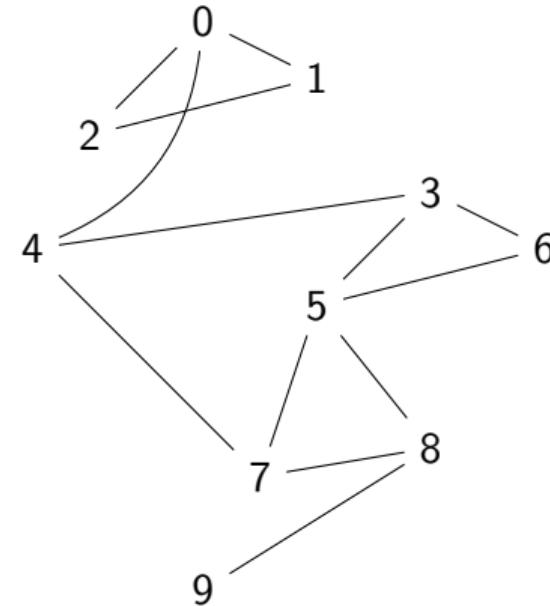
Breadth first search (BFS)

- Explore the graph level by level
 - First visit vertices one step away
 - Then two steps away
 - ...
- Each **visited** vertex has to be **explored**
 - Extend the search to its neighbours
 - Do this only once for each vertex!



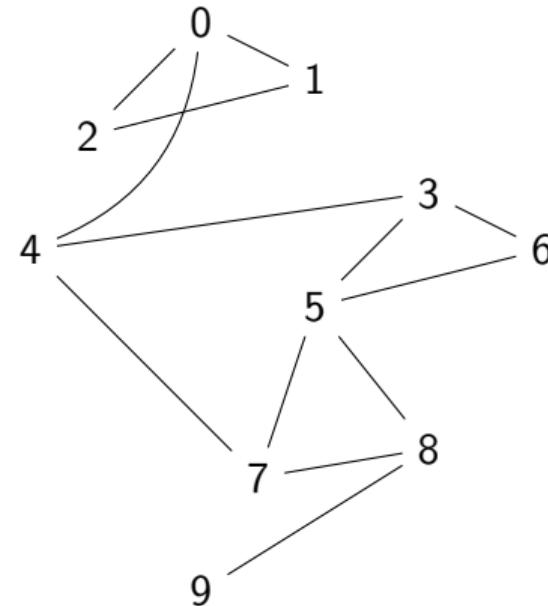
Breadth first search (BFS)

- Explore the graph level by level
 - First visit vertices one step away
 - Then two steps away
 - ...
- Each **visited** vertex has to be **explored**
 - Extend the search to its neighbours
 - Do this only once for each vertex!
- Maintain information about vertices
 - Which vertices have been visited already
 - Among these, which are yet to be explored



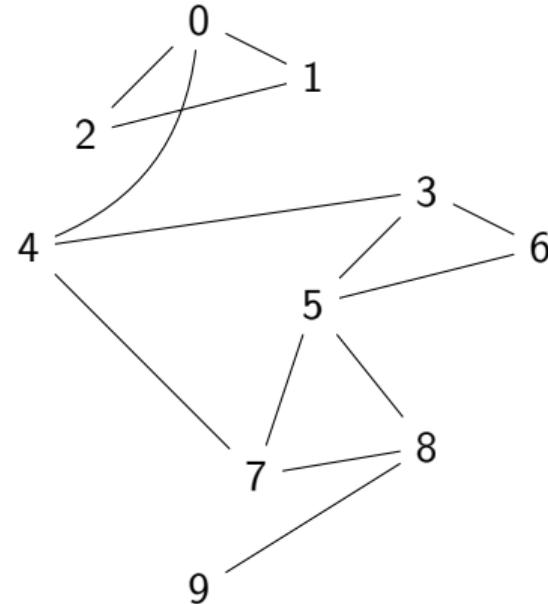
Breadth first search (BFS) ...

- Assume $V = \{0, 1, \dots, n - 1\}$



Breadth first search (BFS) ...

- Assume $V = \{0, 1, \dots, n - 1\}$
- `visited` : $V \rightarrow \{\text{True}, \text{False}\}$ tells us whether $v \in V$ has been visited
 - Initially, `visited(v) = False` for all $v \in V$



Breadth first search (BFS) ...

- Assume $V = \{0, 1, \dots, n - 1\}$
- $\text{visited} : V \rightarrow \{\text{True}, \text{False}\}$ tells us whether $v \in V$ has been visited
 - Initially, $\text{visited}(v) = \text{False}$ for all $v \in V$
- Maintain a sequence of visited vertices yet be explored
 - A **queue** — first in, first out
 - Initially empty

```
class Queue:  
    def __init__(self):  
        self.queue = []  
  
    def addq(self,v):  
        self.queue.append(v)  
  
    def delq(self):  
        v = None  
        if not self.isEmpty():  
            v = self.queue[0]  
            self.queue = self.queue[1:]  
        return(v)  
  
    def isEmpty(self):  
        return(self.queue == [])  
  
    def __str__(self):  
        return(str(self.queue))
```

Breadth first search (BFS) ...

- Assume $V = \{0, 1, \dots, n - 1\}$
- $\text{visited} : V \rightarrow \{\text{True}, \text{False}\}$ tells us whether $v \in V$ has been visited
 - Initially, $\text{visited}(v) = \text{False}$ for all $v \in V$
- Maintain a sequence of visited vertices yet be explored
 - A **queue** — first in, first out
 - Initially empty

```
q = Queue()

for i in range(3):
    q.addq(i)
    print(q)
print(q.isempty())

for j in range(3):
    print(q.delq(), q)
print(q.isempty())
```

```
[0]
[0, 1]
[0, 1, 2]
False
0 [1, 2]
1 [2]
2 []
True
```

Breadth first search (BFS) ...

- Assume $V = \{0, 1, \dots, n - 1\}$
- $\text{visited} : V \rightarrow \{\text{True}, \text{False}\}$ tells us whether $v \in V$ has been visited
 - Initially, $\text{visited}(v) = \text{False}$ for all $v \in V$
- Maintain a sequence of visited vertices yet be explored
 - A **queue** — first in, first out
 - Initially empty
- Exploring a vertex i
 - For each edge (i, j) , if $\text{visited}(j)$ is **False**,
 - Set $\text{visited}(j)$ to **True**
 - Append j to the queue

```
q = Queue()

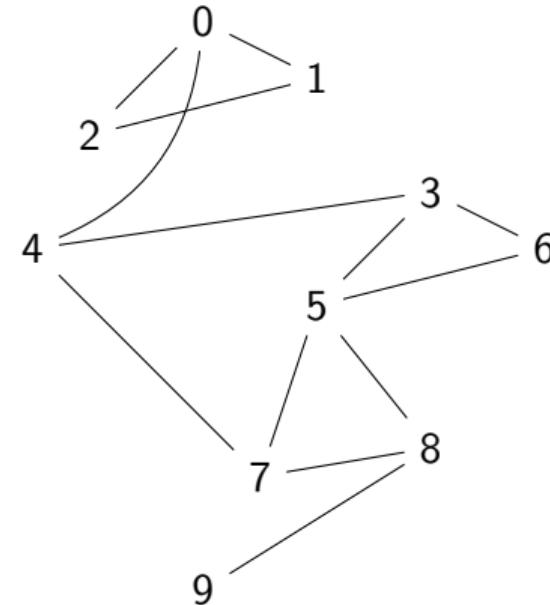
for i in range(3):
    q.addq(i)
    print(q)
print(q.isempty())

for j in range(3):
    print(q.delq(), q)
print(q.isempty())

[0]
[0, 1]
[0, 1, 2]
False
0 [1, 2]
1 [2]
2 []
True
```

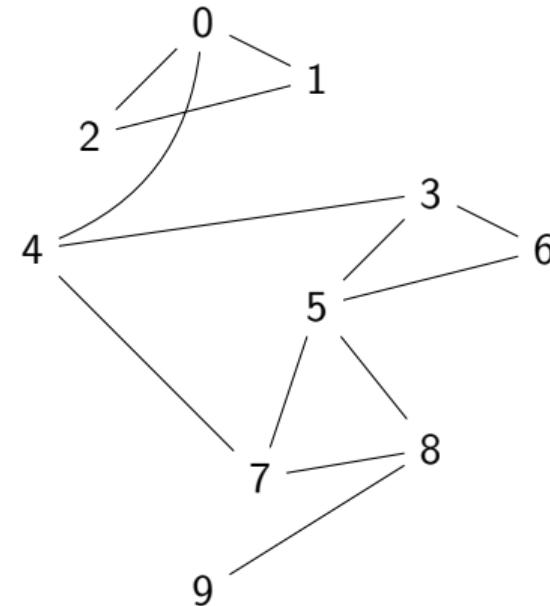
Breadth first search (BFS) ...

- Initially
 - $\text{visited}(v) = \text{False}$ for all $v \in V$
 - Queue of vertices to be explored is empty



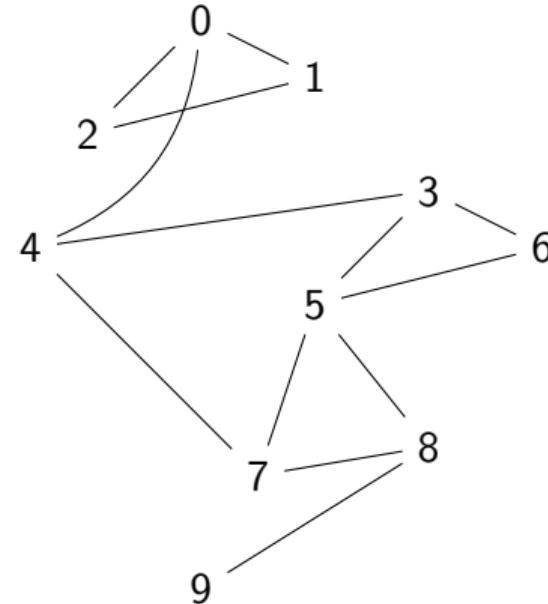
Breadth first search (BFS) ...

- Initially
 - $\text{visited}(v) = \text{False}$ for all $v \in V$
 - Queue of vertices to be explored is empty
- Start BFS from vertex j
 - Set $\text{visited}(j) = \text{True}$
 - Add j to the queue



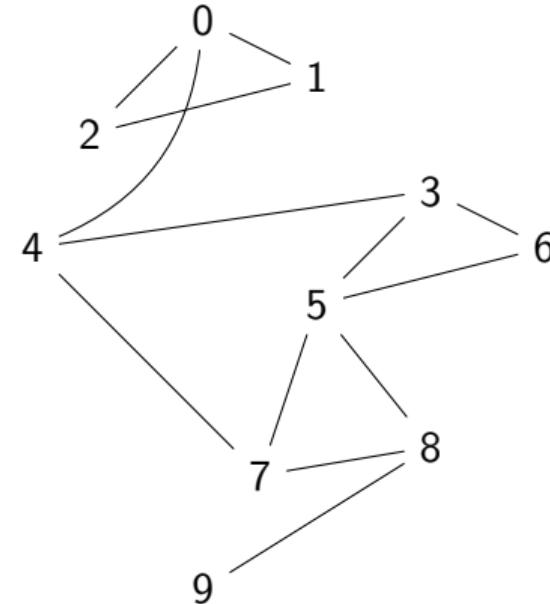
Breadth first search (BFS) ...

- Initially
 - $\text{visited}(v) = \text{False}$ for all $v \in V$
 - Queue of vertices to be explored is empty
- Start BFS from vertex j
 - Set $\text{visited}(j) = \text{True}$
 - Add j to the queue
- Remove and explore vertex i at head of queue
 - For each edge (i,j) , if $\text{visited}(j)$ is False,
 - Set $\text{visited}(j)$ to True
 - Append j to the queue



Breadth first search (BFS) ...

- Initially
 - $\text{visited}(v) = \text{False}$ for all $v \in V$
 - Queue of vertices to be explored is empty
- Start BFS from vertex j
 - Set $\text{visited}(j) = \text{True}$
 - Add j to the queue
- Remove and explore vertex i at head of queue
 - For each edge (i,j) , if $\text{visited}(j)$ is False,
 - Set $\text{visited}(j)$ to True
 - Append j to the queue
- Stop when queue is empty



Breadth first search (BFS) ...

- Initially
 - $\text{visited}(v) = \text{False}$ for all $v \in V$
 - Queue of vertices to be explored is empty
- Start BFS from vertex j
 - Set $\text{visited}(j) = \text{True}$
 - Add j to the queue
- Remove and explore vertex i at head of queue
 - For each edge (i,j) , if $\text{visited}(j)$ is False,
 - Set $\text{visited}(j)$ to True
 - Append j to the queue
- Stop when queue is empty

```
def BFS(AMat,v):  
    (rows,cols) = AMat.shape  
    visited = {}  
    for i in range(rows):  
        visited[i] = False  
    q = Queue()  
  
    visited[v] = True  
    q.addq(v)  
  
    while(not q.isempty()):  
        j = q.delq()  
        for k in neighbours(AMat,j):  
            if (not visited[k]):  
                visited[k] = True  
                q.addq(k)  
  
    return(visited)
```

Breadth first search (BFS) ...

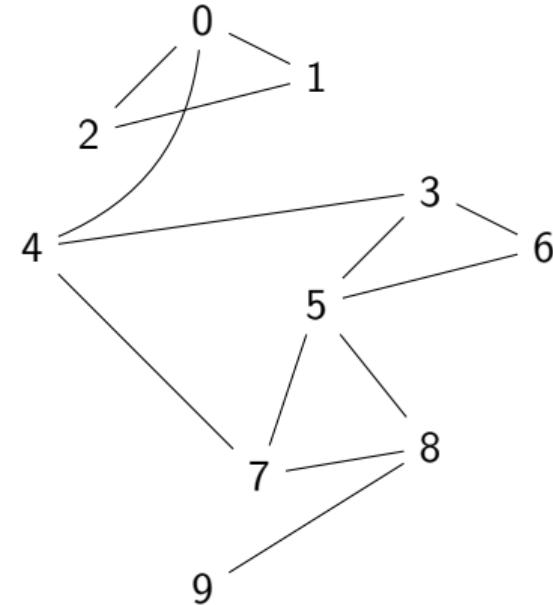
- Initially
 - $\text{visited}(v) = \text{False}$ for all $v \in V$
 - Queue of vertices to be explored is empty
- Start BFS from vertex j
 - Set $\text{visited}(j) = \text{True}$
 - Add j to the queue
- Remove and explore vertex i at head of queue
 - For each edge (i,j) , if $\text{visited}(j)$ is False,
 - Set $\text{visited}(j)$ to True
 - Append j to the queue
- Stop when queue is empty

```
def BFSLList(AList,v):  
    visited = {}  
    for i in AList.keys():  
        visited[i] = False  
    q = Queue()  
  
    visited[v] = True  
    q.addq(v)  
  
    while(not q.isempty()):  
        j = q.delq()  
        for k in AList[j]:  
            if (not visited[k]):  
                visited[k] = True  
                q.addq(k)  
  
    return(visited)
```

BFS from vertex 7

Visited	
0	False
1	False
2	False
3	False
4	False
5	False
6	False
7	False
8	False
9	False

To explore queue								

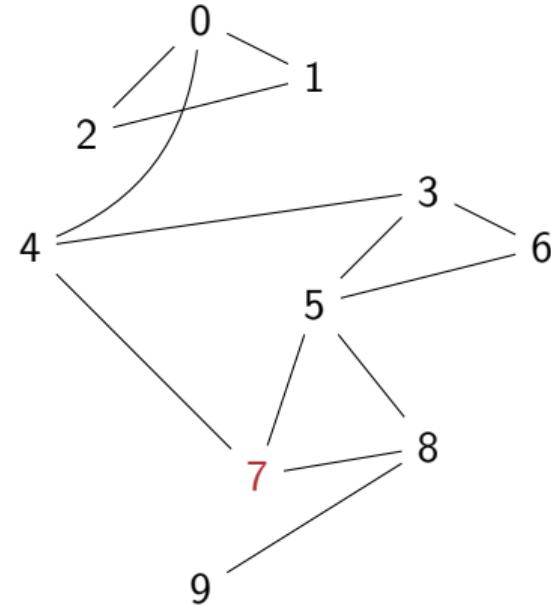


BFS from vertex 7

Visited	
0	False
1	False
2	False
3	False
4	False
5	False
6	False
7	True
8	False
9	False

To explore queue								
7								

- Mark 7 and add to queue

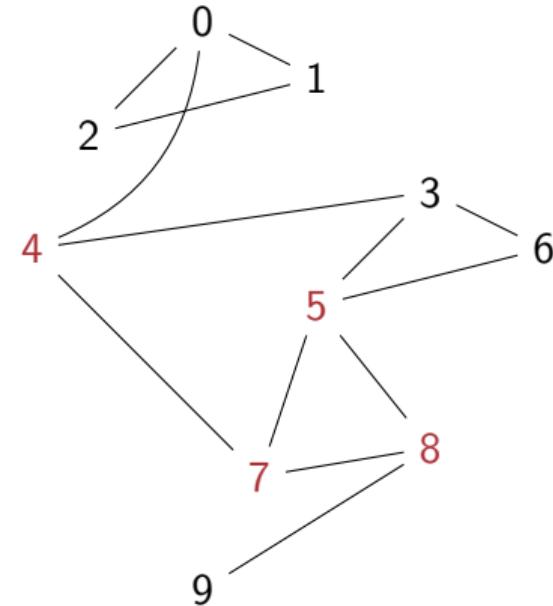


BFS from vertex 7

Visited	
0	False
1	False
2	False
3	False
4	True
5	True
6	False
7	True
8	True
9	False

To explore queue								
4	5	8						

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}

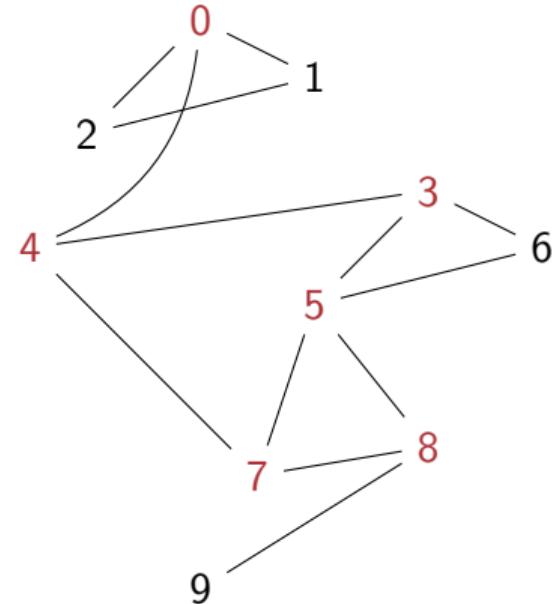


BFS from vertex 7

Visited	
0	True
1	False
2	False
3	True
4	True
5	True
6	False
7	True
8	True
9	False

To explore queue								
5	8	0	3					

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}

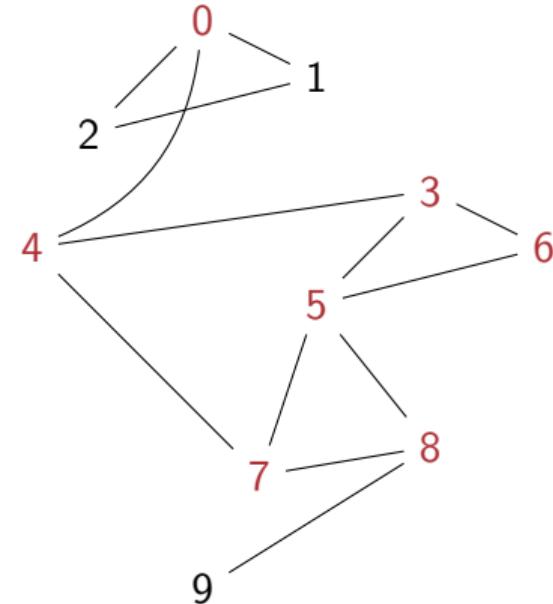


BFS from vertex 7

Visited	
0	True
1	False
2	False
3	True
4	True
5	True
6	True
7	True
8	True
9	False

To explore queue								
8	0	3	6					

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}

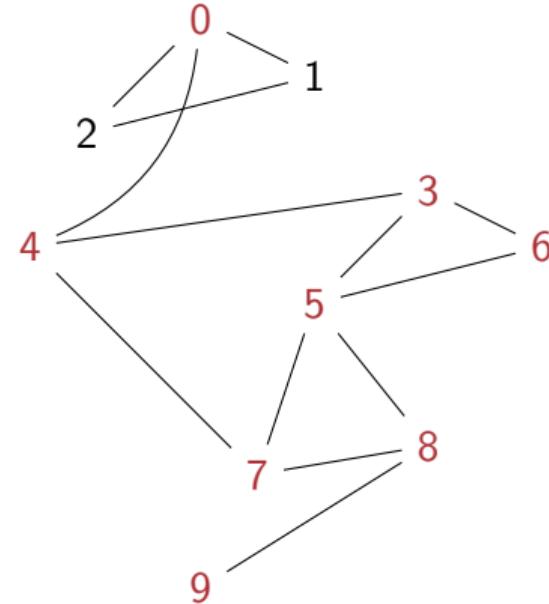


BFS from vertex 7

Visited	
0	True
1	False
2	False
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue								
0	3	6	9					

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}

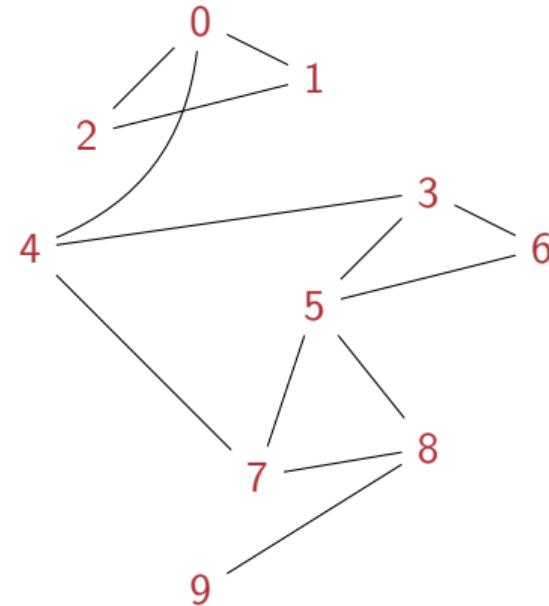


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue									
3	6	9	1	2					

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}

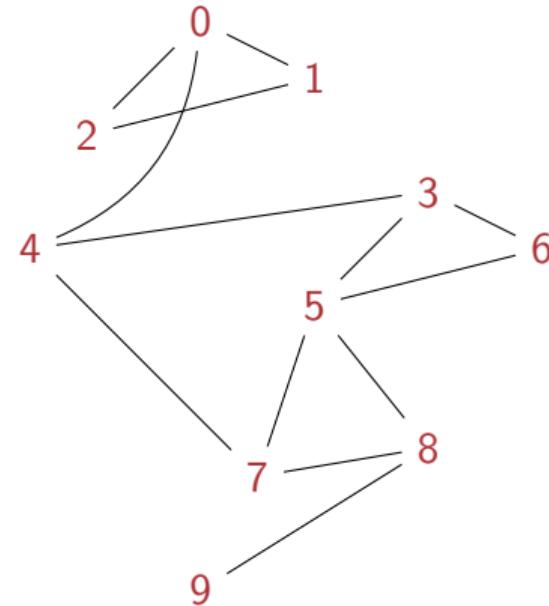


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue								
6	9	1	2					

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3

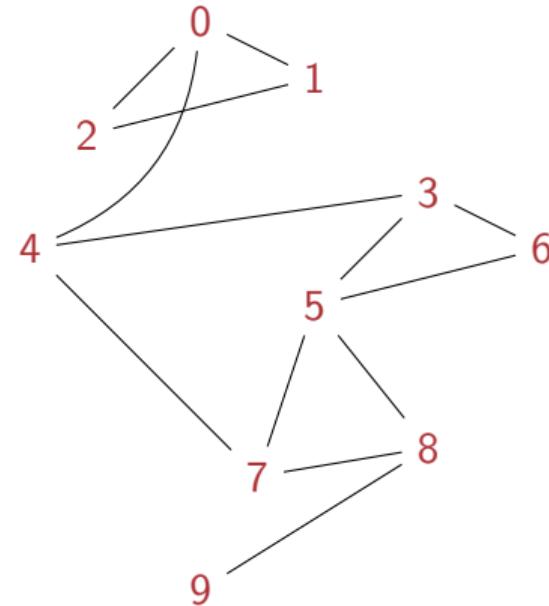


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue								
9	1	2						

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6

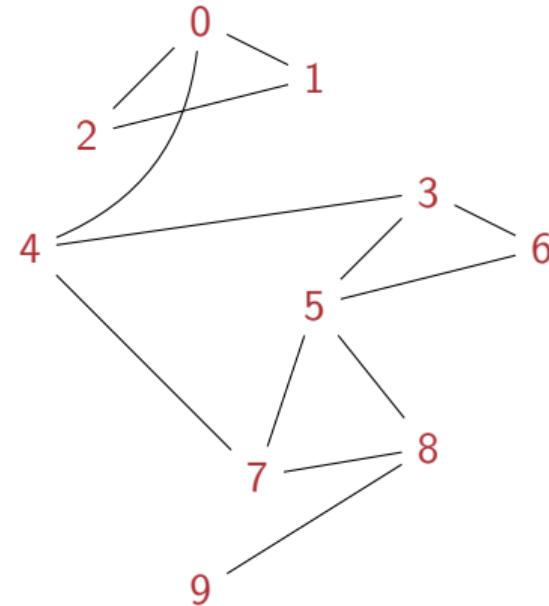


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue								
1	2							

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9

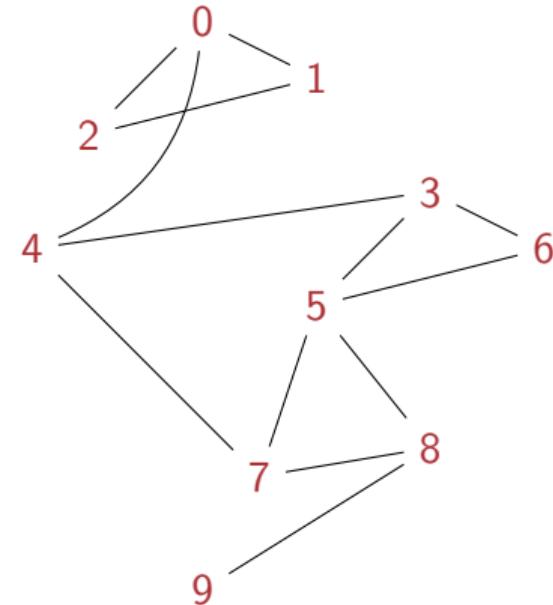


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue							
2							

- Mark 7 and add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1

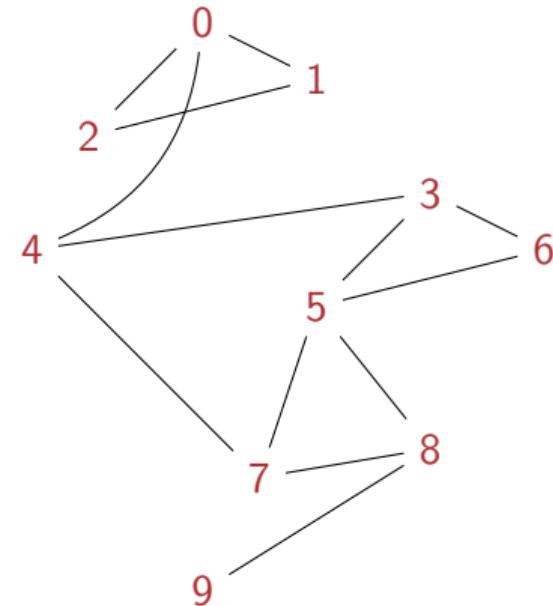


BFS from vertex 7

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

To explore queue

- Mark 7 and add to queue
 - Explore 7, visit $\{4,5,8\}$
 - Explore 4, visit $\{0,3\}$
 - Explore 5, visit $\{6\}$
 - Explore 8, visit $\{9\}$
 - Explore 0, visit $\{1,2\}$
 - Explore 3
 - Explore 6
 - Explore 9
 - Explore 1
 - Explore 2



Complexity of BFS

- $G = (V, E)$
 - $|V| = n$
 - $|E| = m$
 - If G is connected, m can vary from $n - 1$ to $n(n - 1)/2$

Complexity of BFS

- $G = (V, E)$
 - $|V| = n$
 - $|E| = m$
 - If G is connected, m can vary from $n - 1$ to $n(n - 1)/2$
- In BFS, each reachable vertex is processed exactly once
 - Visit the vertex: add to queue
 - Explore the vertex: remove from queue
 - Visit and explore at most n vertices

Complexity of BFS

- $G = (V, E)$
 - $|V| = n$
 - $|E| = m$
 - If G is connected, m can vary from $n - 1$ to $n(n - 1)/2$
- In BFS, each reachable vertex is processed exactly once
 - Visit the vertex: add to queue
 - Explore the vertex: remove from queue
 - Visit and explore at most n vertices
- Exploring a vertex
 - Check all outgoing edges
 - How long does this take?

Complexity of BFS

- $G = (V, E)$
 - $|V| = n$
 - $|E| = m$
 - If G is connected, m can vary from $n - 1$ to $n(n - 1)/2$
- In BFS, each reachable vertex is processed exactly once
 - Visit the vertex: add to queue
 - Explore the vertex: remove from queue
 - Visit and explore at most n vertices
- Exploring a vertex
 - Check all outgoing edges
 - How long does this take?

Adjacency matrix

- To explore i , scan $\text{neighbours}(i)$
- Look up n entries in row i , regardless of $\text{degree}(i)$

Complexity of BFS

- $G = (V, E)$
 - $|V| = n$
 - $|E| = m$
 - If G is connected, m can vary from $n - 1$ to $n(n - 1)/2$
- In BFS, each reachable vertex is processed exactly once
 - Visit the vertex: add to queue
 - Explore the vertex: remove from queue
 - Visit and explore at most n vertices
- Exploring a vertex
 - Check all outgoing edges
 - How long does this take?

Adjacency matrix

- To explore i , scan $\text{neighbours}(i)$
- Look up n entries in row i , regardless of $\text{degree}(i)$

Adjacency list

- List $\text{neighbours}(i)$ is directly available
- Time to explore i is $\text{degree}(i)$
- Degree varies across vertices

Complexity of BFS

- $G = (V, E)$
 - $|V| = n$
 - $|E| = m$
 - If G is connected, m can vary from $n - 1$ to $n(n - 1)/2$
- In BFS, each reachable vertex is processed exactly once
 - Visit the vertex: add to queue
 - Explore the vertex: remove from queue
 - Visit and explore at most n vertices
- Exploring a vertex
 - Check all outgoing edges
 - How long does this take?

Adjacency matrix

- To explore i , scan $\text{neighbours}(i)$
- Look up n entries in row i , regardless of $\text{degree}(i)$

Adjacency list

- List $\text{neighbours}(i)$ is directly available
- Time to explore i is $\text{degree}(i)$
- Degree varies across vertices

Sum of degrees

- Sum of degrees is $2m$
- Each edge (i, j) contributes to $\text{degree}(i)$ and $\text{degree}(j)$

Complexity of BFS

BFS with adjacency matrix

- n steps to initialize each vertex
- n steps to explore each vertex
- Overall time is $O(n^2)$

Complexity of BFS

BFS with adjacency matrix

- n steps to initialize each vertex
- n steps to explore each vertex
- Overall time is $O(n^2)$

BFS with adjacency list

- n steps to initialize each vertex
- $2m$ steps (sum of degrees) to explore all vertices
 - An example of **amortized** analysis
- Overall time is $O(n + m)$

Complexity of BFS

BFS with adjacency matrix

- n steps to initialize each vertex
- n steps to explore each vertex
- Overall time is $O(n^2)$

- If $m \ll n^2$, working with adjacency lists is much more efficient
 - This is why we treat m and n as separate parameters

BFS with adjacency list

- n steps to initialize each vertex
- $2m$ steps (sum of degrees) to explore all vertices
 - An example of **amortized** analysis
- Overall time is $O(n + m)$

Complexity of BFS

BFS with adjacency matrix

- n steps to initialize each vertex
- n steps to explore each vertex
- Overall time is $O(n^2)$

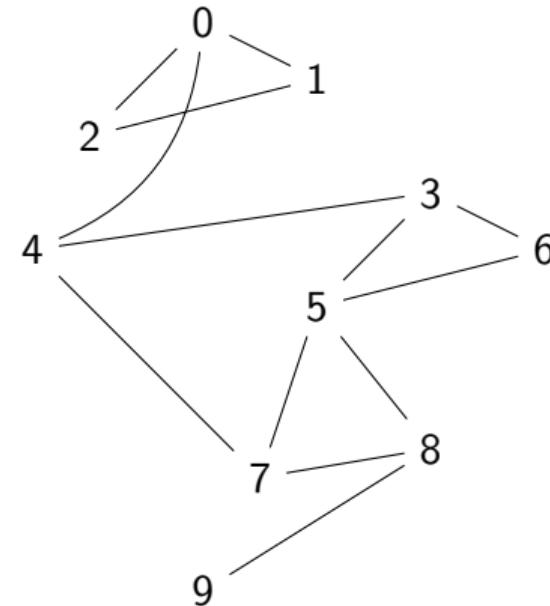
BFS with adjacency list

- n steps to initialize each vertex
- $2m$ steps (sum of degrees) to explore all vertices
 - An example of **amortized** analysis
- Overall time is $O(n + m)$

- If $m \ll n^2$, working with adjacency lists is much more efficient
 - This is why we treat m and n as separate parameters
- For graphs, $O(m + n)$ is typically the best possible complexity
 - Need to see each each vertex and edge at least once
 - Linear time

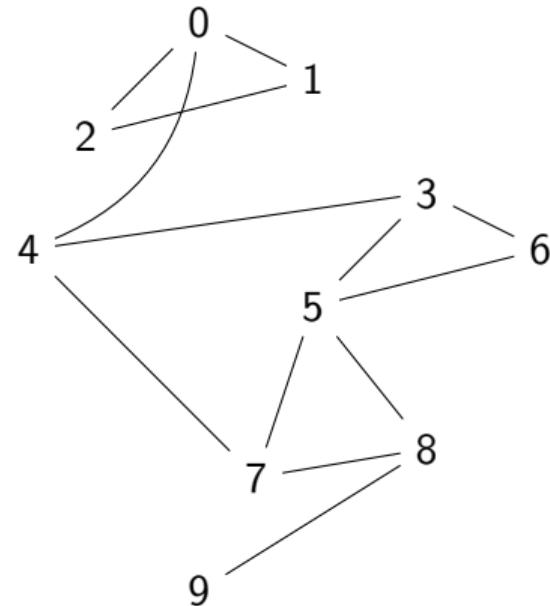
Enhancing BFS to record paths

- If BFS from i sets `visited(k) = True`, we know that k is reachable from i
- How do we recover a path from i to k ?



Enhancing BFS to record paths

- If BFS from i sets $\text{visited}(k) = \text{True}$, we know that k is reachable from i
- How do we recover a path from i to k ?
- $\text{visited}(k)$ was set to True when exploring some vertex j



Enhancing BFS to record paths

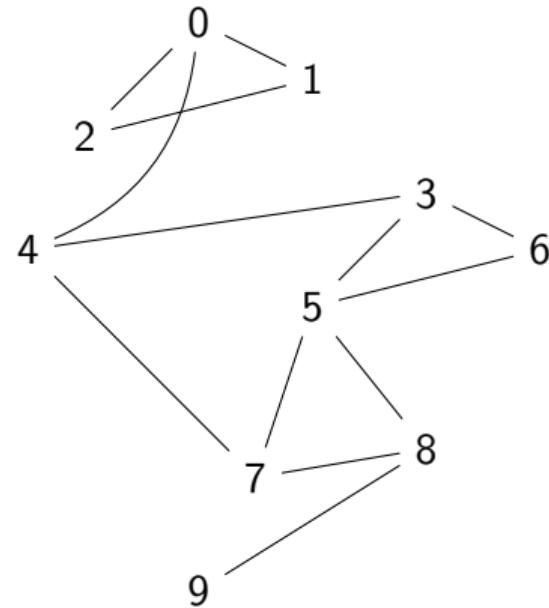
- If BFS from i sets $\text{visited}(k) = \text{True}$, we know that k is reachable from i
- How do we recover a path from i to k ?
- $\text{visited}(k)$ was set to True when exploring some vertex j
- Record $\text{parent}(k) = j$
- From k , follow parent links to trace back a path to i

```
def BFSListPath(AList,v):  
    (visited,parent) = ({},{})  
    for i in AList.keys():  
        visited[i] = False  
        parent[i] = -1  
    q = Queue()  
  
    visited[v] = True  
    q.addq(v)  
  
    while(not q.isempty()):  
        j = q.delq()  
        for k in AList[j]:  
            if (not visited[k]):  
                visited[k] = True  
                parent[k] = j  
                q.addq(k)  
  
    return(visited,parent)
```

BFS from vertex 7 with parent information

	Visited	Parent
0	False	-1
1	False	-1
2	False	-1
3	False	-1
4	False	-1
5	False	-1
6	False	-1
7	False	-1
8	False	-1
9	False	-1

To explore queue							

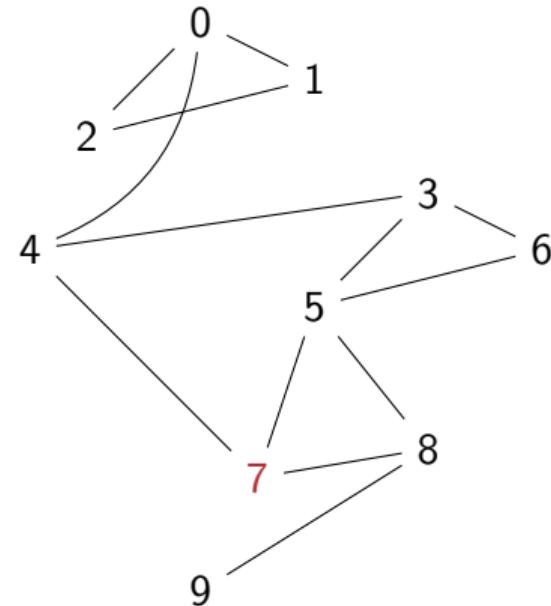


BFS from vertex 7 with parent information

	Visited	Parent
0	False	-1
1	False	-1
2	False	-1
3	False	-1
4	False	-1
5	False	-1
6	False	-1
7	True	-1
8	False	-1
9	False	-1

To explore queue							
7							

- Mark 7, add to queue

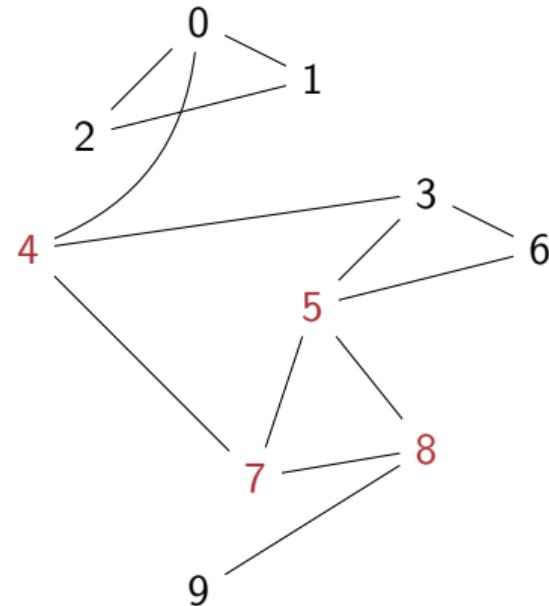


BFS from vertex 7 with parent information

	Visited	Parent
0	False	-1
1	False	-1
2	False	-1
3	False	-1
4	True	7
5	True	7
6	False	-1
7	True	-1
8	True	7
9	False	-1

To explore queue								
4	5	8						

- Mark 7, add to queue
- Explore 7, visit {4,5,8}

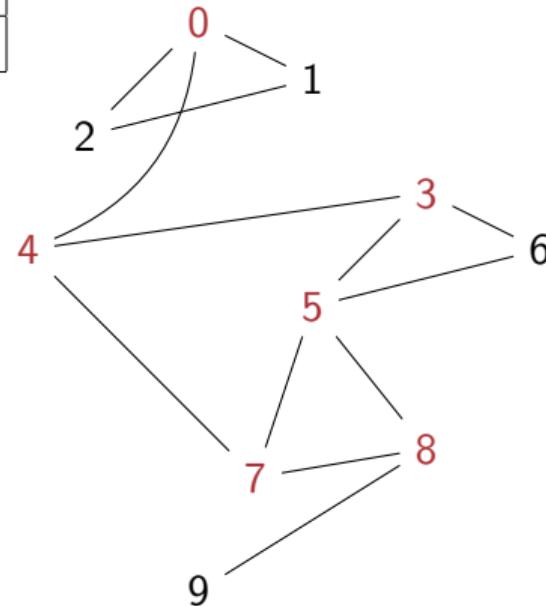


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	False	-1
2	False	-1
3	True	4
4	True	7
5	True	7
6	False	-1
7	True	-1
8	True	7
9	False	-1

To explore queue								
5	8	0	3					

- Mark 7, add to queue
 - Explore 7, visit {4,5,8}
 - Explore 4, visit {0,3}

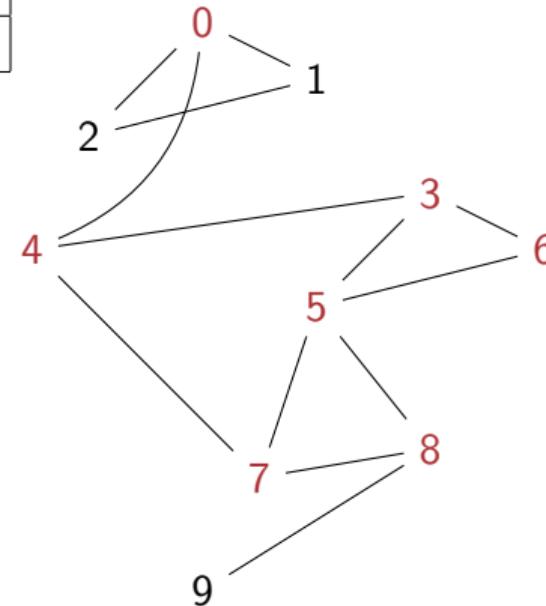


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	False	-1
2	False	-1
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	False	-1

To explore queue							
8	0	3	6				

- Mark 7, add to queue
 - Explore 7, visit {4,5,8}
 - Explore 4, visit {0,3}
 - Explore 5, visit {6}

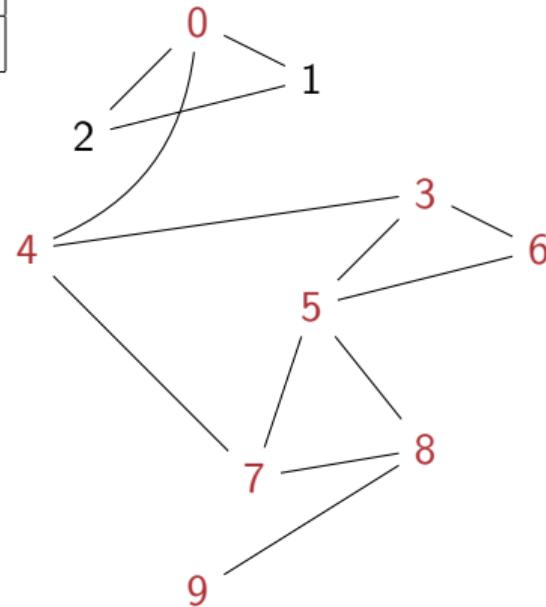


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	False	-1
2	False	-1
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue									
0	3	6	9						

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}

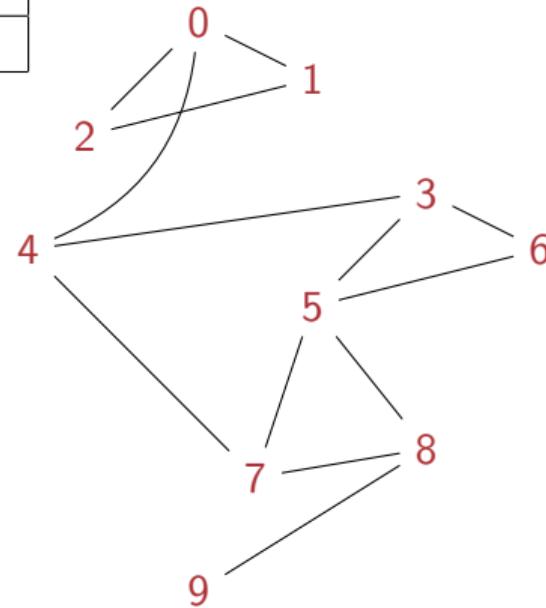


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue									
3	6	9	1	2					

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}

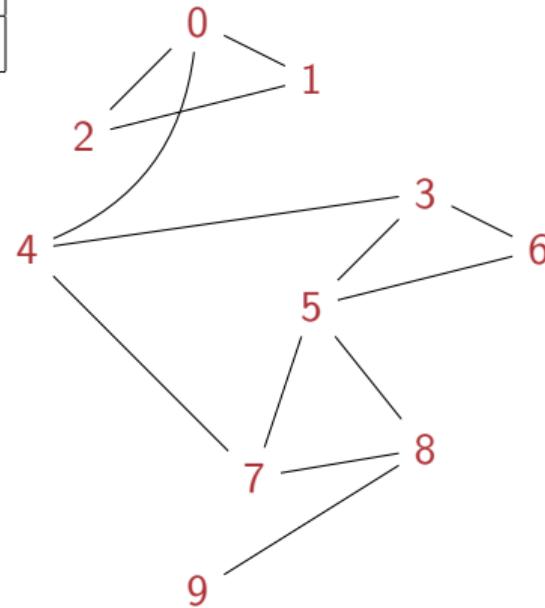


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue									
6	9	1	2						

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3

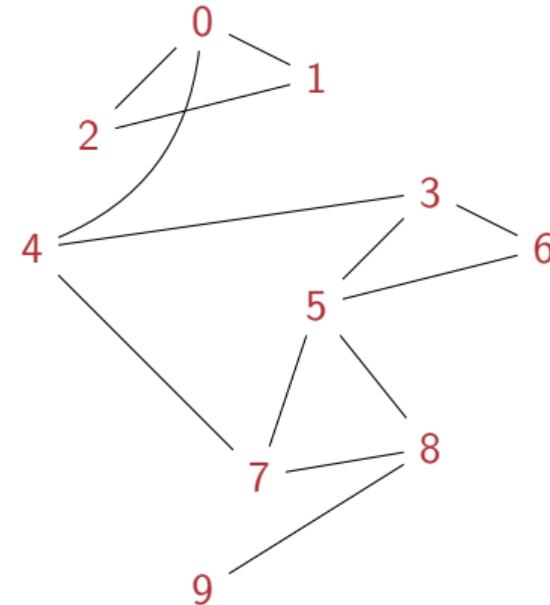


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue								
9	1	2						

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6

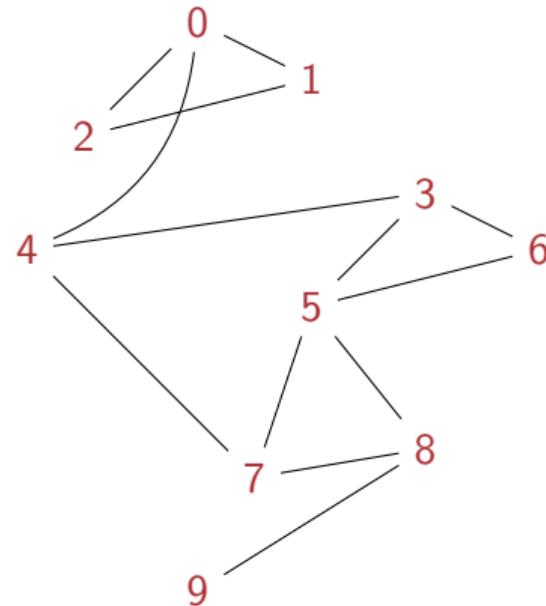


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue								
1	2							

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9

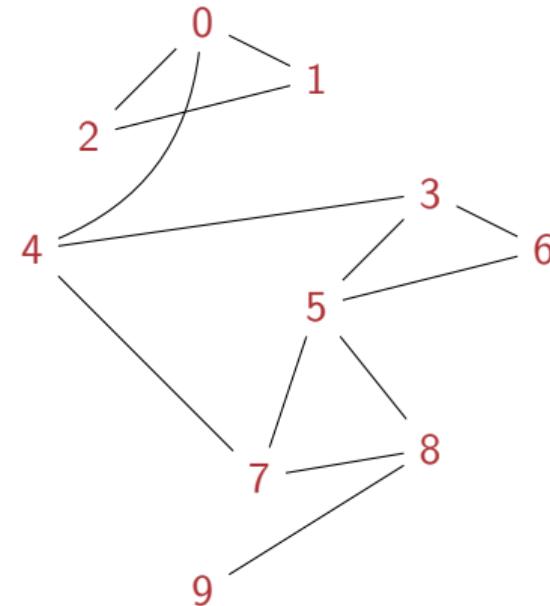


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue								
2								

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1

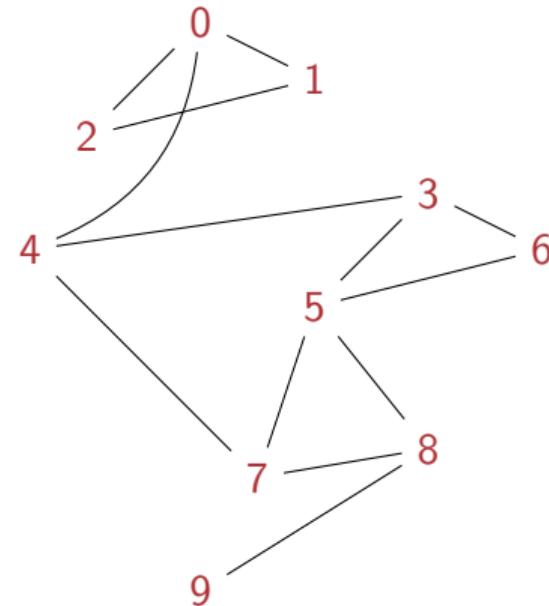


BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

To explore queue

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



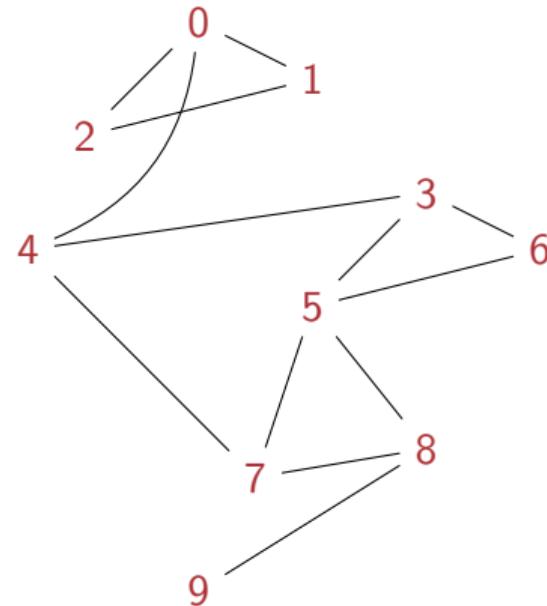
BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

Path from 7 to 6 is
7–5–6

To explore queue							

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



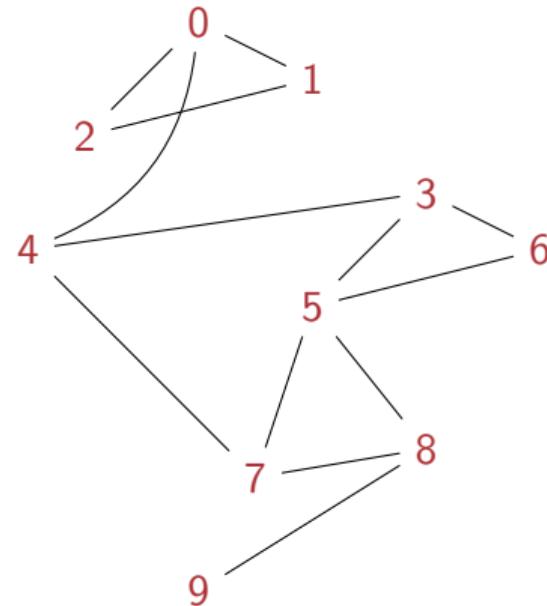
BFS from vertex 7 with parent information

	Visited	Parent
0	True	4
1	True	0
2	True	0
3	True	4
4	True	7
5	True	7
6	True	5
7	True	-1
8	True	7
9	True	8

Path from 7 to 2 is
7–4–0–2

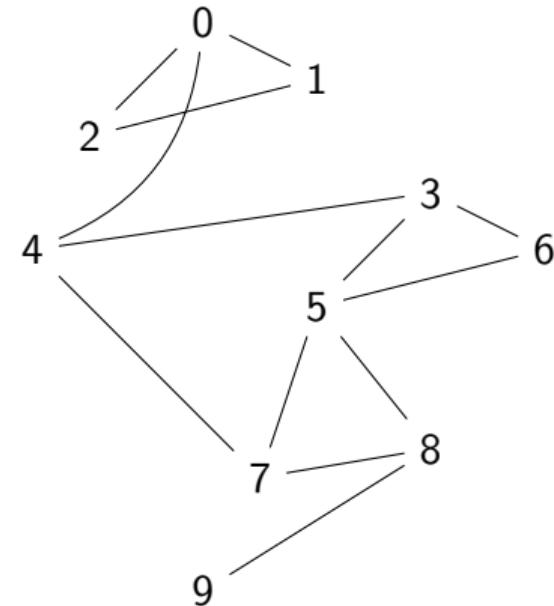
To explore queue							

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



Enhancing BFS to record distance

- BFS explores neighbours level by level
- By recording the level at which a vertex is visited, we get its distance from the source vertex



Enhancing BFS to record distance

- BFS explores neighbours level by level
- By recording the level at which a vertex is visited, we get its distance from the source vertex
- Instead of `visited(j)`, maintain `level(j)`

```
def BFSListPathLevel(AList,v):  
    (level,parent) = ({},{} )  
    for i in AList.keys():  
        level[i] = -1  
        parent[i] = -1  
    q = Queue()  
  
    level[v] = 0  
    q.addq(v)  
  
    while(not q.isempty()):  
        j = q.delq()  
        for k in AList[j]:  
            if (level[k] == -1):  
                level[k] = level[j]+1  
                parent[k] = j  
                q.addq(k)  
  
    return(level,parent)
```

Enhancing BFS to record distance

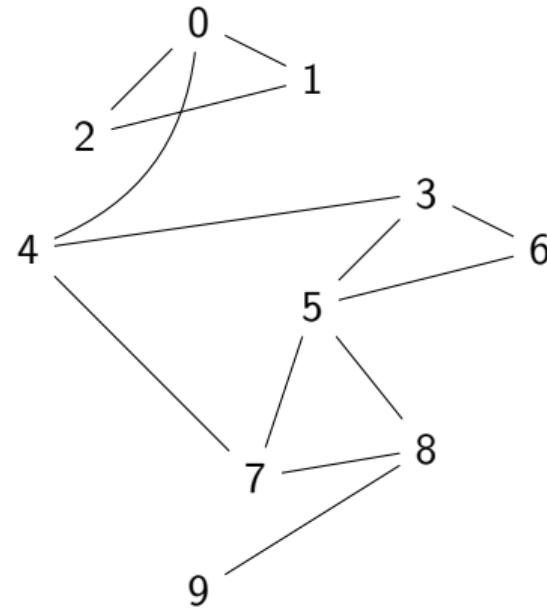
- BFS explores neighbours level by level
- By recording the level at which a vertex is visited, we get its distance from the source vertex
- Instead of `visited(j)`, maintain `level(j)`
- Initialize $\text{level}(j) = -1$ for all j
- Set $\text{level}(i) = 0$ for source vertex
- If we visit k from j , set $\text{level}(k)$ to $\text{level}(j) + 1$
- $\text{level}(j)$ is the length of the shortest path from the source vertex, in number of edges

```
def BFSListPathLevel(AList,v):  
    (level,parent) = ({},{} )  
    for i in AList.keys():  
        level[i] = -1  
        parent[i] = -1  
    q = Queue()  
  
    level[v] = 0  
    q.addq(v)  
  
    while(not q.isempty()):  
        j = q.delq()  
        for k in AList[j]:  
            if (level[k] == -1):  
                level[k] = level[j]+1  
                parent[k] = j  
                q.addq(k)  
  
    return(level,parent)
```

BFS from vertex 7 with parent and distance information

	Level	Parent
0	-1	-1
1	-1	-1
2	-1	-1
3	-1	-1
4	-1	-1
5	-1	-1
6	-1	-1
7	-1	-1
8	-1	-1
9	-1	-1

To explore queue

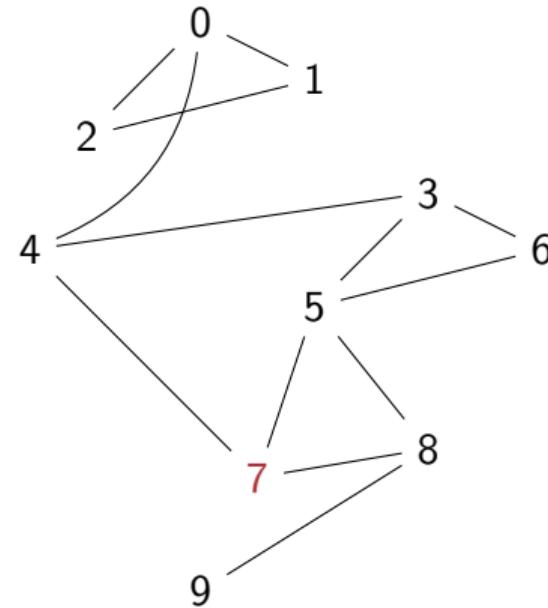


BFS from vertex 7 with parent and distance information

	Level	Parent
0	-1	-1
1	-1	-1
2	-1	-1
3	-1	-1
4	-1	-1
5	-1	-1
6	-1	-1
7	0	-1
8	-1	-1
9	-1	-1

To explore queue

- Mark 7, add to queue

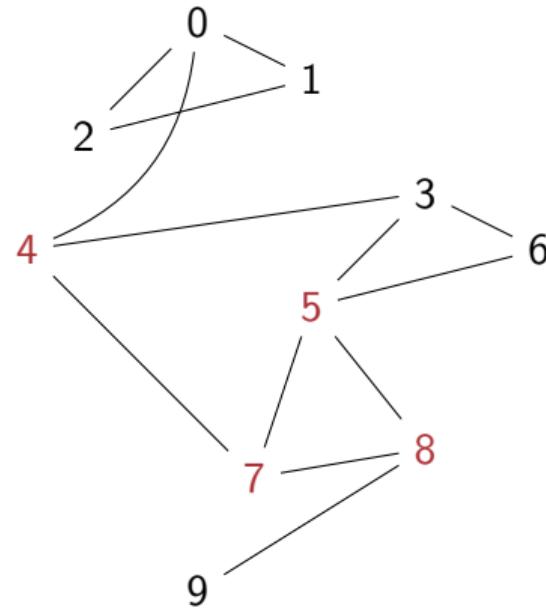


BFS from vertex 7 with parent and distance information

	Level	Parent
0	-1	-1
1	-1	-1
2	-1	-1
3	-1	-1
4	1	7
5	1	7
6	-1	-1
7	0	-1
8	1	7
9	-1	-1

To explore queue								
4	5	8						

- Mark 7, add to queue
- Explore 7, visit {4,5,8}

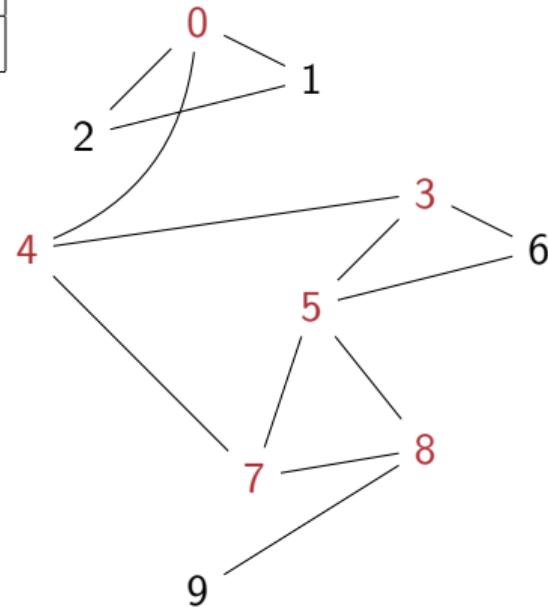


BFS from vertex 7 with parent and distance information

	Level	Parent
0	2	4
1	-1	-1
2	-1	-1
3	2	4
4	1	7
5	1	7
6	-1	-1
7	0	-1
8	1	7
9	-1	-1

To explore queue								
5	8	0	3					

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}

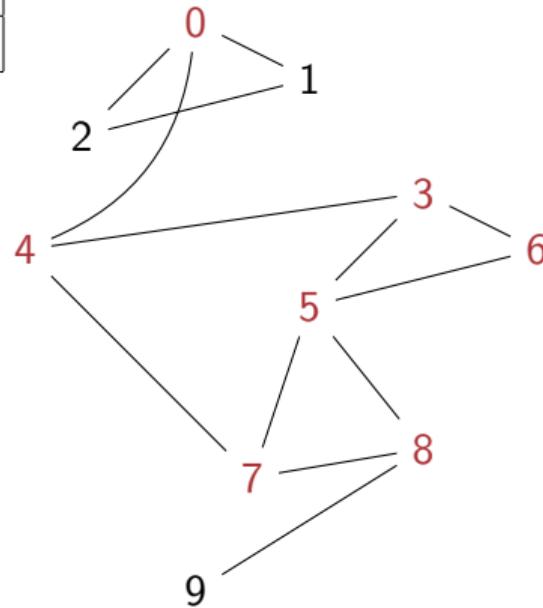


BFS from vertex 7 with parent and distance information

	Level	Parent
0	2	4
1	-1	-1
2	-1	-1
3	2	4
4	1	7
5	1	7
6	2	5
7	0	-1
8	1	7
9	-1	-1

To explore queue								
8	0	3	6					

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}

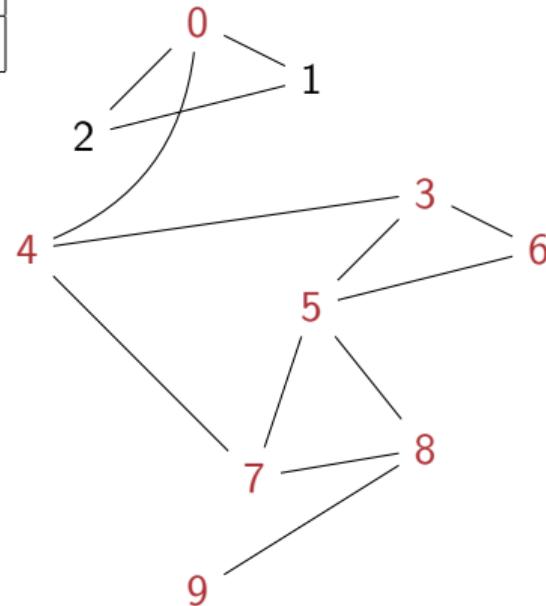


BFS from vertex 7 with parent and distance information

	Level	Parent
0	2	4
1	-1	-1
2	-1	-1
3	2	4
4	1	7
5	1	7
6	2	5
7	0	-1
8	1	7
9	2	8

To explore queue								
0	3	6	9					

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}

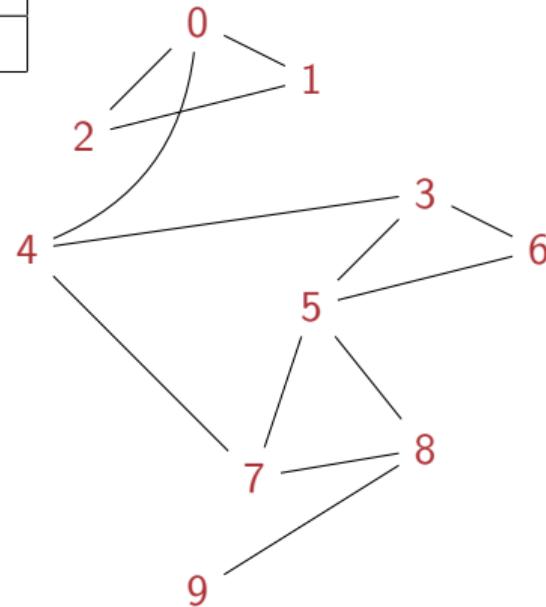


BFS from vertex 7 with parent and distance information

	Level	Parent
0	2	4
1	3	0
2	3	0
3	2	4
4	1	7
5	1	7
6	2	5
7	0	-1
8	1	7
9	2	8

To explore queue									
3	6	9	1	2					

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}

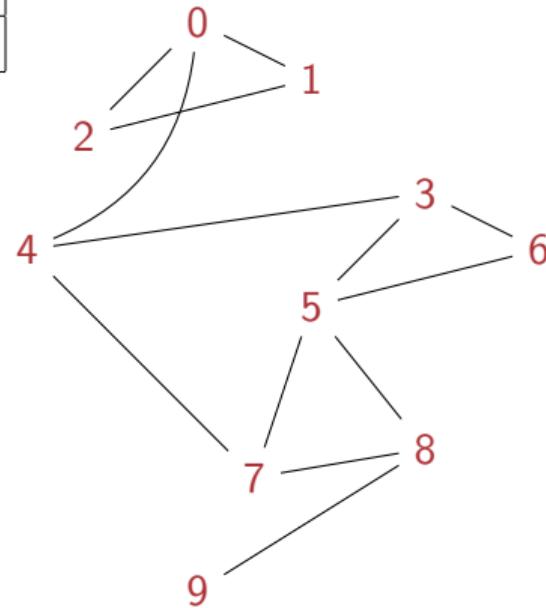


BFS from vertex 7 with parent and distance information

	Level	Parent
0	2	4
1	3	0
2	3	0
3	2	4
4	1	7
5	1	7
6	2	5
7	0	-1
8	1	7
9	2	8

To explore queue									
6	9	1	2						

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3

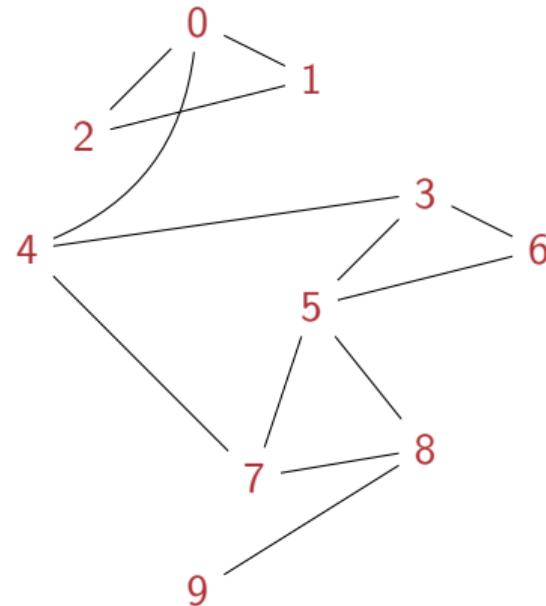


BFS from vertex 7 with parent and distance information

	Level	Parent
0	2	4
1	3	0
2	3	0
3	2	4
4	1	7
5	1	7
6	2	5
7	0	-1
8	1	7
9	2	8

To explore queue								
9	1	2						

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6

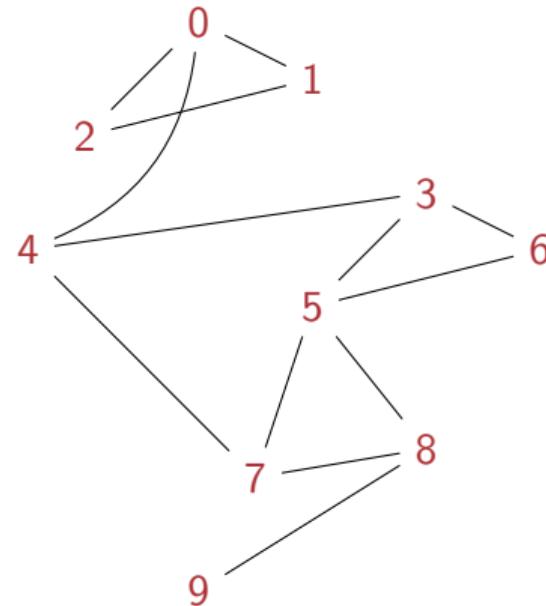


BFS from vertex 7 with parent and distance information

	Level	Parent
0	2	4
1	3	0
2	3	0
3	2	4
4	1	7
5	1	7
6	2	5
7	0	-1
8	1	7
9	2	8

To explore queue							
1	2						

- Mark 7, add to queue
 - Explore 7, visit {4,5,8}
 - Explore 4, visit {0,3}
 - Explore 5, visit {6}
 - Explore 8, visit {9}
 - Explore 0, visit {1,2}
 - Explore 3
 - Explore 6
 - Explore 9

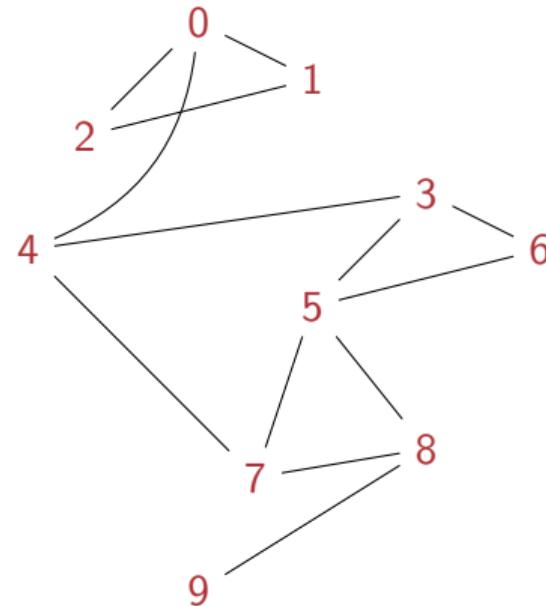


BFS from vertex 7 with parent and distance information

	Level	Parent
0	2	4
1	3	0
2	3	0
3	2	4
4	1	7
5	1	7
6	2	5
7	0	-1
8	1	7
9	2	8

To explore queue							
2							

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1

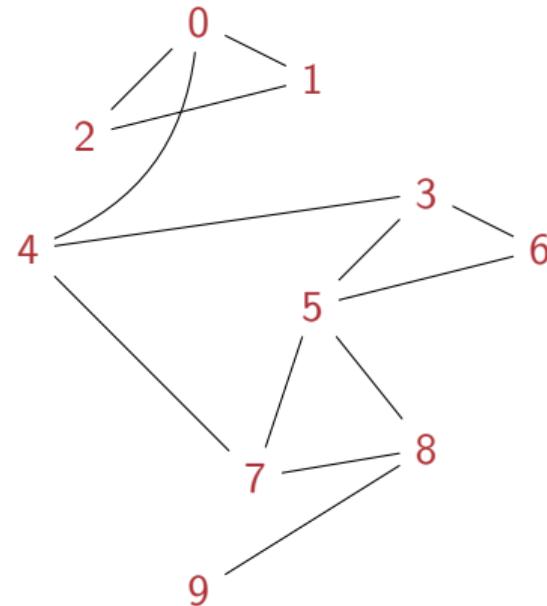


BFS from vertex 7 with parent and distance information

	Level	Parent
0	2	4
1	3	0
2	3	0
3	2	4
4	1	7
5	1	7
6	2	5
7	0	-1
8	1	7
9	2	8

To explore queue

- Mark 7, add to queue
- Explore 7, visit {4,5,8}
- Explore 4, visit {0,3}
- Explore 5, visit {6}
- Explore 8, visit {9}
- Explore 0, visit {1,2}
- Explore 3
- Explore 6
- Explore 9
- Explore 1
- Explore 2



Summary

- Breadth first search is a systematic strategy to explore a graph, level by level

Summary

- Breadth first search is a systematic strategy to explore a graph, level by level
- Record which vertices have been visited

Summary

- Breadth first search is a systematic strategy to explore a graph, level by level
- Record which vertices have been visited
- Maintain visited but unexplored vertices in a queue

Summary

- Breadth first search is a systematic strategy to explore a graph, level by level
- Record which vertices have been visited
- Maintain visited but unexplored vertices in a queue
- Complexity is $O(n^2)$ using adjacency matrix, $O(m + n)$ using adjacency list

Summary

- Breadth first search is a systematic strategy to explore a graph, level by level
- Record which vertices have been visited
- Maintain visited but unexplored vertices in a queue
- Complexity is $O(n^2)$ using adjacency matrix, $O(m + n)$ using adjacency list
- Add parent information to recover the path to each reachable vertex

Summary

- Breadth first search is a systematic strategy to explore a graph, level by level
- Record which vertices have been visited
- Maintain visited but unexplored vertices in a queue
- Complexity is $O(n^2)$ using adjacency matrix, $O(m + n)$ using adjacency list
- Add parent information to recover the path to each reachable vertex
- Maintain level information to record length of the shortest path, in terms of number of edges

Summary

- Breadth first search is a systematic strategy to explore a graph, level by level
- Record which vertices have been visited
- Maintain visited but unexplored vertices in a queue
- Complexity is $O(n^2)$ using adjacency matrix, $O(m + n)$ using adjacency list
- Add parent information to recover the path to each reachable vertex
- Maintain level information to record length of the shortest path, in terms of number of edges
 - In general, edges are labelled with a **cost** (distance, time, ticket price, ...)

Summary

- Breadth first search is a systematic strategy to explore a graph, level by level
- Record which vertices have been visited
- Maintain visited but unexplored vertices in a queue
- Complexity is $O(n^2)$ using adjacency matrix, $O(m + n)$ using adjacency list
- Add parent information to recover the path to each reachable vertex
- Maintain level information to record length of the shortest path, in terms of number of edges
 - In general, edges are labelled with a **cost** (distance, time, ticket price, ...)
 - Will look at **weighted graphs**, where shortest paths are in terms of cost, not number of edges

Depth First Search

Madhavan Mukund

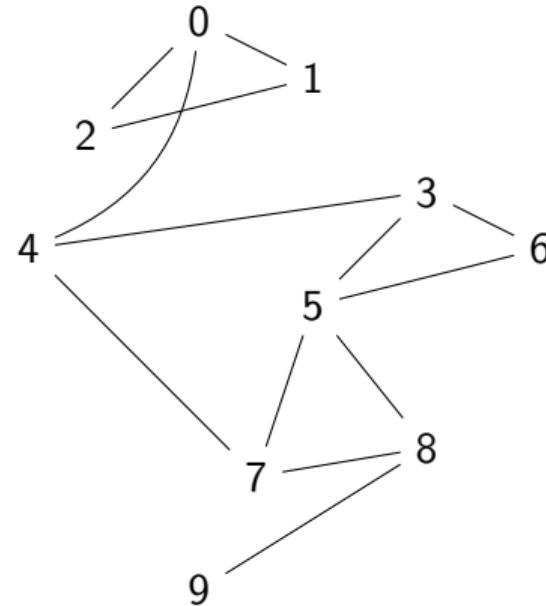
<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 4

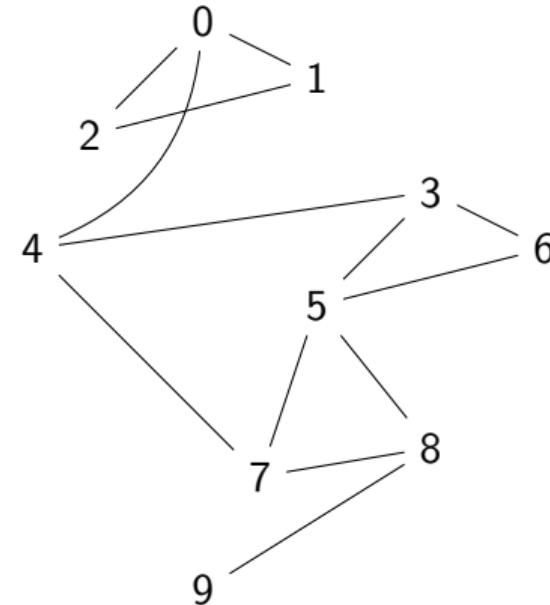
Depth first search (DFS)

- Start from i , visit an unexplored neighbour j



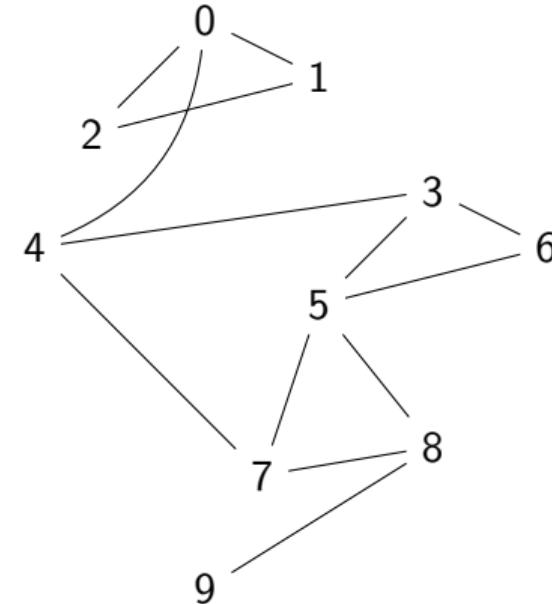
Depth first search (DFS)

- Start from i , visit an unexplored neighbour j
- Suspend the exploration of i and explore j instead



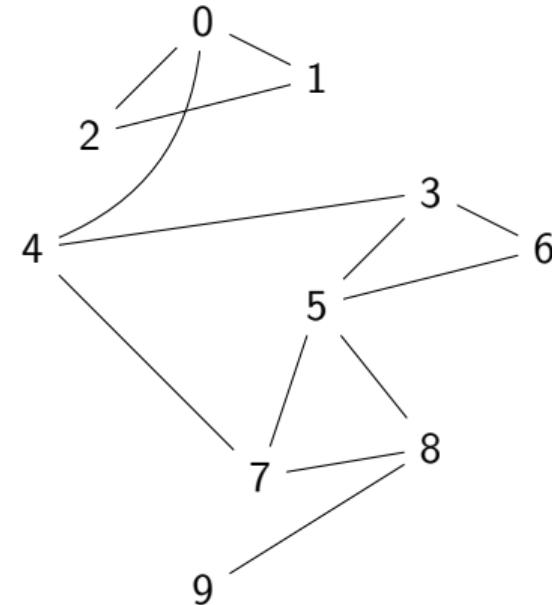
Depth first search (DFS)

- Start from i , visit an unexplored neighbour j
- Suspend the exploration of i and explore j instead
- Continue till you reach a vertex with no unexplored neighbours



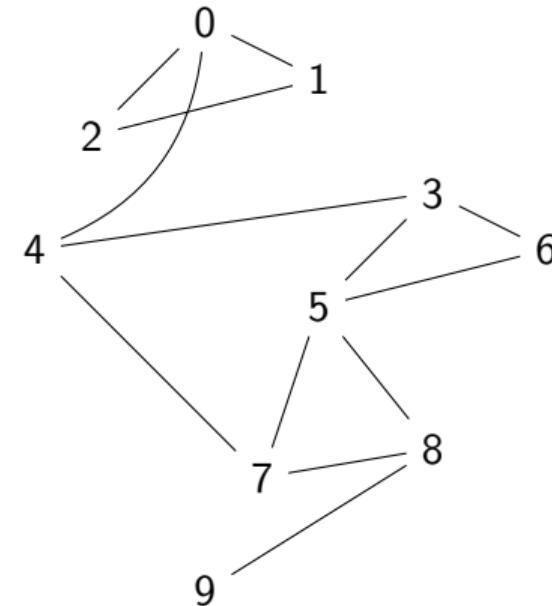
Depth first search (DFS)

- Start from i , visit an unexplored neighbour j
- Suspend the exploration of i and explore j instead
- Continue till you reach a vertex with no unexplored neighbours
- Backtrack to nearest suspended vertex that still has an unexplored neighbour



Depth first search (DFS)

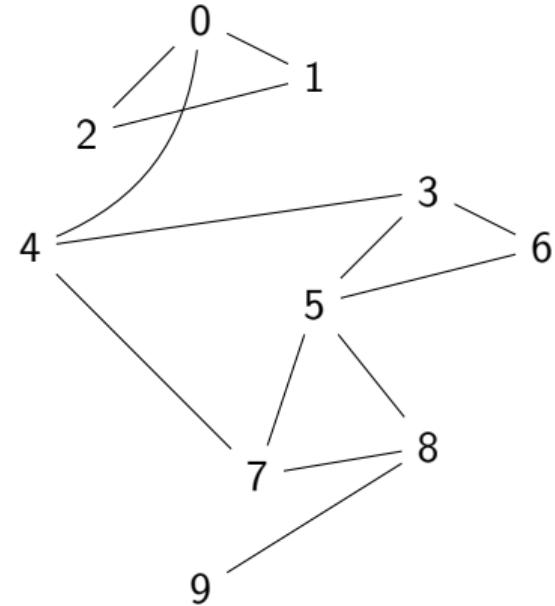
- Start from i , visit an unexplored neighbour j
- Suspend the exploration of i and explore j instead
- Continue till you reach a vertex with no unexplored neighbours
- Backtrack to nearest suspended vertex that still has an unexplored neighbour
- Suspended vertices are stored in a **stack**
 - Last in, first out
 - Most recently suspended is checked first



DFS from vertex 4

Visited	
0	False
1	False
2	False
3	False
4	False
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices								

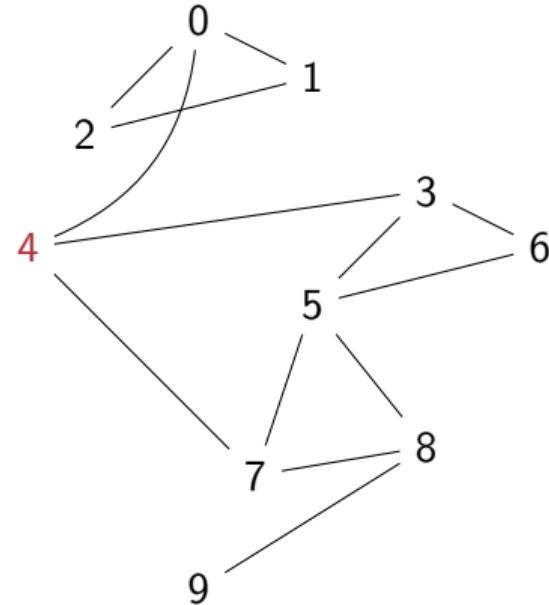


DFS from vertex 4

Visited	
0	False
1	False
2	False
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices								

■ Mark 4,

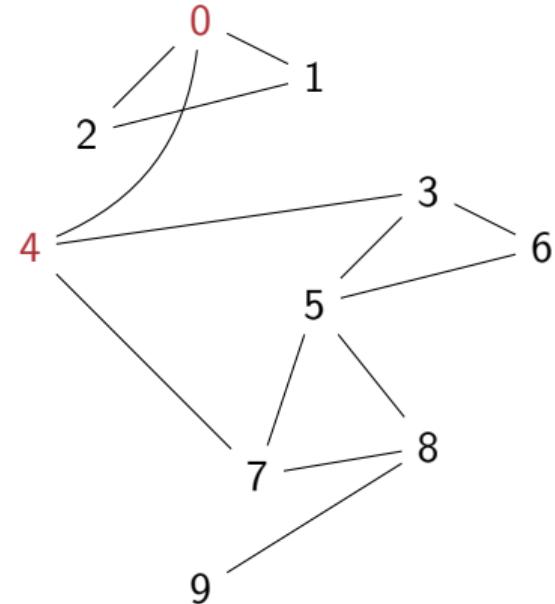


DFS from vertex 4

Visited	
0	True
1	False
2	False
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices								
4								

- Mark 4, Suspend 4, explore 0

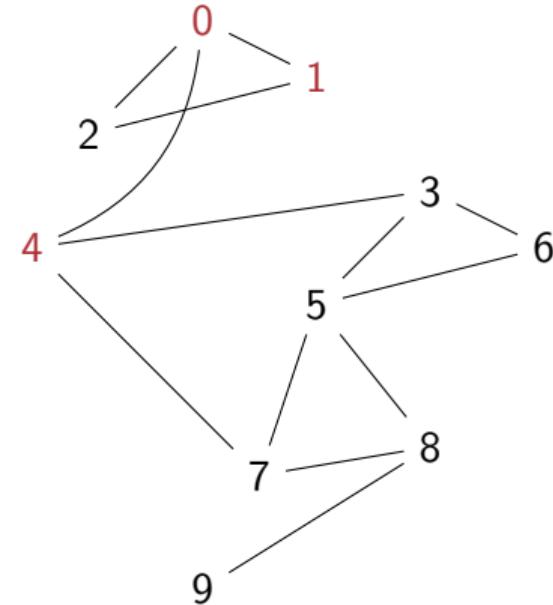


DFS from vertex 4

Visited	
0	True
1	True
2	False
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices	
4	0

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1

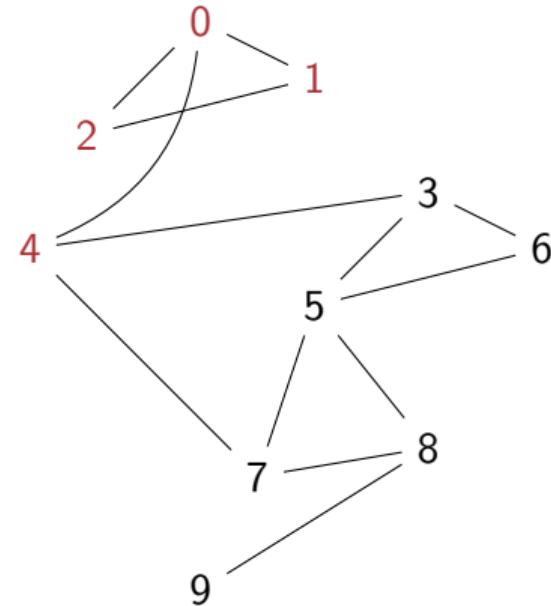


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices								
4	0	1						

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2

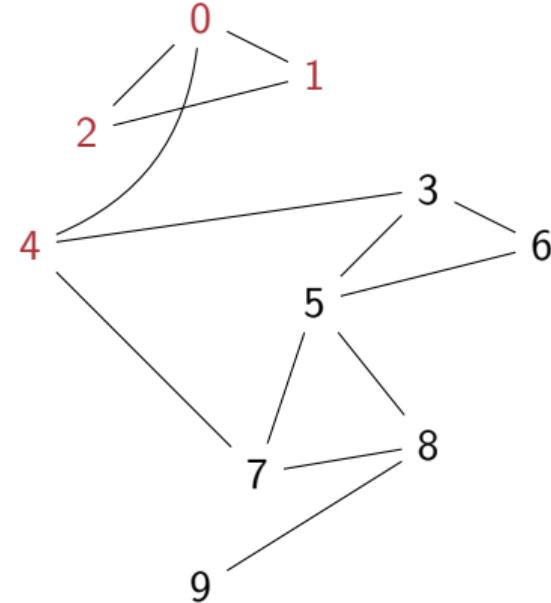


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices	
4	0

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1,

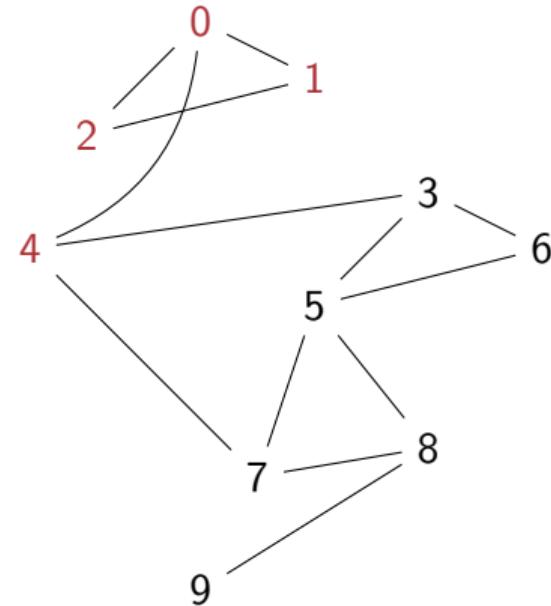


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	False
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices								
4								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0,

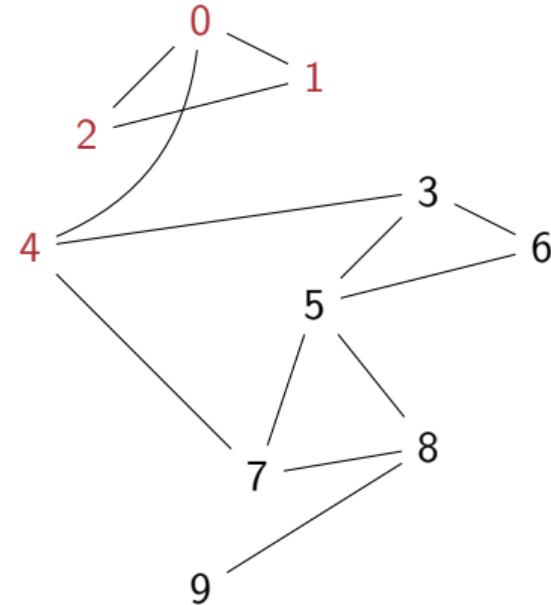


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4

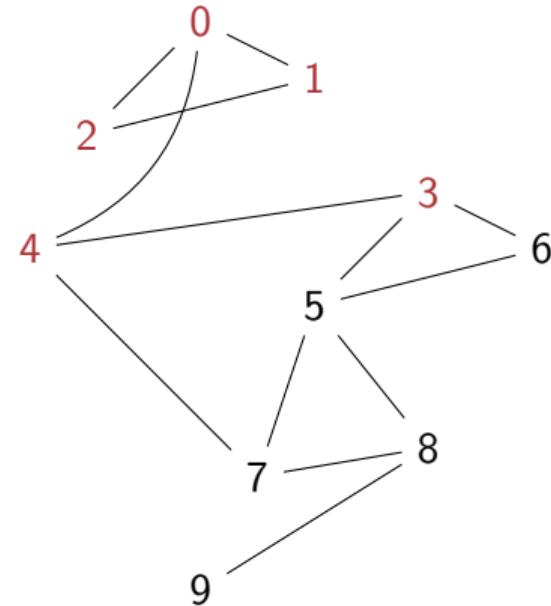


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	False
6	False
7	False
8	False
9	False

Stack of suspended vertices								
4								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3

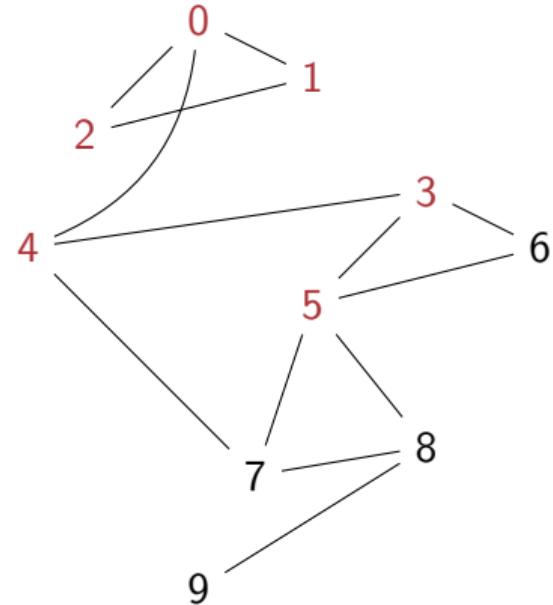


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	False
7	False
8	False
9	False

Stack of suspended vertices	
4	3

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5

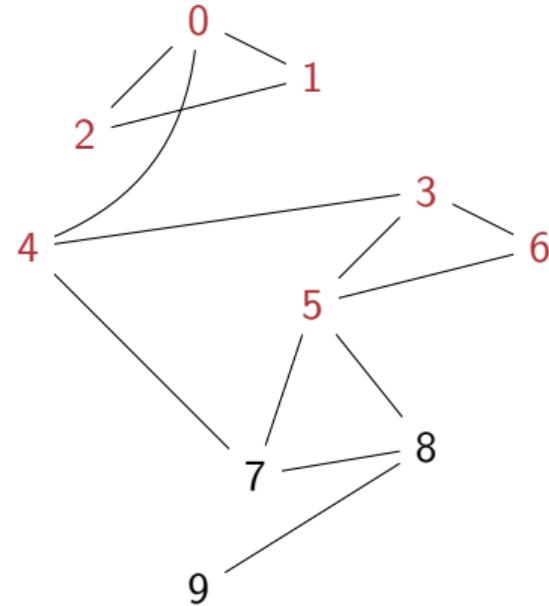


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	False
8	False
9	False

Stack of suspended vertices								
4	3	5						

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6

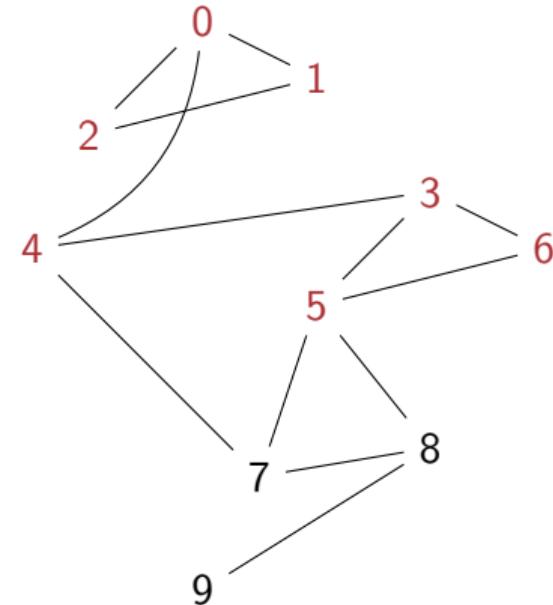


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	False
8	False
9	False

Stack of suspended vertices	
4	3

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5,

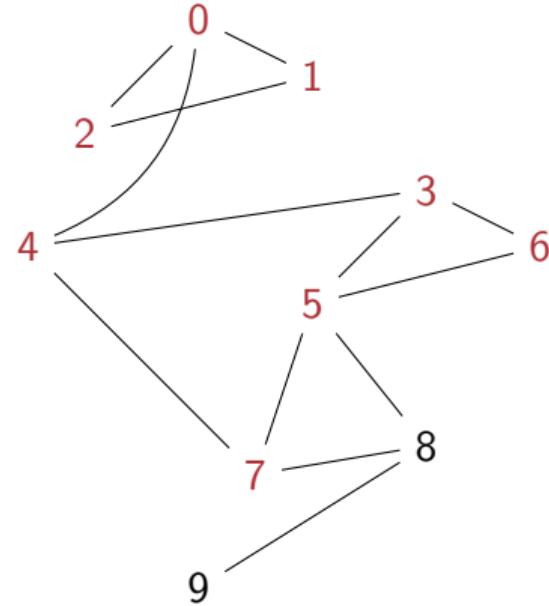


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	False
9	False

Stack of suspended vertices								
4	3	5						

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7

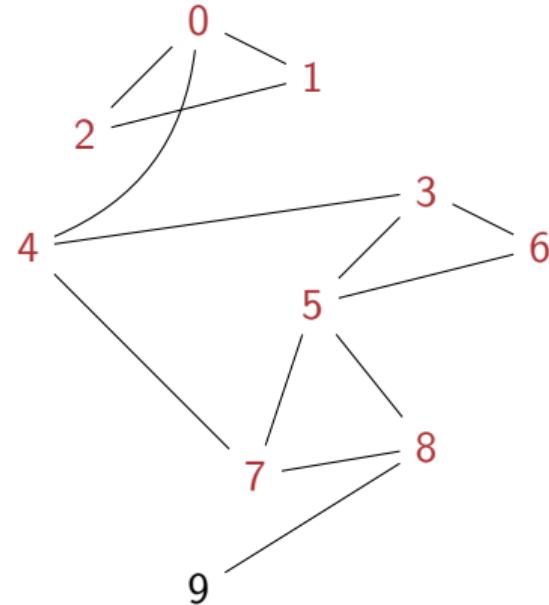


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	False

Stack of suspended vertices								
4	3	5	7					

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8

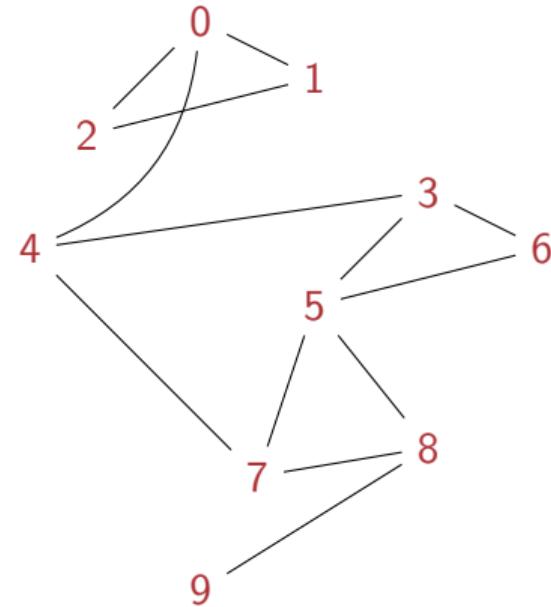


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices								
4	3	5	7	8				

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9

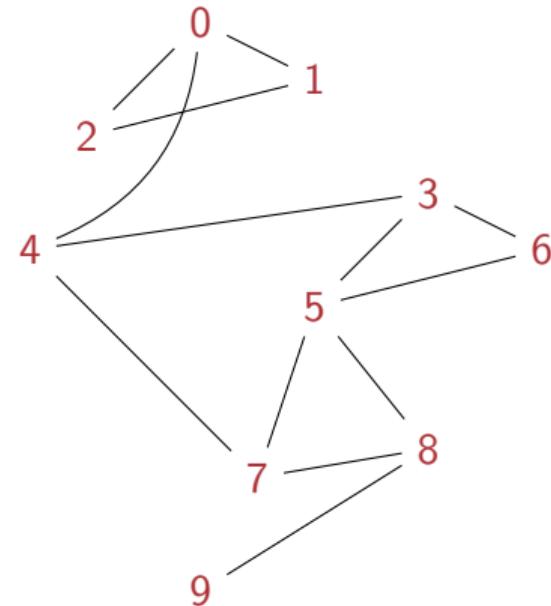


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices								
4	3	5	7					

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9
- Backtrack to 8,

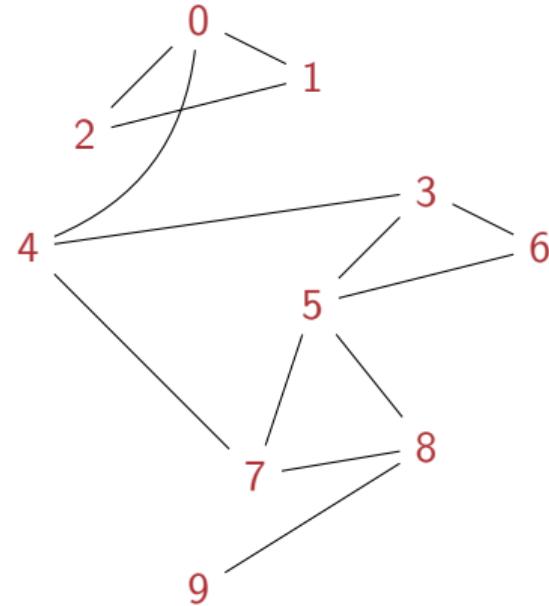


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices								
4	3	5						

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9
- Backtrack to 8, 7,

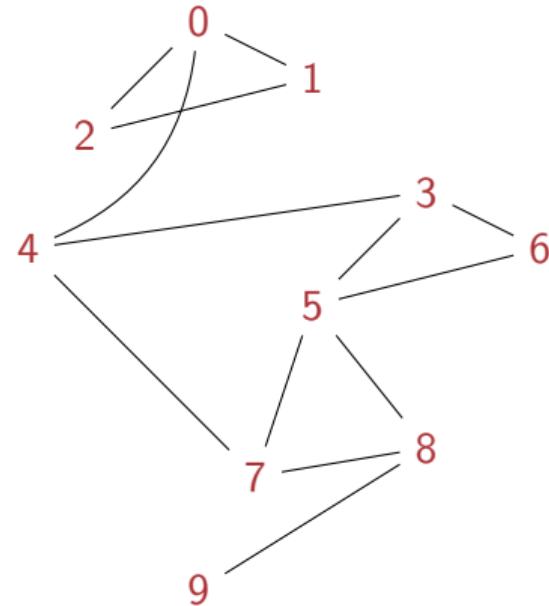


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices	
4	3

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9
- Backtrack to 8, 7, 5,

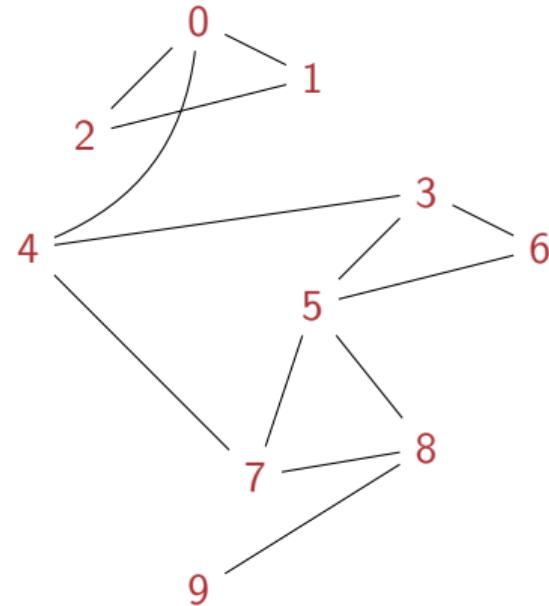


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices	
4	

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9
- Backtrack to 8, 7, 5, 3,

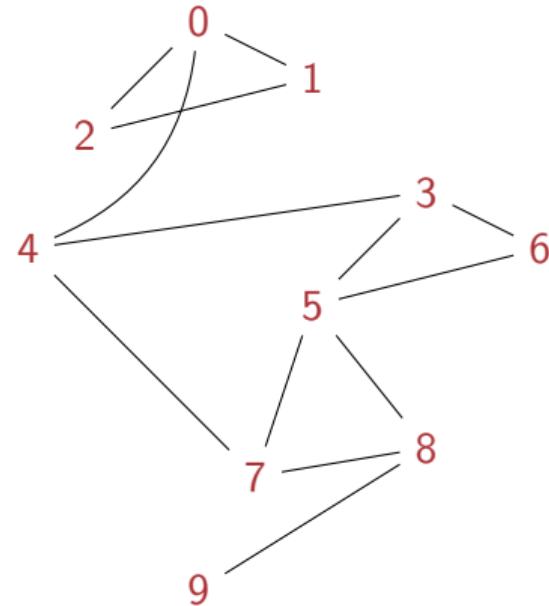


DFS from vertex 4

Visited	
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True

Stack of suspended vertices								

- Mark 4, Suspend 4, explore 0
- suspend 0, explore 1
- Suspend 1, explore 2
- Backtrack to 1, 0, 4
- Suspend 4, explore 3
- Suspend 3, explore 5
- Suspend 5, explore 6
- Backtrack to 5, suspend 5, explore 7
- Suspend 7, explore 8
- Suspend 8, explore 9
- Backtrack to 8, 7, 5, 3, 4



Implementing DFS

- DFS is most natural to implement recursively
 - For each unvisited neighbour of v , call $DFS(v)$

```
def DFSInit(AMat):  
    # Initialization  
    (rows,cols) = AMat.shape  
    (visited,parent) = ({},{} )  
    for i in range(rows):  
        visited[i] = False  
        parent[i] = -1  
    return(visited,parent)  
  
def DFS(AMat,visited,parent,v):  
    visited[v] = True  
  
    for k in neighbours(AMat,v):  
        if (not visited[k]):  
            parent[k] = v  
            (visited,parent) =  
                DFS(AMat,visited,parent,k)  
  
    return(visited,parent)
```

Implementing DFS

- DFS is most natural to implement recursively
 - For each unvisited neighbour of v , call $DFS(v)$
- No need to maintain a stack
 - Recursion implicitly maintains stack
 - Separate initialization step

```
def DFSInit(AMat):  
    # Initialization  
    (rows,cols) = AMat.shape  
    (visited,parent) = ({},{} )  
    for i in range(rows):  
        visited[i] = False  
        parent[i] = -1  
    return(visited,parent)  
  
def DFS(AMat,visited,parent,v):  
    visited[v] = True  
  
    for k in neighbours(AMat,v):  
        if (not visited[k]):  
            parent[k] = v  
            (visited,parent) =  
                DFS(AMat,visited,parent,k)  
  
    return(visited,parent)
```

Implementing DFS

- DFS is most natural to implement recursively
 - For each unvisited neighbour of v , call $DFS(v)$
- No need to maintain a stack
 - Recursion implicitly maintains stack
 - Separate initialization step
- Can make `visited` and `parent` global
 - Still need to initialize them according to the size of input adjacency matrix/list

```
(visited,parent) = ({},{})

def DFSInitGlobal(AMat):
    # Initialization
    (rows,cols) = AMat.shape
    for i in range(rows):
        visited[i] = False
        parent[i] = -1
    return

def DFSGlobal(AMat,v):
    visited[v] = True

    for k in neighbours(AMat,v):
        if (not visited[k]):
            parent[k] = v
            DFSGlobal(AMat,k)

    return
```

Implementing DFS

- DFS is most natural to implement recursively
 - For each unvisited neighbour of v , call $DFS(v)$
- No need to maintain a stack
 - Recursion implicitly maintains stack
 - Separate initialization step
- Can make `visited` and `parent` global
 - Still need to initialize them according to the size of input adjacency matrix/list
- Use an adjacency list instead

```
def DFSInitList(AList):  
    # Initialization  
    (visited,parent) = ({},{})  
    for i in AList.keys():  
        visited[i] = False  
        parent[i] = -1  
    return(visited,parent)  
  
def DFSList(AList,visited,parent,v):  
    visited[v] = True  
  
    for k in AList[v]:  
        if (not visited[k]):  
            parent[k] = v  
            (visited,parent) =  
                DFSList(AList,visited,parent,k)  
  
    return(visited,parent)
```

Implementing DFS

- DFS is most natural to implement recursively
 - For each unvisited neighbour of v , call $\text{DFS}(v)$
- No need to maintain a stack
 - Recursion implicitly maintains stack
 - Separate initialization step
- Can make `visited` and `parent` global
 - Still need to initialize them according to the size of input adjacency matrix/list
- Use an adjacency list instead

```
(visited,parent) = ({},{})

def DFSInitListGlobal(AList):
    # Initialization
    for i in AList.keys():
        visited[i] = False
        parent[i] = -1
    return

def DFSListGlobal(AList,v):
    visited[v] = True

    for k in AList[v]:
        if (not visited[k]):
            parent[k] = v
            DFSListGlobal(AList,k)

    return
```

Complexity of DFS

- Like BFS, each vertex is marked and explored once

Complexity of DFS

- Like BFS, each vertex is marked and explored once
- Exploring vertex v requires scanning all neighbours of v
 - $O(n)$ time for adjacency matrix, independent of $\text{degree}(v)$
 - $\text{degree}(v)$ time for adjacency list
 - Total time is $O(m)$ across all vertices

Complexity of DFS

- Like BFS, each vertex is marked and explored once
- Exploring vertex v requires scanning all neighbours of v
 - $O(n)$ time for adjacency matrix, independent of $\text{degree}(v)$
 - $\text{degree}(v)$ time for adjacency list
 - Total time is $O(m)$ across all vertices
- Overall complexity is same as BFS
 - $O(n^2)$ using adjacency matrix
 - $O(m + n)$ using adjacency list

Summary

- DFS is another systematic strategy to explore a graph

Summary

- DFS is another systematic strategy to explore a graph
- DFS uses a stack to suspend exploration and move to unexplored neighbours

Summary

- DFS is another systematic strategy to explore a graph
- DFS uses a stack to suspend exploration and move to unexplored neighbours
- Paths discovered by DFS are not shortest paths, unlike BFS

Summary

- DFS is another systematic strategy to explore a graph
- DFS uses a stack to suspend exploration and move to unexplored neighbours
- Paths discovered by DFS are not shortest paths, unlike BFS
- Useful features can be found by recording the order in which DFS visits vertices

Applications of BFS and DFS

Madhavan Mukund

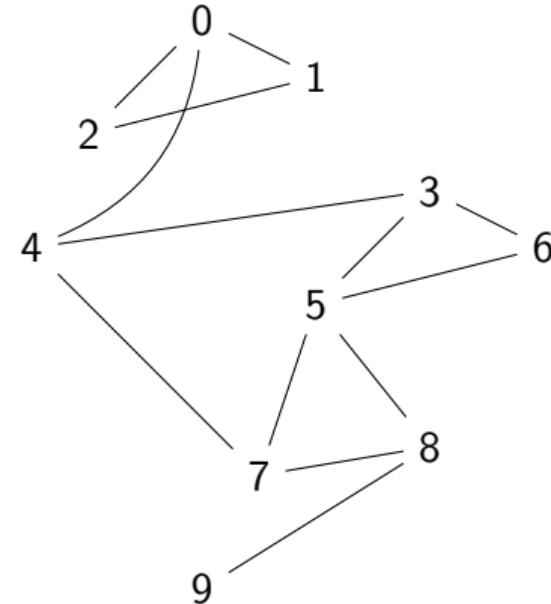
<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 4

BFS and DFS

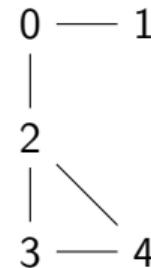
- BFS and DFS systematically compute reachability in graphs
- BFS works level by level
 - Discovers shortest paths in terms of number of edges
- DFS explores a vertex as soon as it is visited neighbours
 - Suspend a vertex while exploring its neighbours
 - DFS numbering describes the order in which vertices are explored
- Beyond reachability, what can we find out about a graph using BFS/DFS?



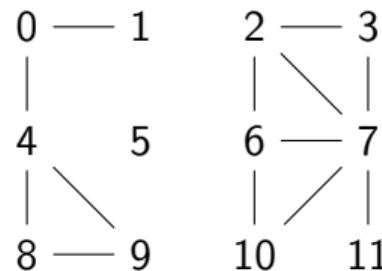
Connectivity

- An undirected graph is **connected** if every vertex is reachable from every other vertex

Connected Graph



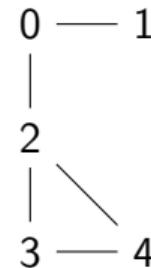
Disconnected Graph



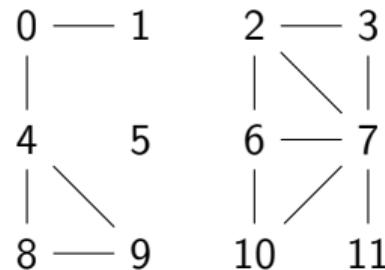
Connectivity

- An undirected graph is **connected** if every vertex is reachable from every other vertex
- In a disconnected graph, we can identify the connected components

Connected Graph



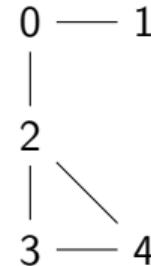
Disconnected Graph



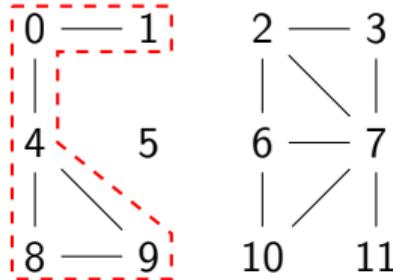
Connectivity

- An undirected graph is **connected** if every vertex is reachable from every other vertex
- In a disconnected graph, we can identify the connected components
 - Maximal subsets of vertices that are connected

Connected Graph



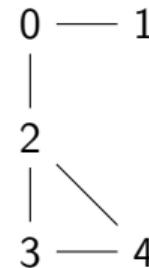
Disconnected Graph



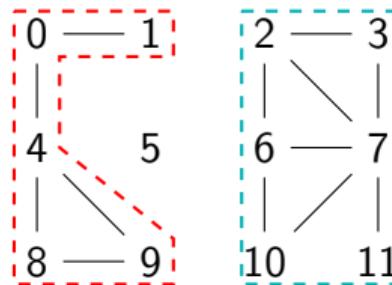
Connectivity

- An undirected graph is **connected** if every vertex is reachable from every other vertex
- In a disconnected graph, we can identify the connected components
 - Maximal subsets of vertices that are connected

Connected Graph



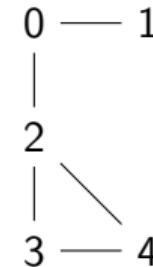
Disconnected Graph



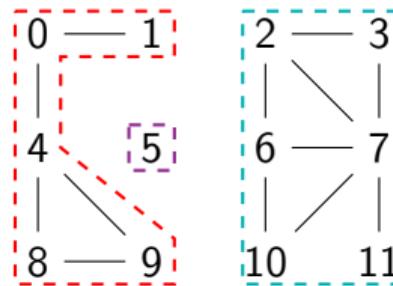
Connectivity

- An undirected graph is **connected** if every vertex is reachable from every other vertex
- In a disconnected graph, we can identify the connected components
 - Maximal subsets of vertices that are connected
 - Isolated vertices are trivial components

Connected Graph



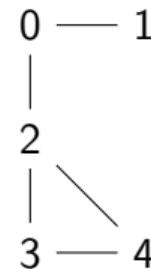
Disconnected Graph



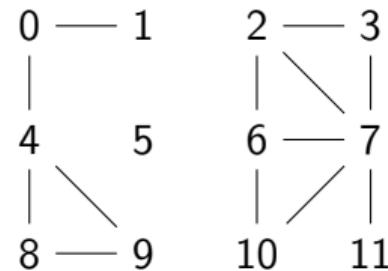
Identifying connected components

- Assign each vertex a component number

Connected Graph



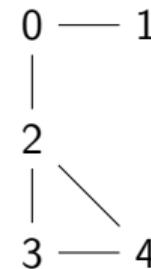
Disconnected Graph



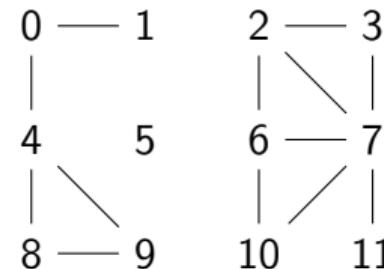
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0

Connected Graph



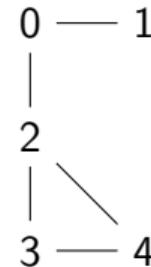
Disconnected Graph



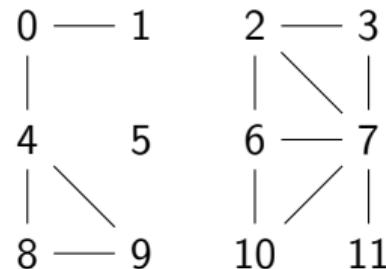
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
 - Initialize component number to 0

Connected Graph



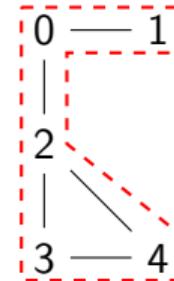
Disconnected Graph



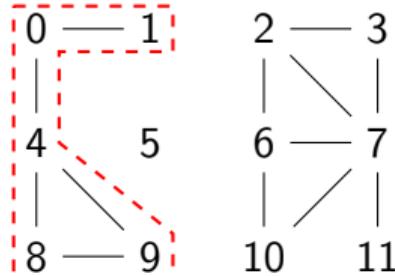
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
 - Initialize component number to 0
 - All visited nodes form a connected component

Connected Graph



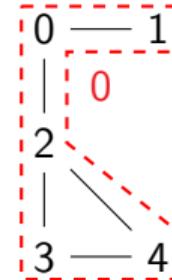
Disconnected Graph



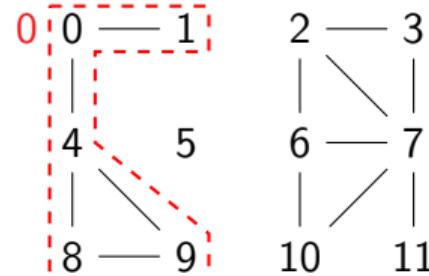
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
 - Initialize component number to 0
 - All visited nodes form a connected component
 - Assign each visited node component number 0

Connected Graph



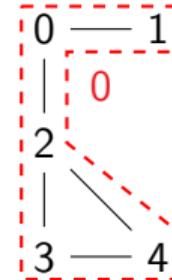
Disconnected Graph



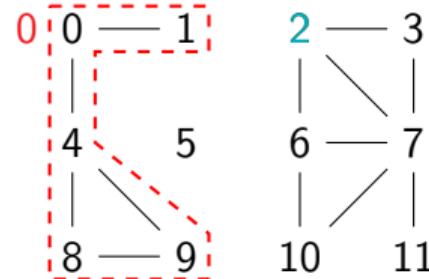
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
 - Initialize component number to 0
 - All visited nodes form a connected component
 - Assign each visited node component number 0
- Pick smallest unvisited node j

Connected Graph



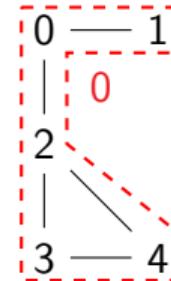
Disconnected Graph



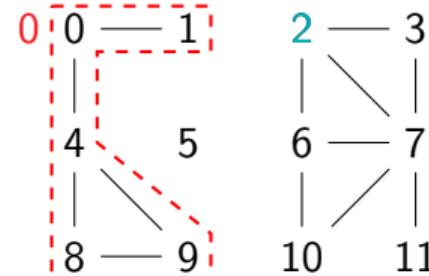
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
 - Initialize component number to 0
 - All visited nodes form a connected component
 - Assign each visited node component number 0
- Pick smallest unvisited node j
 - Increment component number to 1

Connected Graph



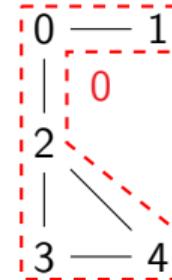
Disconnected Graph



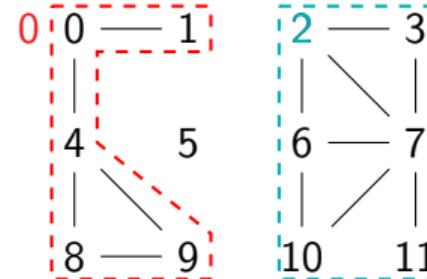
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
 - Initialize component number to 0
 - All visited nodes form a connected component
 - Assign each visited node component number 0
- Pick smallest unvisited node j
 - Increment component number to 1
 - Run BFS/DFS from node j

Connected Graph



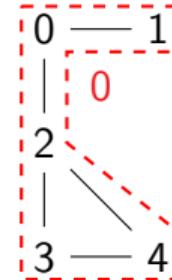
Disconnected Graph



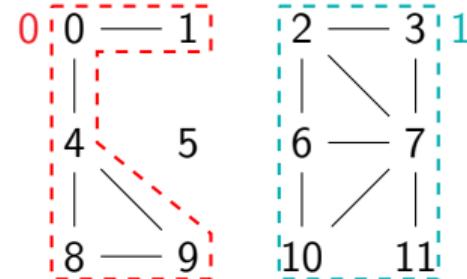
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
 - Initialize component number to 0
 - All visited nodes form a connected component
 - Assign each visited node component number 0
- Pick smallest unvisited node j
 - Increment component number to 1
 - Run BFS/DFS from node j
 - Assign each visited node component number 1

Connected Graph



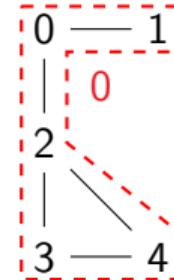
Disconnected Graph



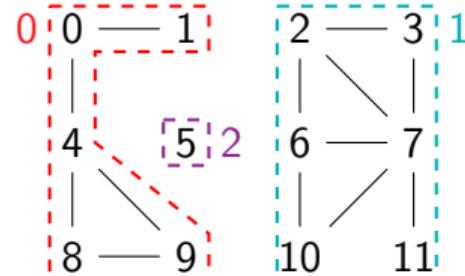
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
 - Initialize component number to 0
 - All visited nodes form a connected component
 - Assign each visited node component number 0
- Pick smallest unvisited node j
 - Increment component number to 1
 - Run BFS/DFS from node j
 - Assign each visited node component number 1
- Repeat until all nodes are visited

Connected Graph



Disconnected Graph



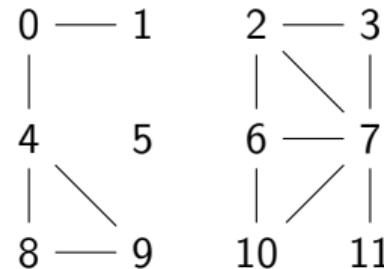
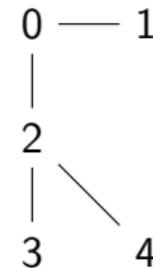
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
 - Initialize component number to 0
 - All visited nodes form a connected component
 - Assign each visited node component number 0
- Pick smallest unvisited node j
 - Increment component number to 1
 - Run BFS/DFS from node j
 - Assign each visited node component number 1
- Repeat until all nodes are visited

```
def Components(AList):  
    component = {}  
    for i in AList.keys():  
        component[i] = -1  
  
(compid,seen) = (0,0)  
  
while seen <= max(AList.keys()):  
    startv = min([i for i in AList.keys()  
                 if component[i] == -1])  
    visited = BFSList(AList,startv)  
    for i in visited.keys():  
        if visited[i]:  
            seen = seen + 1  
            component[i] = compid  
    compid = compid + 1  
  
return(component)
```

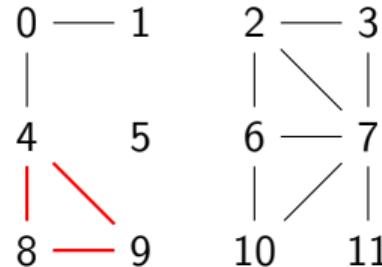
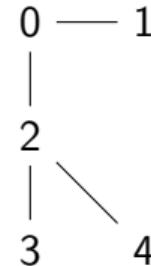
Detecting cycles

- A **cycle** is a path (technically, a walk) that starts and ends at the same vertex



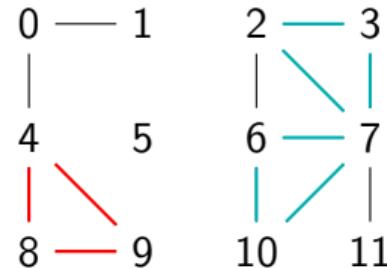
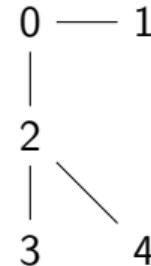
Detecting cycles

- A **cycle** is a path (technically, a walk) that starts and ends at the same vertex
 - $4 - 8 - 9 - 4$ is a cycle



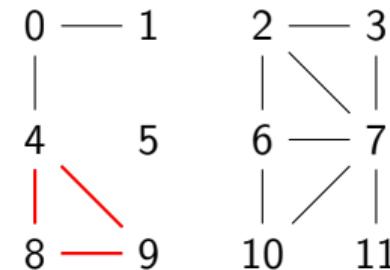
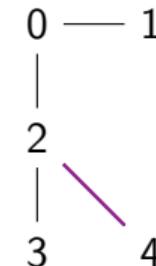
Detecting cycles

- A **cycle** is a path (technically, a walk) that starts and ends at the same vertex
 - $4 - 8 - 9 - 4$ is a cycle
 - Cycle may repeat a vertex:
 $2 - 3 - 7 - 10 - 6 - 7 - 2$



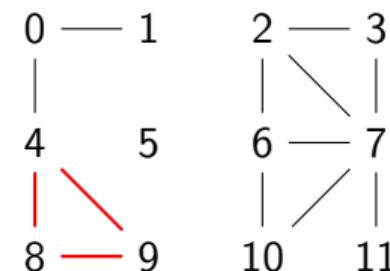
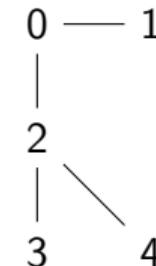
Detecting cycles

- A **cycle** is a path (technically, a walk) that starts and ends at the same vertex
 - $4 - 8 - 9 - 4$ is a cycle
 - Cycle may repeat a vertex:
 $2 - 3 - 7 - 10 - 6 - 7 - 2$
 - Cycle should not repeat edges: $i - j - i$ is **not** a cycle, e.g., $2 - 4 - 2$



Detecting cycles

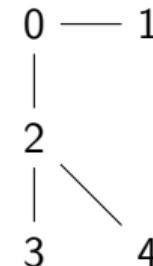
- A **cycle** is a path (technically, a walk) that starts and ends at the same vertex
 - $4 - 8 - 9 - 4$ is a cycle
 - Cycle may repeat a vertex:
 $2 - 3 - 7 - 10 - 6 - 7 - 2$
 - Cycle should not repeat edges: $i - j - i$ is **not** a cycle, e.g., $2 - 4 - 2$
 - **Simple cycle** — only repeated vertices are start and end



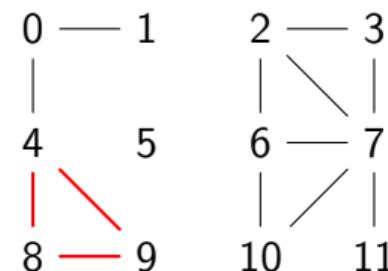
Detecting cycles

- A **cycle** is a path (technically, a walk) that starts and ends at the same vertex
 - $4 - 8 - 9 - 4$ is a cycle
 - Cycle may repeat a vertex:
 $2 - 3 - 7 - 10 - 6 - 7 - 2$
 - Cycle should not repeat edges: $i - j - i$ is **not** a cycle, e.g., $2 - 4 - 2$
 - **Simple cycle** — only repeated vertices are start and end
- A graph is acyclic if it has no cycles

Acyclic Graph



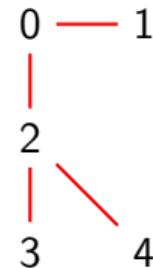
Graph with cycles



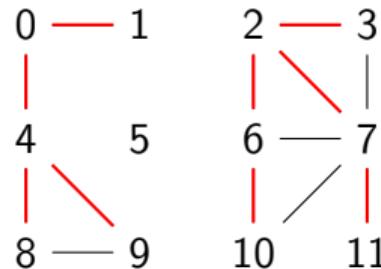
BFS tree

- Edges explored by BFS form a **tree**
 - Technically, one tree per component
 - Collection of trees is a **forest**

Acyclic Graph



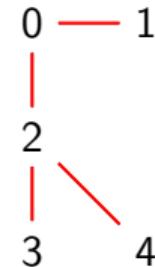
Graph with cycles



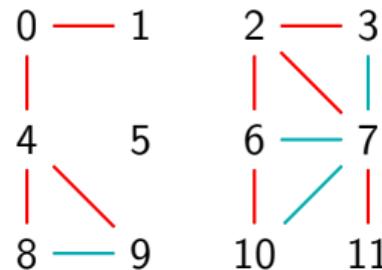
BFS tree

- Edges explored by BFS form a **tree**
 - Technically, one tree per component
 - Collection of trees is a **forest**
- Any non-tree edge creates a cycle
 - Detect cycles by searching for non-tree edges

Acyclic Graph

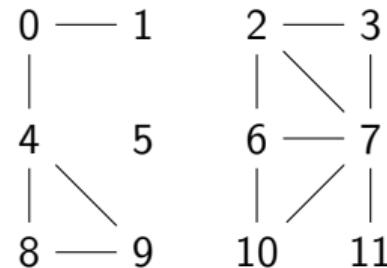


Graph with cycles



DFS tree

- Maintain a DFS counter, initially 0

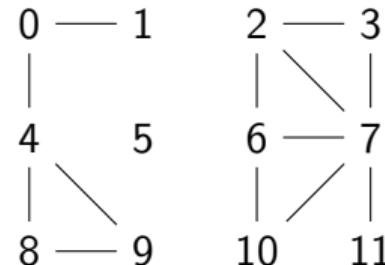


DFS tree

- Maintain a DFS counter, initially

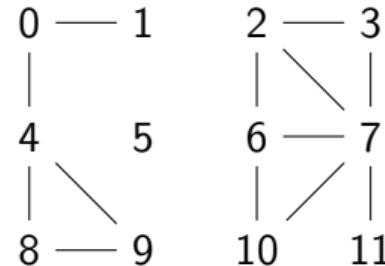
0

- Increment counter each time we start and finish exploring a node



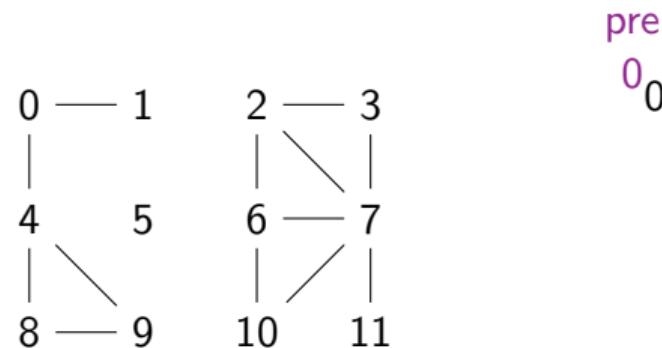
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



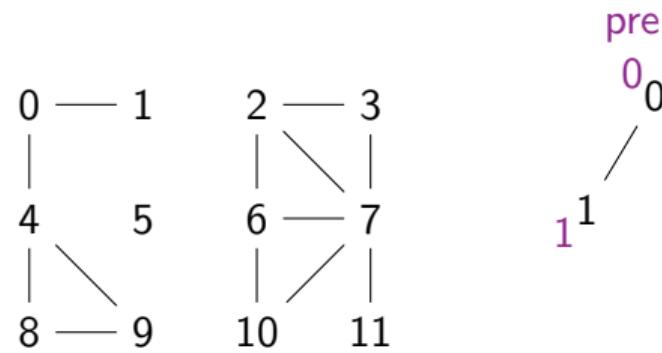
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



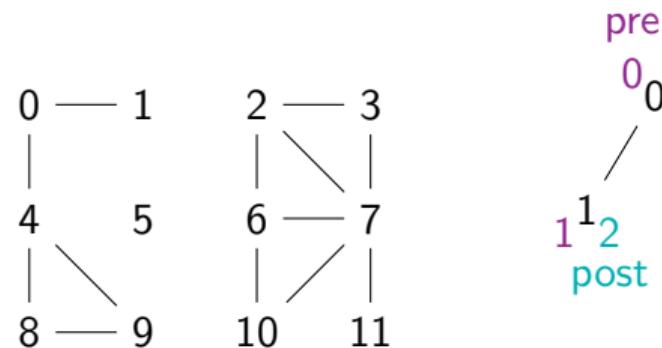
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



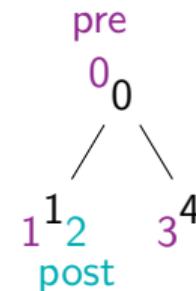
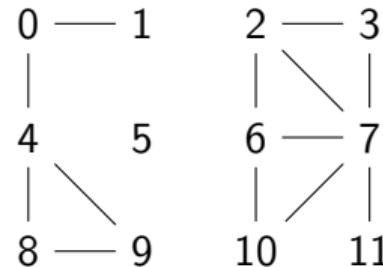
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



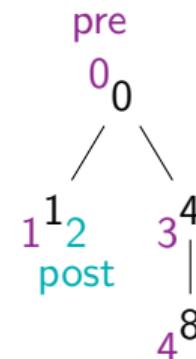
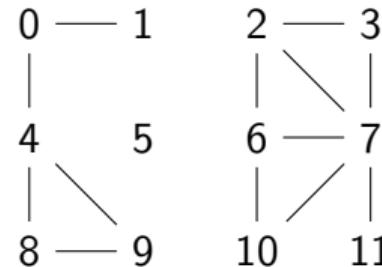
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



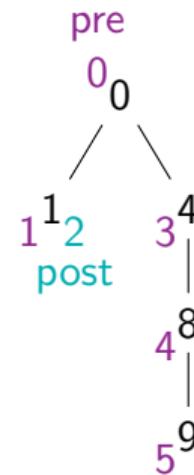
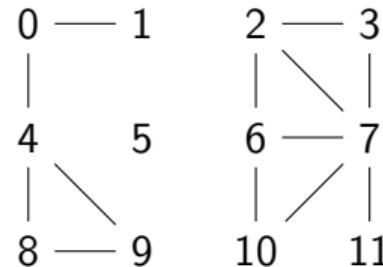
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



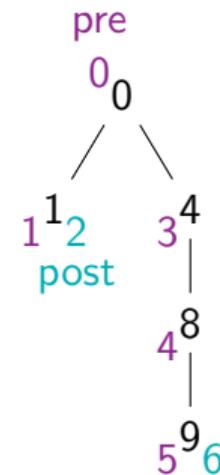
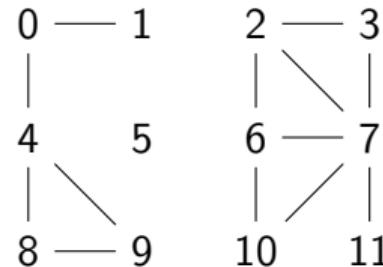
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



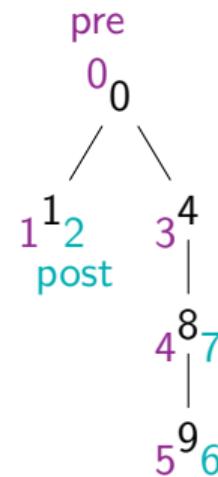
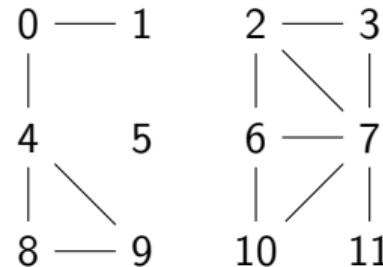
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



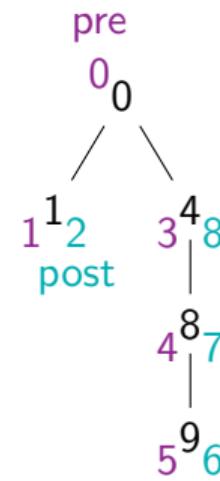
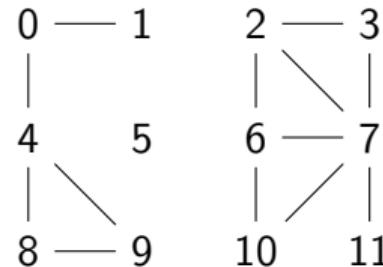
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



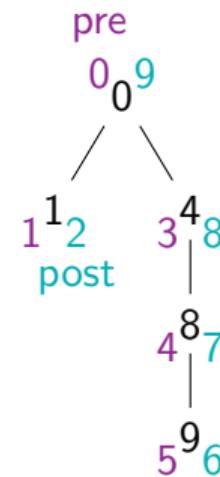
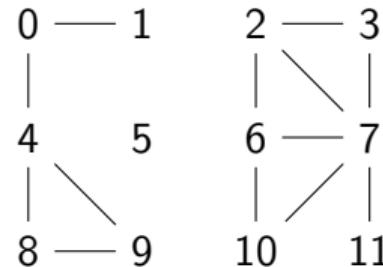
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



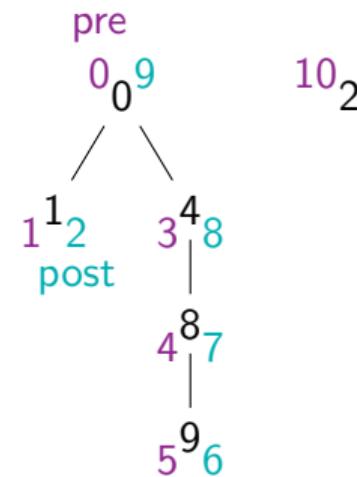
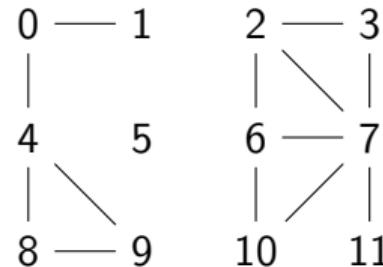
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



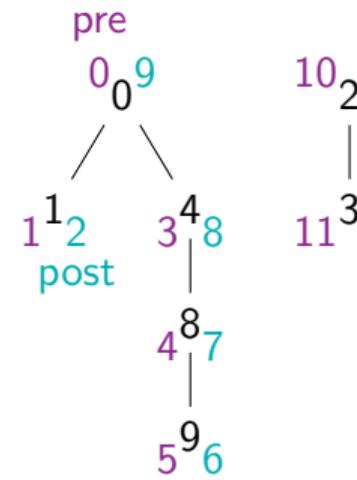
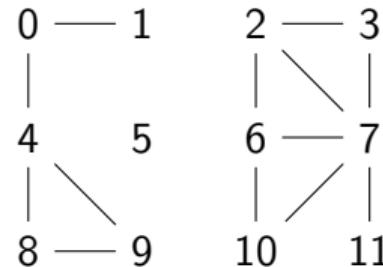
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



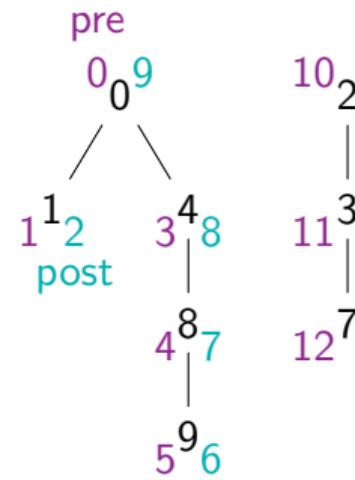
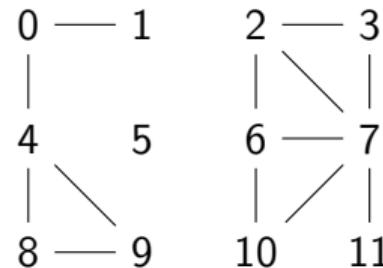
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



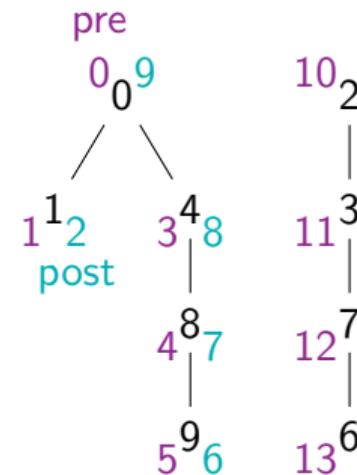
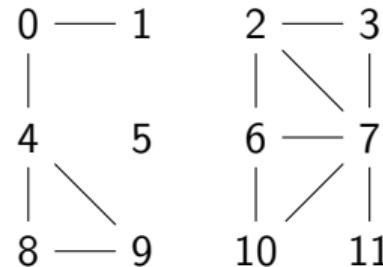
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



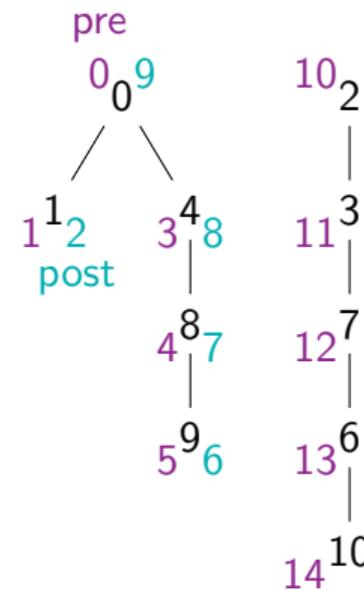
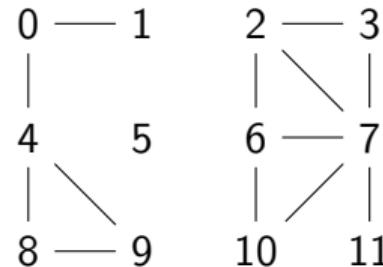
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



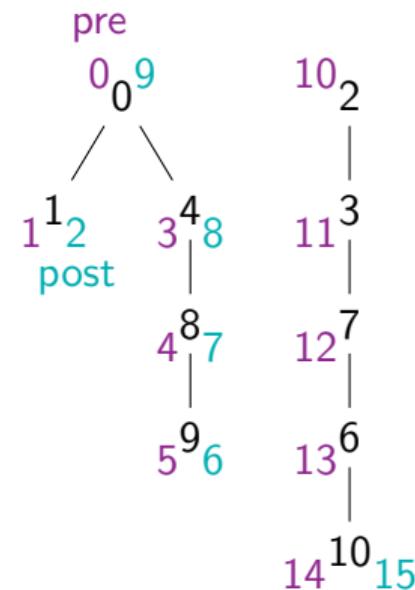
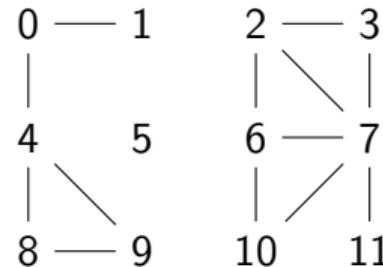
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



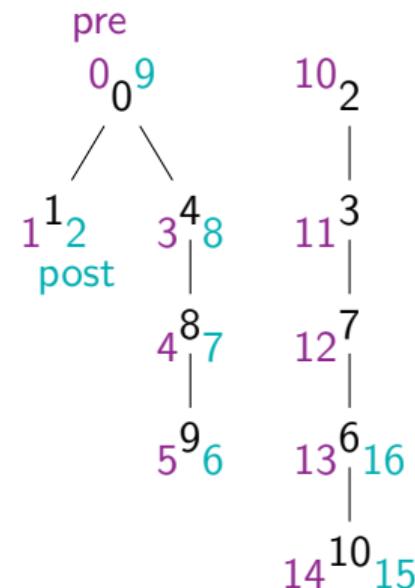
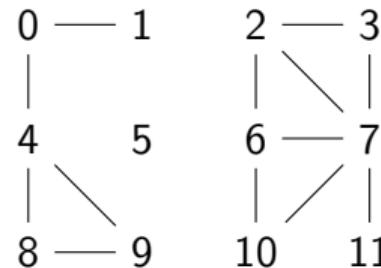
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



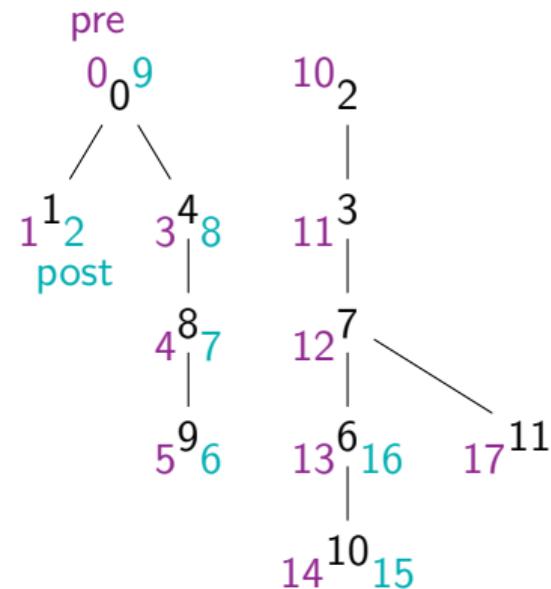
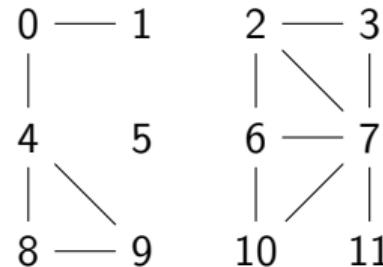
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



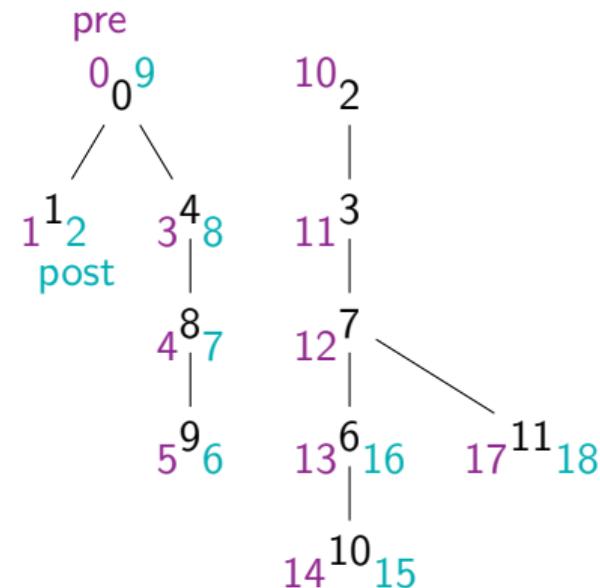
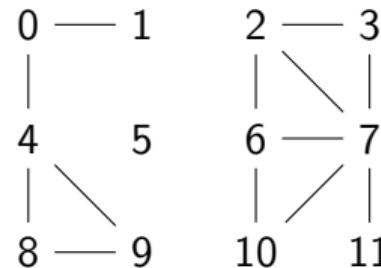
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



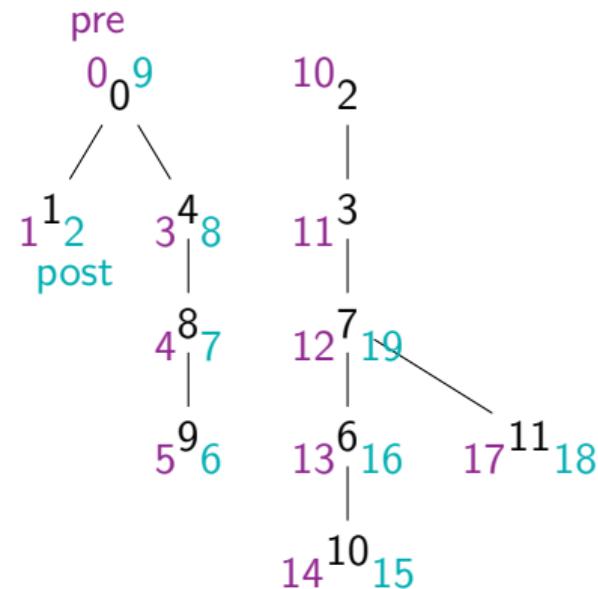
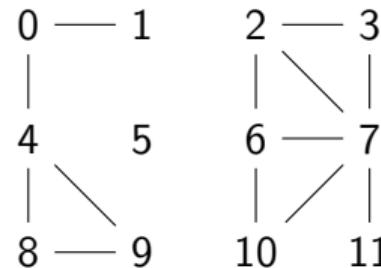
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



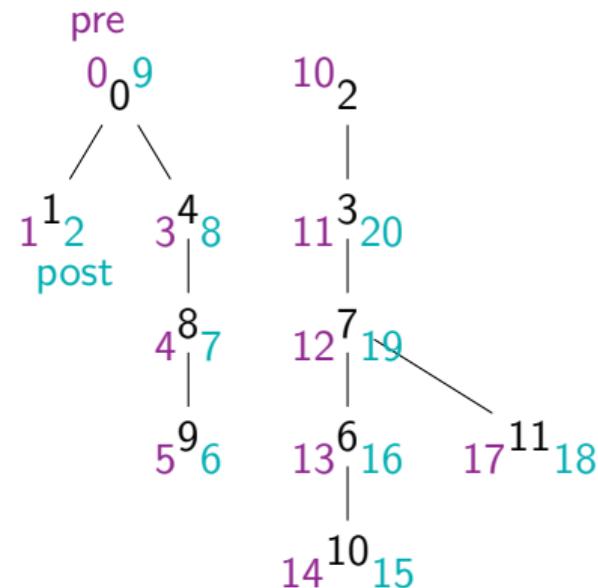
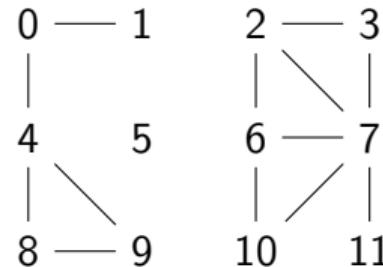
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



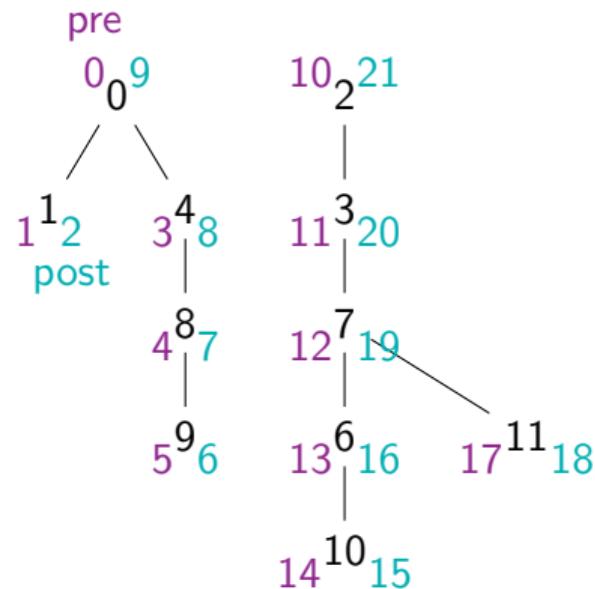
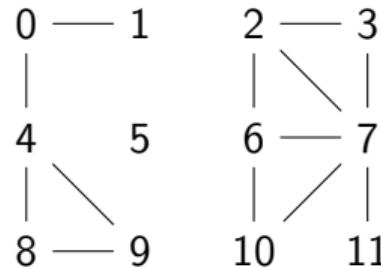
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



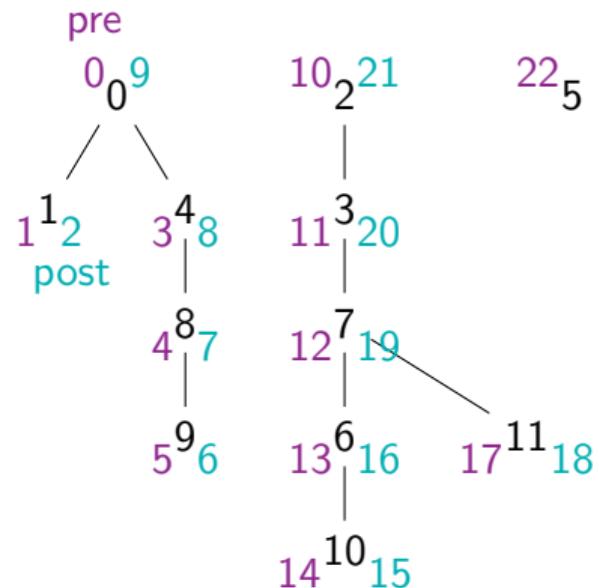
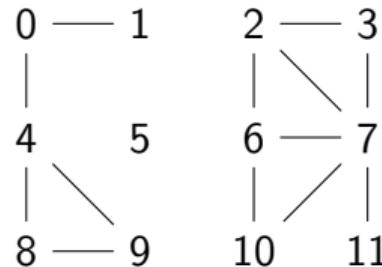
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



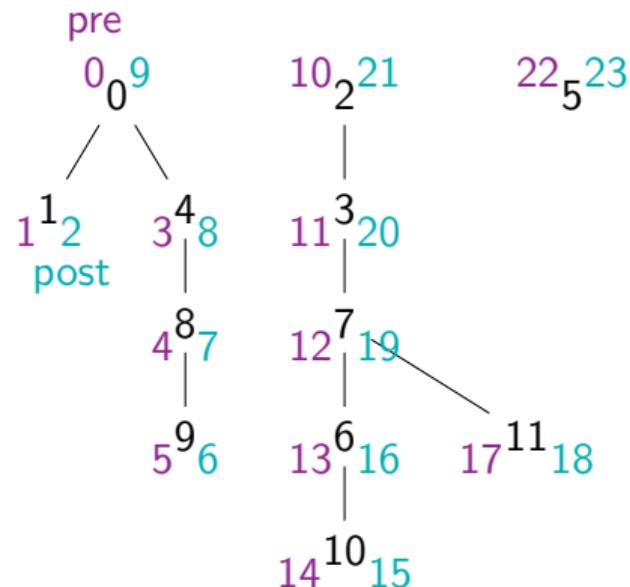
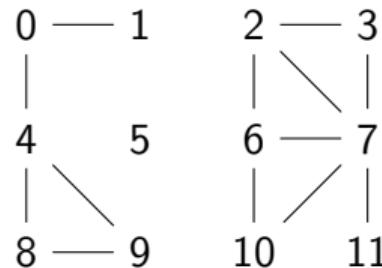
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)



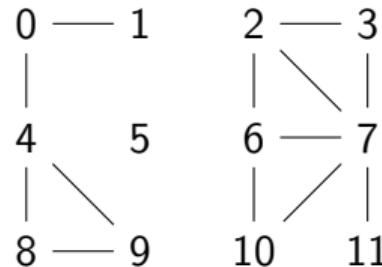
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)

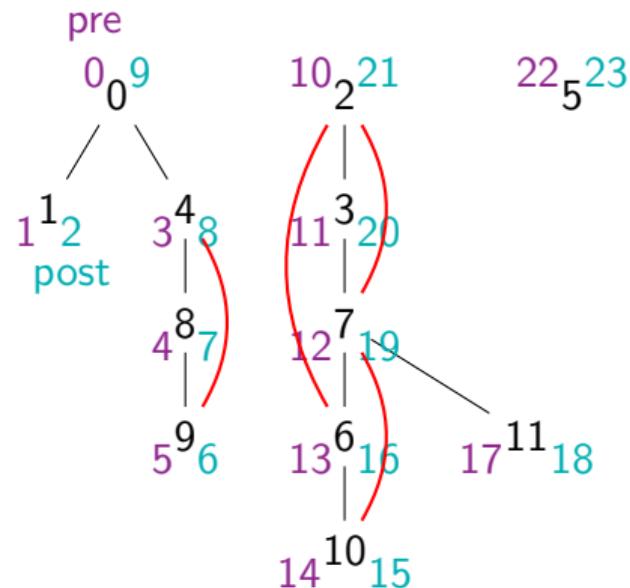


DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (**pre**) and exit number (**post**)

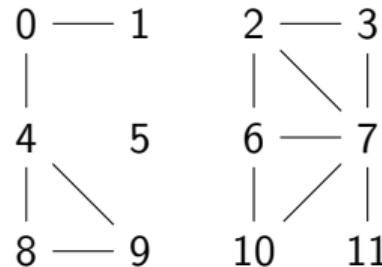


- As before, non-tree edges generate cycles



DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (`pre`) and exit number (`post`)



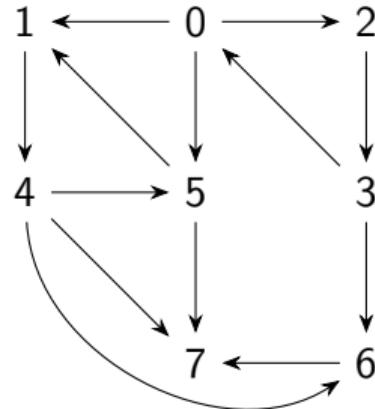
- As before, non-tree edges generate cycles
- To compute `pre` and `post` pass counter via recursive DFS calls

```
(visited,pre,post) = ({},[],[])
```

```
def DFSInitPrePost(AList):  
    # Initialization  
    for i in AList.keys():  
        visited[i] = False  
        pre[i],post[i]) = (-1,-1)  
    return  
  
def DFSPrePost(AList,v,count):  
    visited[v] = True  
    pre[v] = count  
    count = count+1  
    for k in AList[v]:  
        if (not visited[k]):  
            count = DFSPrePost(AList,k,count)  
    post[v] = count  
    count = count+1  
    return(count)
```

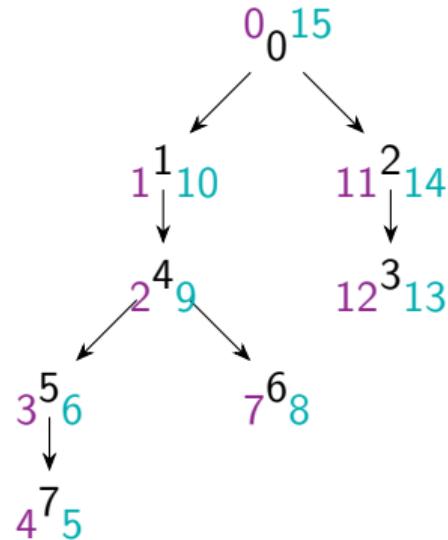
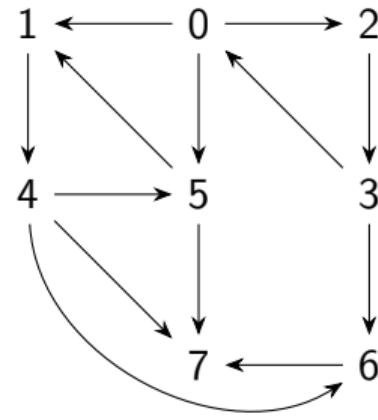
Directed cycles

- In a directed graph, a cycle must follow same direction
 - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
 - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not



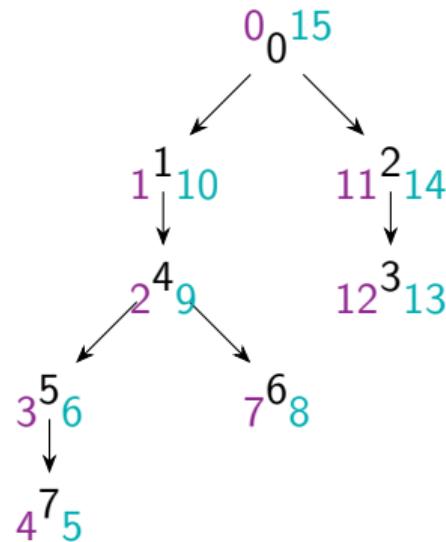
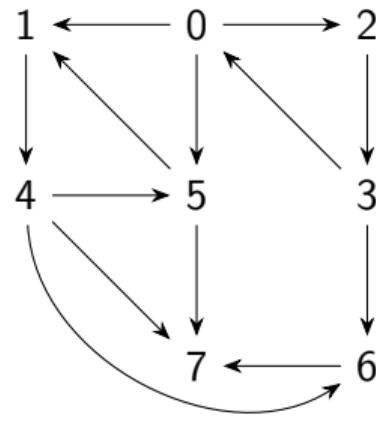
Directed cycles

- In a directed graph, a cycle must follow same direction
 - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
 - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not



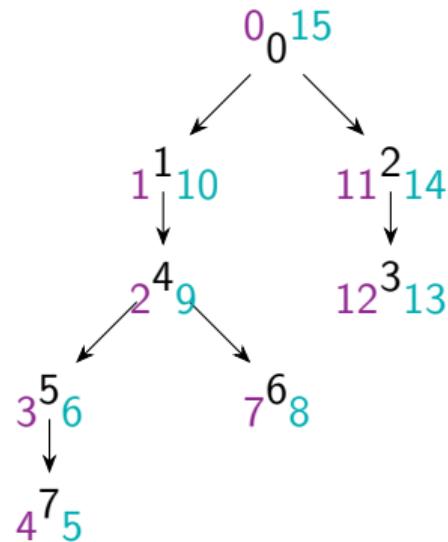
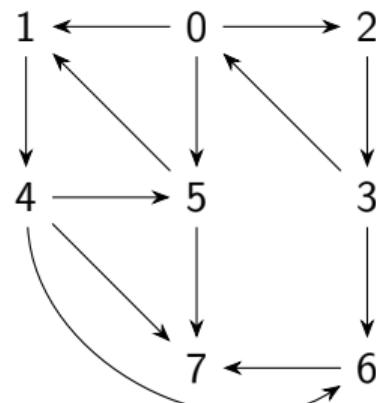
Directed cycles

- In a directed graph, a cycle must follow same direction
 - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
 - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not
- Tree edges



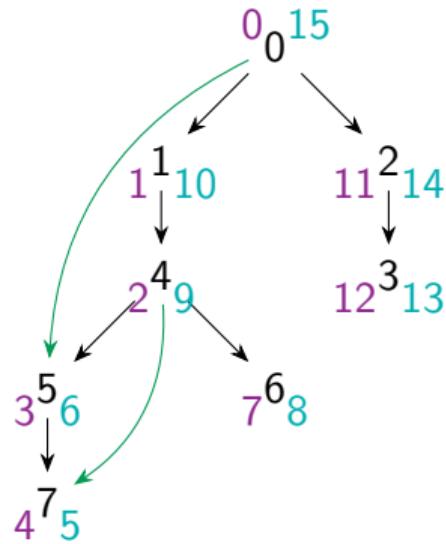
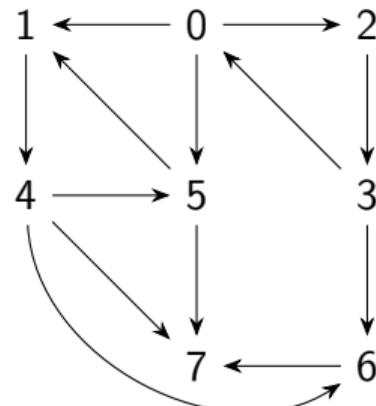
Directed cycles

- In a directed graph, a cycle must follow same direction
 - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
 - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges



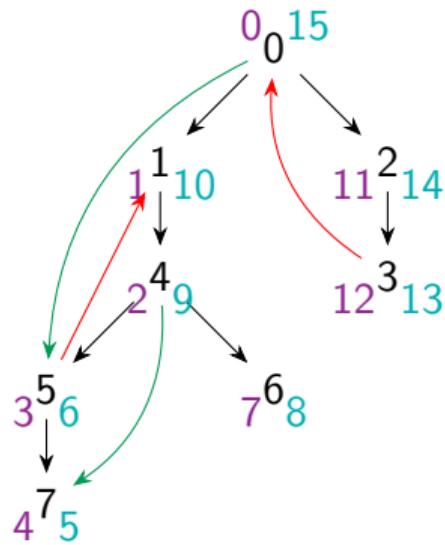
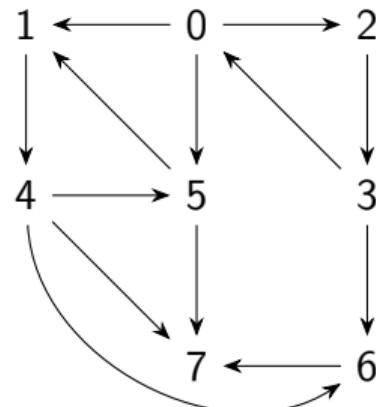
Directed cycles

- In a directed graph, a cycle must follow same direction
 - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
 - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges
 - Forward edges



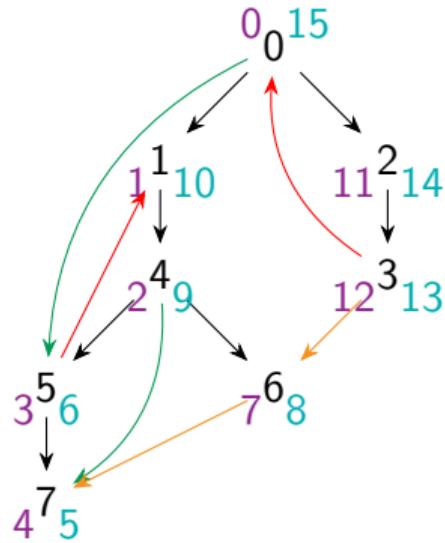
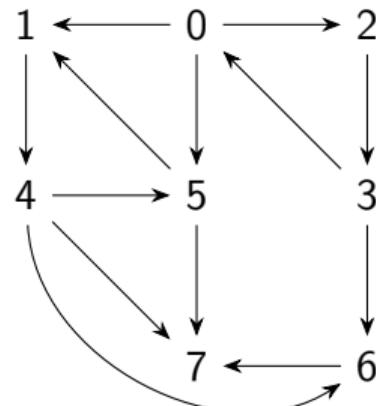
Directed cycles

- In a directed graph, a cycle must follow same direction
 - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
 - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges
 - Forward edges
 - Back edges



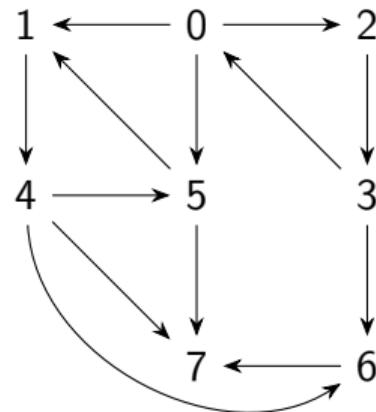
Directed cycles

- In a directed graph, a cycle must follow same direction
 - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
 - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges
 - Forward edges
 - Back edges
 - Cross edges

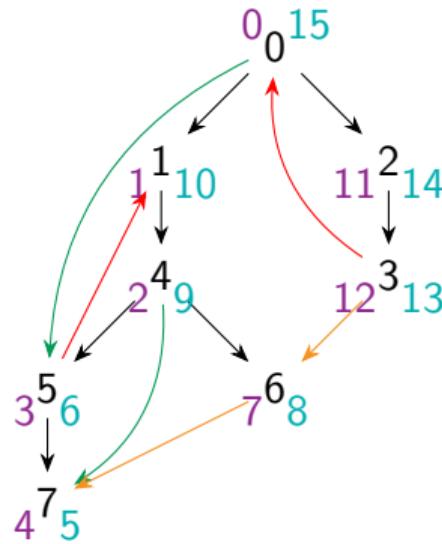


Directed cycles

- In a directed graph, a cycle must follow same direction
 - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
 - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not
 - Tree edges
 - Different types of non-tree edges
 - Forward edges
 - Back edges
 - Cross edges

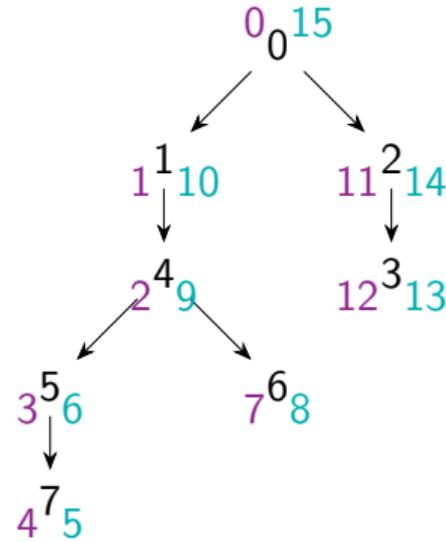
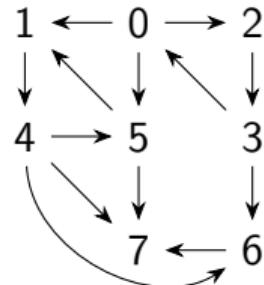


- Only back edges correspond to cycles



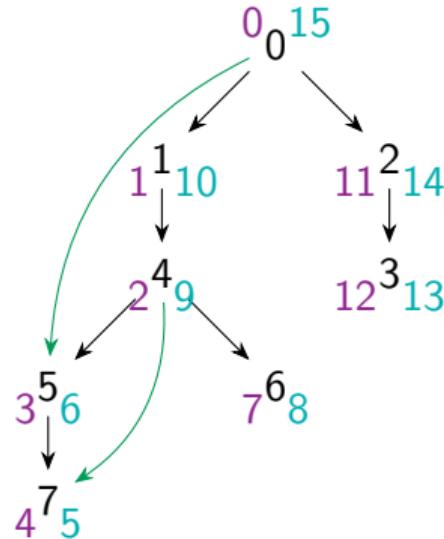
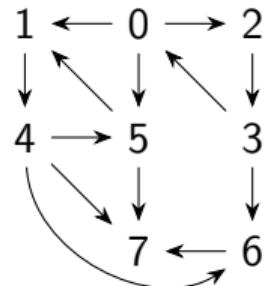
Classifying non-tree edges in directed graphs

- Use pre/post numbers



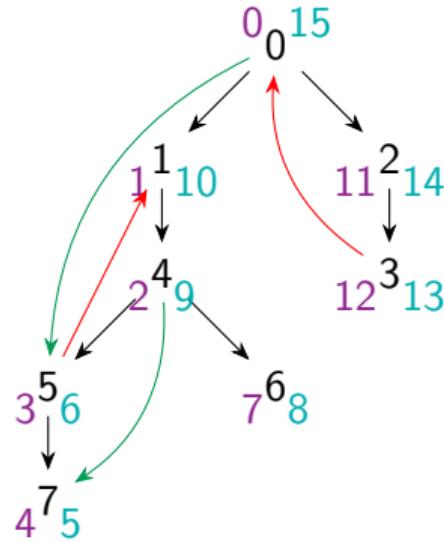
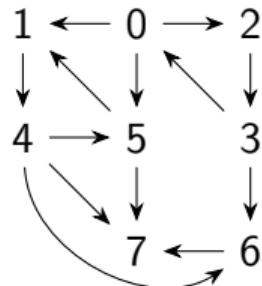
Classifying non-tree edges in directed graphs

- Use pre/post numbers
- Tree edge/forward edge (u, v)
Interval $[\text{pre}(u), \text{post}(u)]$ contains
 $[\text{pre}(v), \text{post}(v)]$



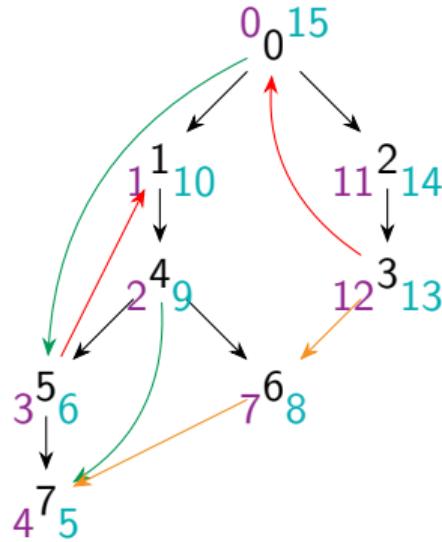
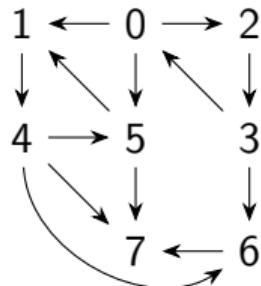
Classifying non-tree edges in directed graphs

- Use pre/post numbers
- Tree edge/forward edge (u, v)
Interval $[\text{pre}(u), \text{post}(u)]$ contains
 $[\text{pre}(v), \text{post}(v)]$
- Back edge (u, v)
Interval $[\text{pre}(v), \text{post}(v)]$ contains
 $[\text{pre}(u), \text{post}(u)]$



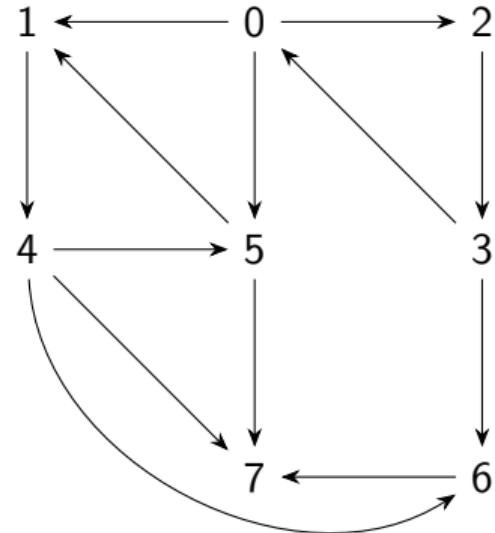
Classifying non-tree edges in directed graphs

- Use pre/post numbers
- Tree edge/forward edge (u, v)
Interval $[\text{pre}(u), \text{post}(u)]$ contains $[\text{pre}(v), \text{post}(v)]$
- Back edge (u, v)
Interval $[\text{pre}(v), \text{post}(v)]$ contains $[\text{pre}(u), \text{post}(u)]$
- Cross edge (u, v)
Intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint



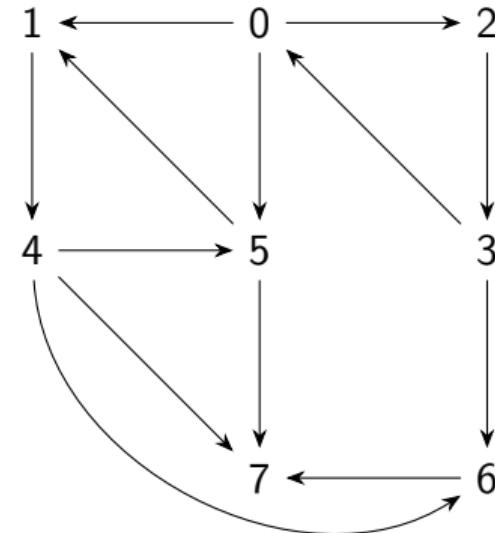
Connectivity in directed graphs

- Take directions into account



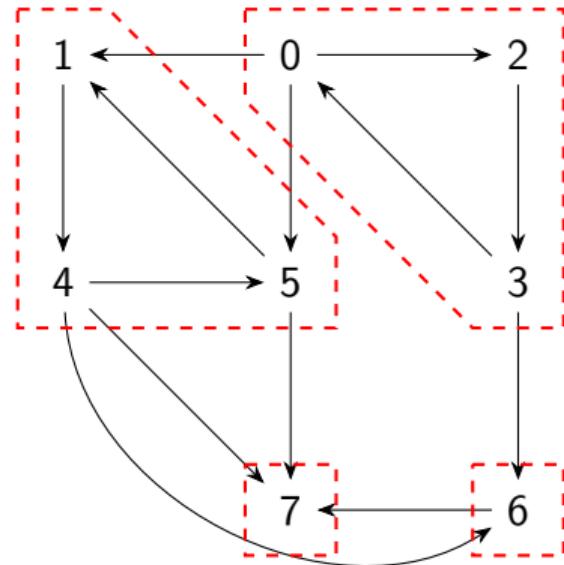
Connectivity in directed graphs

- Take directions into account
- Vertices i and j are **strongly connected** if there is a path from i to j and a path from j to i



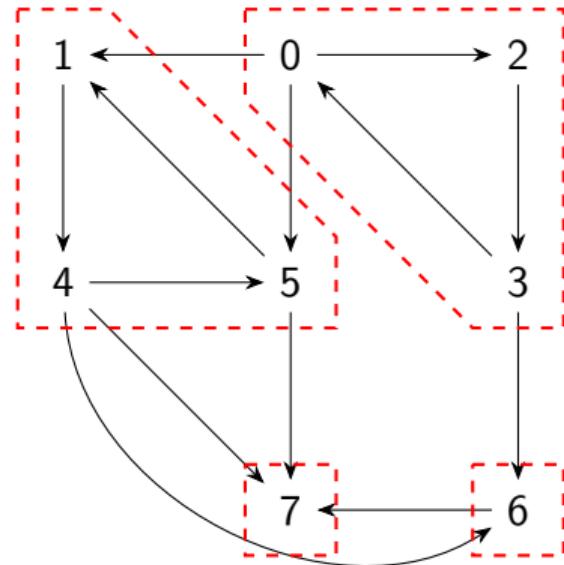
Connectivity in directed graphs

- Take directions into account
- Vertices i and j are **strongly connected** if there is a path from i to j and a path from j to i
- Directed graphs can be decomposed into **strongly connected components (SCCs)**
 - Within an SCC, each pair of vertices is strongly connected



Connectivity in directed graphs

- Take directions into account
- Vertices i and j are **strongly connected** if there is a path from i to j and a path from j to i
- Directed graphs can be decomposed into **strongly connected components (SCCs)**
 - Within an SCC, each pair of vertices is strongly connected
- DFS numbering can be used to compute SCCs



Summary

- BFS and DFS can be used to identify connected components in an undirected graph
 - BFS and DFS identify an underlying tree, non-tree edges generate cycles

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
 - BFS and DFS identify an underlying tree, non-tree edges generate cycles
- In a directed graph, non-tree edges can be forward / back / cross
 - Only back edges generate cycles
 - Classify non-tree edges using DFS numbering

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
 - BFS and DFS identify an underlying tree, non-tree edges generate cycles
- In a directed graph, non-tree edges can be forward / back / cross
 - Only back edges generate cycles
 - Classify non-tree edges using DFS numbering
- Directed graphs decompose into strongly connected components
 - DFS numbering can be used to compute SCC decomposition

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
 - BFS and DFS identify an underlying tree, non-tree edges generate cycles
- In a directed graph, non-tree edges can be forward / back / cross
 - Only back edges generate cycles
 - Classify non-tree edges using DFS numbering
- Directed graphs decompose into strongly connected components
 - DFS numbering can be used to compute SCC decomposition
- DFS numbering can also be used to identify other features such as articulation points (cut vertices) and bridges (cut edges)

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
 - BFS and DFS identify an underlying tree, non-tree edges generate cycles
- In a directed graph, non-tree edges can be forward / back / cross
 - Only back edges generate cycles
 - Classify non-tree edges using DFS numbering
- Directed graphs decompose into strongly connected components
 - DFS numbering can be used to compute SCC decomposition
- DFS numbering can also be used to identify other features such as articulation points (cut vertices) and bridges (cut edges)
- Directed acyclic graphs are useful for representing dependencies
 - Given course prerequisites, find a valid sequence to complete a programme

Directed Acyclic Graphs (DAGs)

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 4

Tasks and dependencies

- Startup moving into new office space
- Major tasks for completing the interiors
 - Lay floor tiles
 - Plaster the walls
 - Paint the walls
 - Lay conduits (pipes) for electrical wires
 - Do electrical wiring
 - Install electrical fittings
 - Lay telecom conduits
 - Do phone and network cabling

Tasks and dependencies

- Startup moving into new office space
- Major tasks for completing the interiors
 - Lay floor tiles
 - Plaster the walls
 - Paint the walls
 - Lay conduits (pipes) for electrical wires
 - Do electrical wiring
 - Install electrical fittings
 - Lay telecom conduits
 - Do phone and network cabling
- Constraints on the sequence
 - Lay conduits before tiles and plastering
 - Lay tiles, plaster wall before painting
 - Finish painting before any cabling/wiring work
 - Electrical wiring before installing fittings

Tasks and dependencies

- Startup moving into new office space
- Major tasks for completing the interiors
 - Lay floor tiles
 - Plaster the walls
 - Paint the walls
 - Lay conduits (pipes) for electrical wires
 - Do electrical wiring
 - Install electrical fittings
 - Lay telecom conduits
 - Do phone and network cabling
- Constraints on the sequence
 - Lay conduits before tiles and plastering
 - Lay tiles, plaster wall before painting
 - Finish painting before any cabling/wiring work
 - Electrical wiring before installing fittings
- Represent constraints as a directed graph
 - Vertices are tasks
 - Edge (t, u) if task t has to be completed before task u

Tasks and dependencies

- Constraints on the sequence
 - Lay conduits before tiles and plastering
 - Lay tiles, plaster wall before painting
 - Finish painting before any cabling/wiring work
 - Electrical wiring before installing fittings
- Represent constraints as a directed graph
 - Vertices are tasks
 - Edge (t, u) if task t has to be completed before task u

Tasks and dependencies

- Constraints on the sequence

- Lay conduits before tiles and plastering
- Lay tiles, plaster wall before painting
- Finish painting before any cabling/wiring work
- Electrical wiring before installing fittings

Conduits (E)

Conduits (T)

Tiling

Plastering

Painting

- Represent constraints as a directed graph

- Vertices are tasks
- Edge (t, u) if task t has to be completed before task u

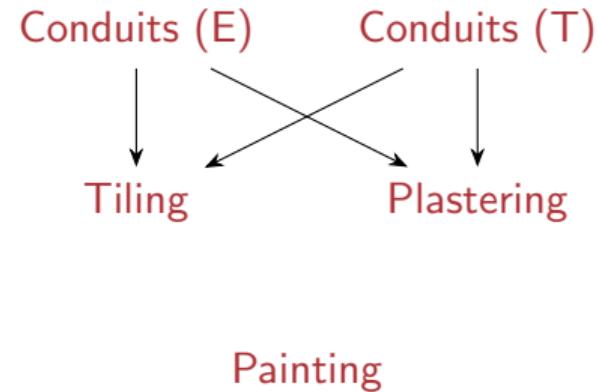
Wiring (E)

Cabling (T)

Fittings (E)

Tasks and dependencies

- Constraints on the sequence
 - Lay conduits before tiles and plastering
 - Lay tiles, plaster wall before painting
 - Finish painting before any cabling/wiring work
 - Electrical wiring before installing fittings



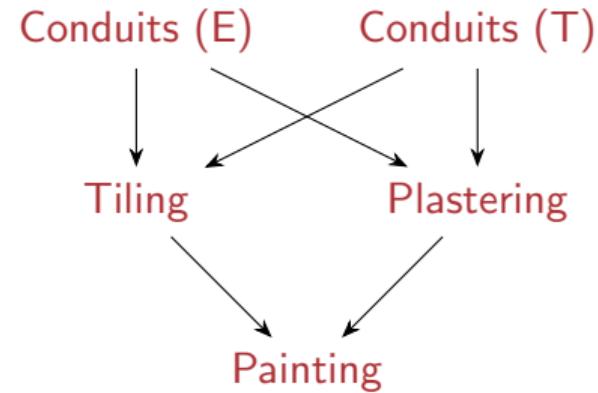
- Represent constraints as a directed graph



- Vertices are tasks
- Edge (t, u) if task t has to be completed before task u

Tasks and dependencies

- Constraints on the sequence
 - Lay conduits before tiles and plastering
 - Lay tiles, plaster wall before painting
 - Finish painting before any cabling/wiring work
 - Electrical wiring before installing fittings



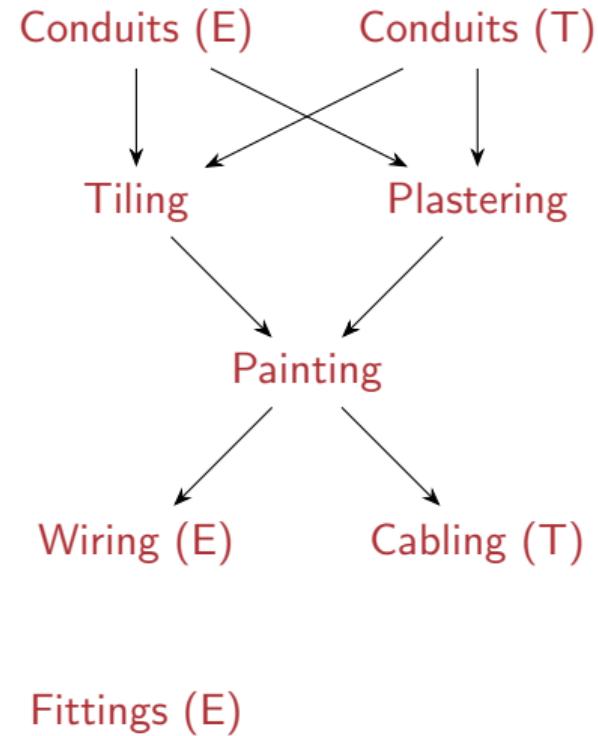
- Represent constraints as a directed graph

- Vertices are tasks
- Edge (t, u) if task t has to be completed before task u



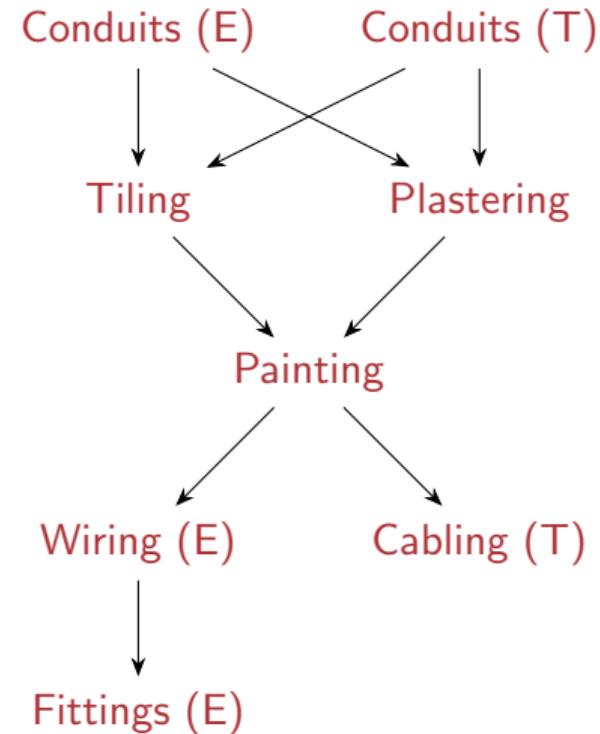
Tasks and dependencies

- Constraints on the sequence
 - Lay conduits before tiles and plastering
 - Lay tiles, plaster wall before painting
 - Finish painting before any cabling/wiring work
 - Electrical wiring before installing fittings
- Represent constraints as a directed graph
 - Vertices are tasks
 - Edge (t, u) if task t has to be completed before task u



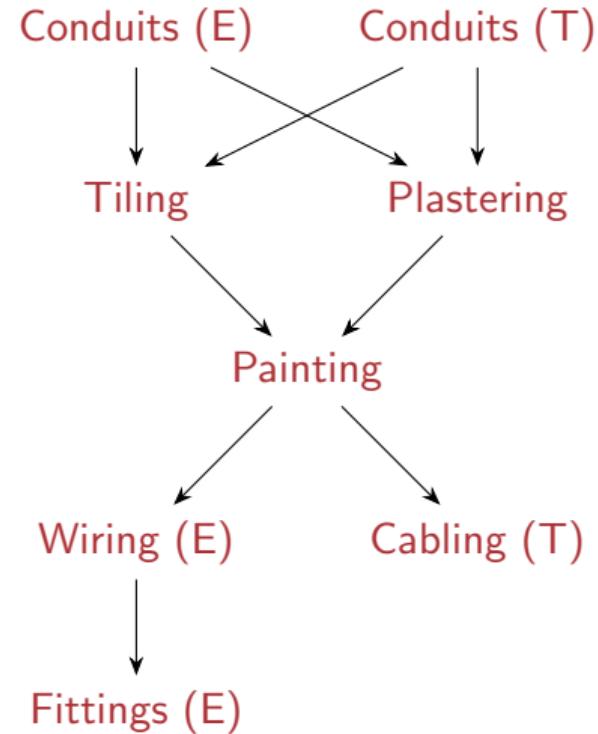
Tasks and dependencies

- Constraints on the sequence
 - Lay conduits before tiles and plastering
 - Lay tiles, plaster wall before painting
 - Finish painting before any cabling/wiring work
 - Electrical wiring before installing fittings
- Represent constraints as a directed graph
 - Vertices are tasks
 - Edge (t, u) if task t has to be completed before task u



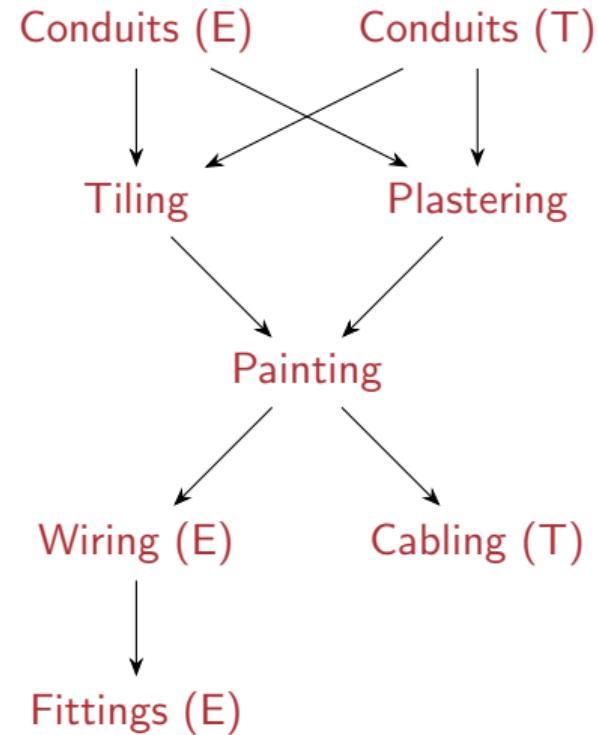
Typical questions

- Schedule the tasks respecting the dependencies



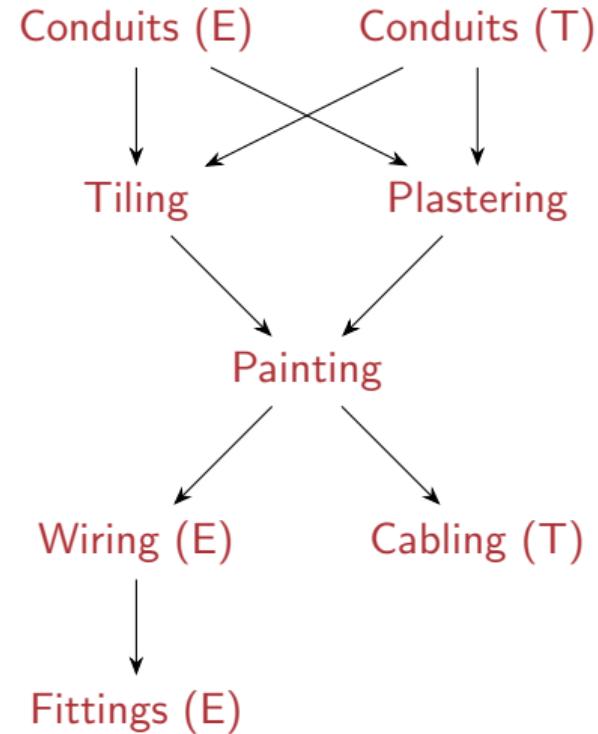
Typical questions

- Schedule the tasks respecting the dependencies
 - Conduits (E) – Conduits (T) – Tiling – Plastering – Painting – Wiring (E) – Cabling (T) – Fittings (E)



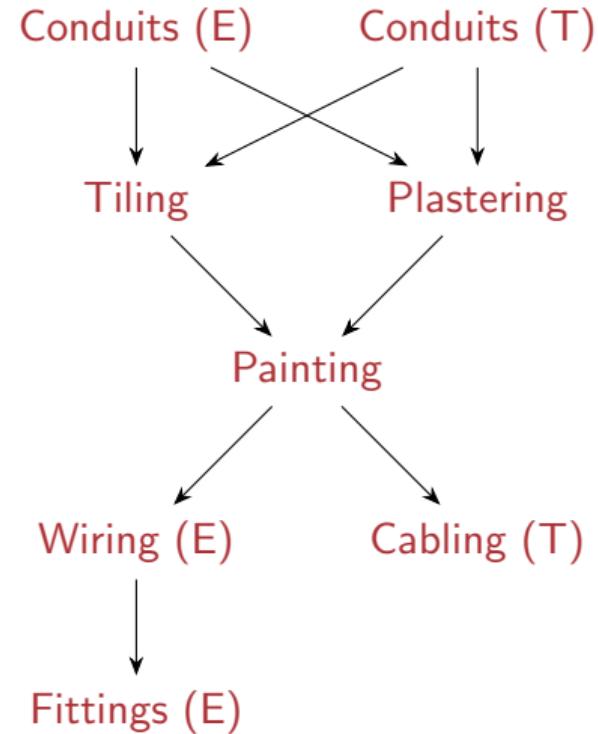
Typical questions

- Schedule the tasks respecting the dependencies
 - Conduits (E) – Conduits (T) – Tiling – Plastering – Painting – Wiring (E) – Cabling (T) – Fittings (E)
 - Conduits (T) – Conduits (E) – Plastering – Tiling – Painting – Wiring (E) – Fittings (E) – Cabling (T)
 - ...



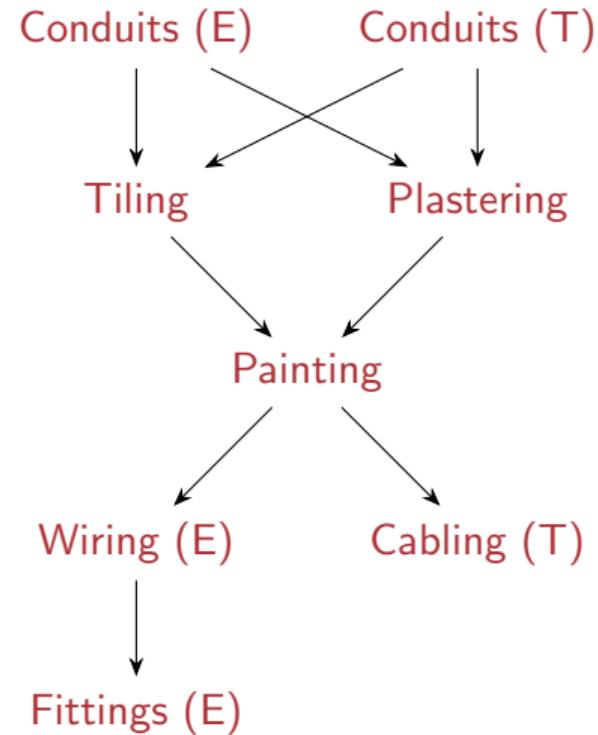
Typical questions

- Schedule the tasks respecting the dependencies
 - Conduits (E) – Conduits (T) – Tiling – Plastering – Painting – Wiring (E) – Cabling (T) – Fittings (E)
 - Conduits (T) – Conduits (E) – Plastering – Tiling – Painting – Wiring (E) – Fittings (E) – Cabling (T)
 - ...
- How long will the work take?



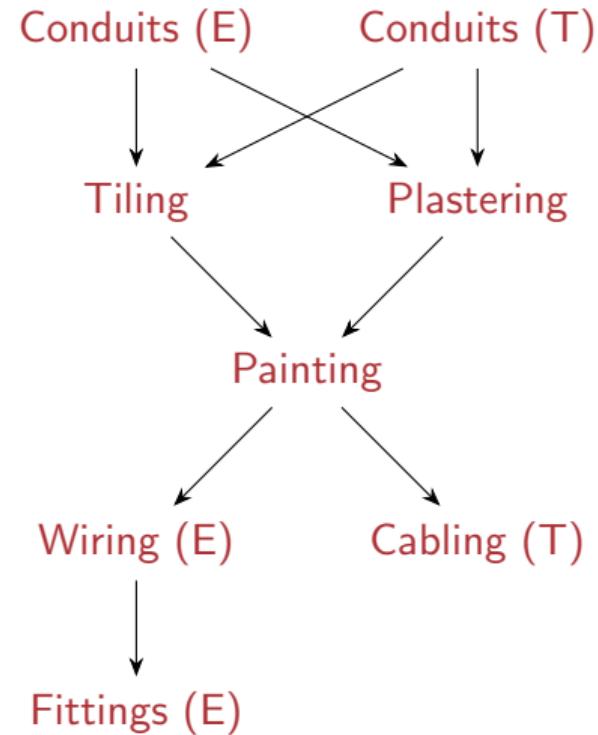
Directed Acyclic Graphs

- Formally, we have a **directed acyclic graph (DAG)**
- $G = (V, E)$, a directed graph without directed cycles



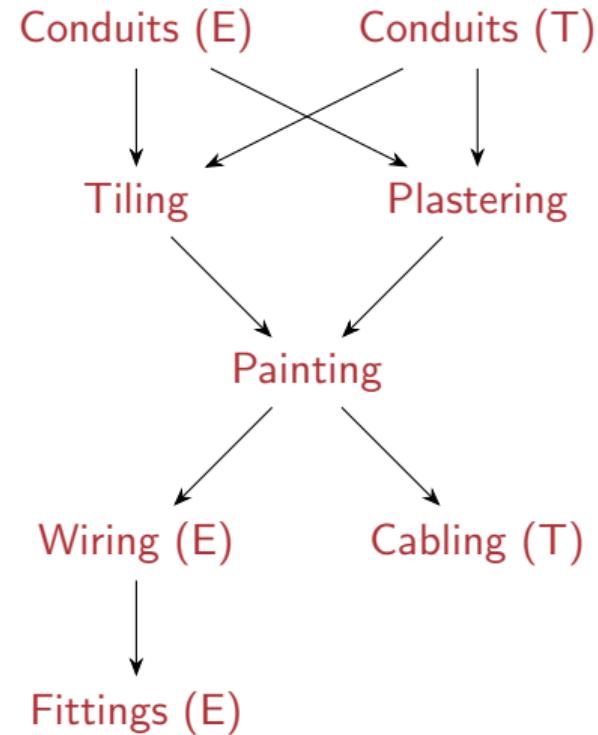
Directed Acyclic Graphs

- Formally, we have a **directed acyclic graph (DAG)**
- $G = (V, E)$, a directed graph without directed cycles
- Find a schedule
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j



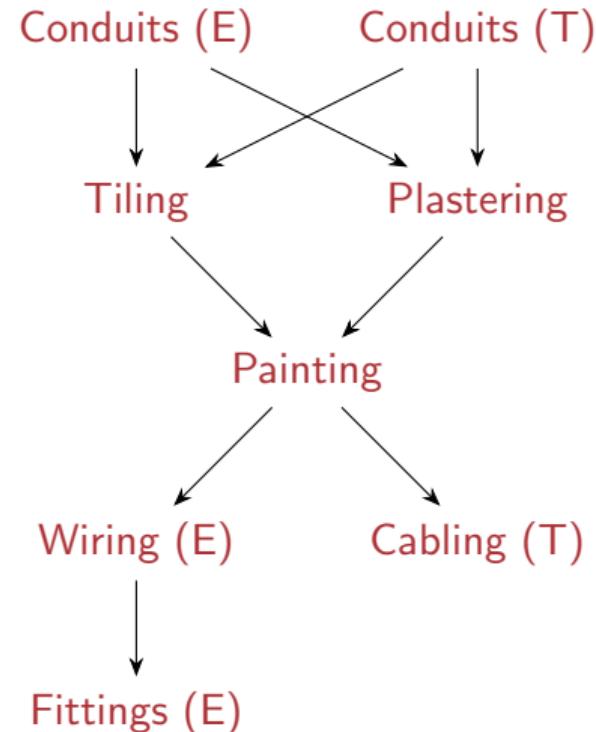
Directed Acyclic Graphs

- Formally, we have a **directed acyclic graph (DAG)**
- $G = (V, E)$, a directed graph without directed cycles
- Find a schedule
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
 - Topological sorting



Directed Acyclic Graphs

- Formally, we have a **directed acyclic graph (DAG)**
- $G = (V, E)$, a directed graph without directed cycles
- Find a schedule
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
 - Topological sorting
- How long will the work take?
 - Find the longest path in the DAG



Summary

- Directed acyclic graphs are a natural way to represent dependencies
- Arise in many contexts
 - Pre-requisites between courses for completing a degree
 - Recipe for cooking
 - Construction projects
 - ...
- Problems to be solved on DAGS
 - Topological sorting
 - Longest paths

Topological Sorting

Madhavan Mukund

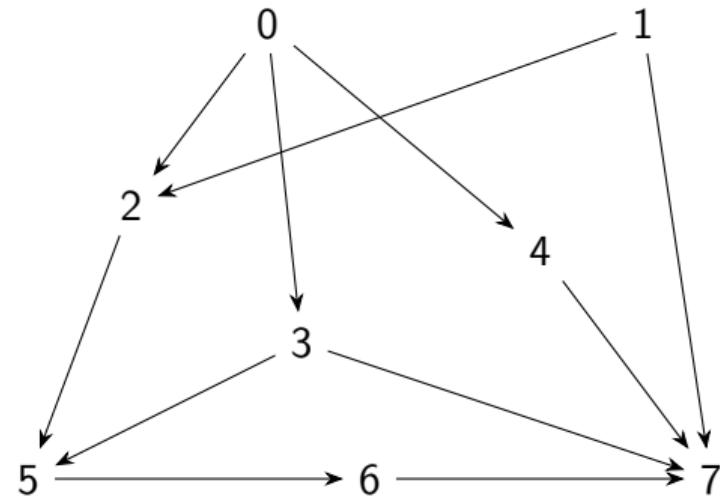
<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 4

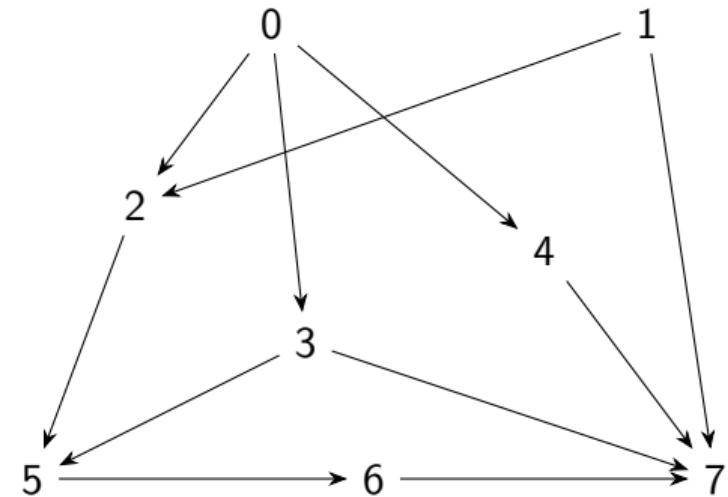
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
- Represents a feasible schedule



Topological Sort

- A graph with directed cycles cannot be sorted topologically
- Path $i \rightsquigarrow j$ means i must be listed before j
- Cycle \Rightarrow vertices i, j such that there are paths $i \rightsquigarrow j$ and $j \rightsquigarrow i$
- i must appear before j , and j must appear before i , impossible!

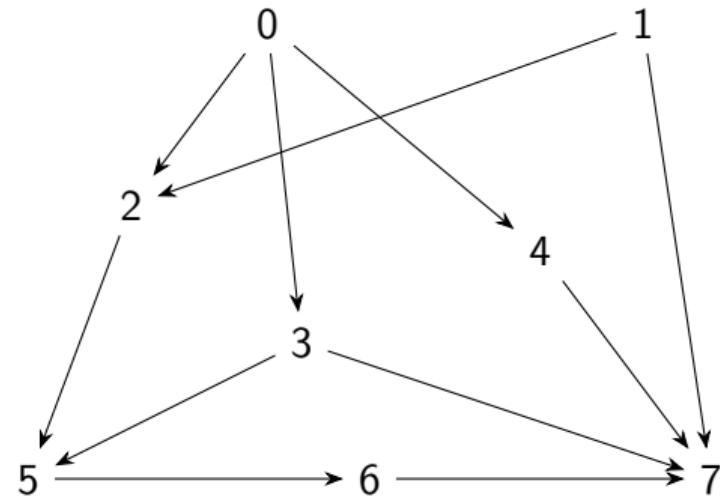


Topological Sort

- A graph with directed cycles cannot be sorted topologically
- Path $i \rightsquigarrow j$ means i must be listed before j
- Cycle \Rightarrow vertices i, j such that there are paths $i \rightsquigarrow j$ and $j \rightsquigarrow i$
- i must appear before j , and j must appear before i , impossible!

Claim

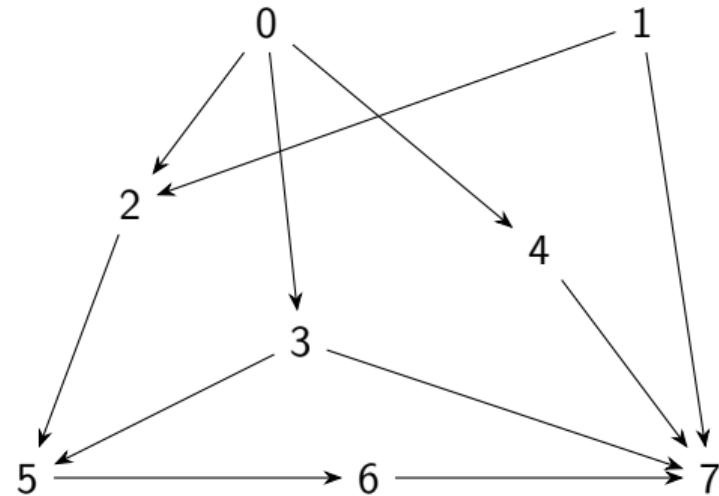
Every DAG can be topologically sorted



How to topologically sort a DAG?

Strategy

- First list vertices with no dependencies
- As we proceed, list vertices whose dependencies have already been listed
- ...



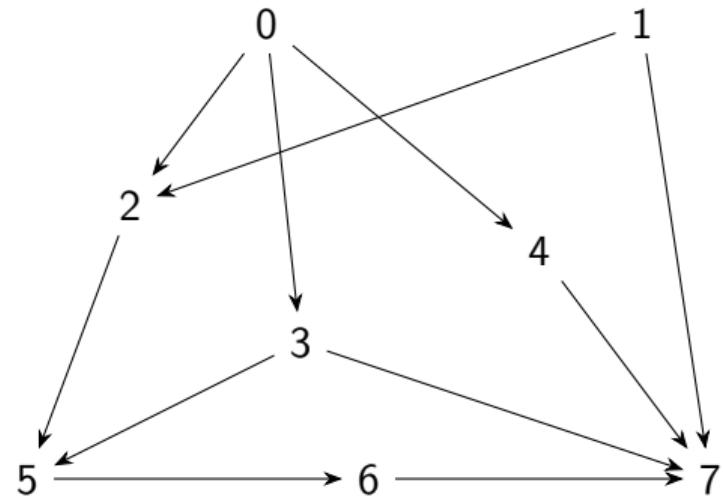
How to topologically sort a DAG?

Strategy

- First list vertices with no dependencies
- As we proceed, list vertices whose dependencies have already been listed
- ...

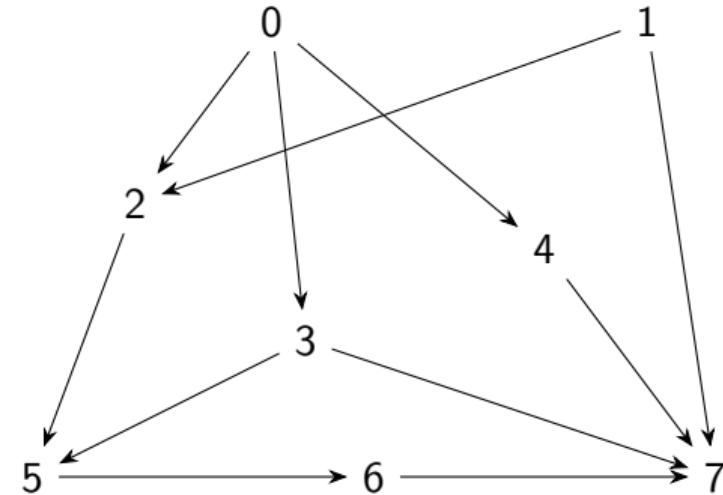
Questions

- Why will there be a starting vertex with no dependencies?
- How do we guarantee we can keep progressing with the listing?



Algorithm for topological sort

- A vertex with no dependencies has no incoming edges, $\text{indegree}(v) = 0$

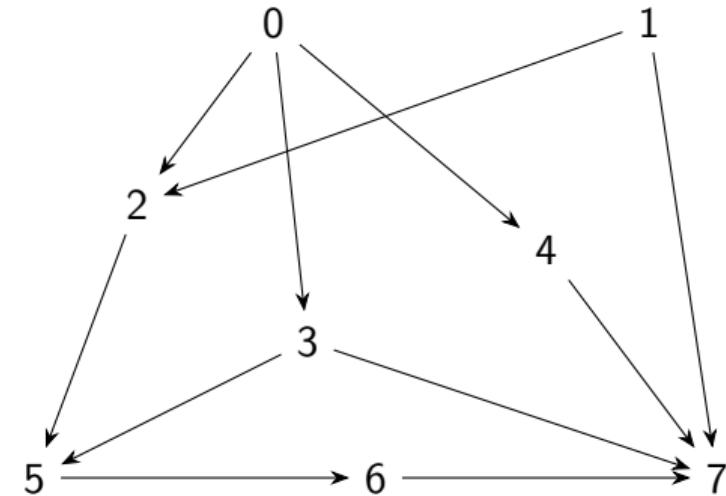


Algorithm for topological sort

- A vertex with no dependencies has no incoming edges, $\text{indegree}(v) = 0$

Claim

Every DAG has a vertex with indegree 0



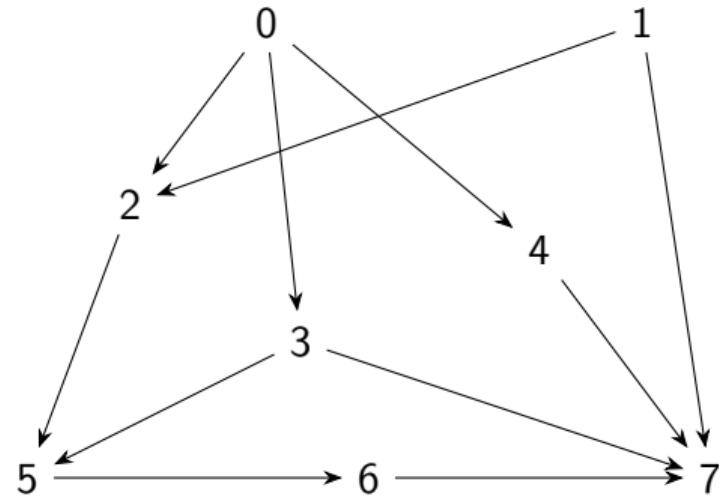
Algorithm for topological sort

- A vertex with no dependencies has no incoming edges, $\text{indegree}(v) = 0$

Claim

Every DAG has a vertex with indegree 0

- Start with any vertex with $\text{indegree} > 0$
- Follow edge back to one of its predecessors
- Repeat so long as $\text{indegree} > 0$
- If we repeat n times, we must have a cycle, which is impossible in a DAG

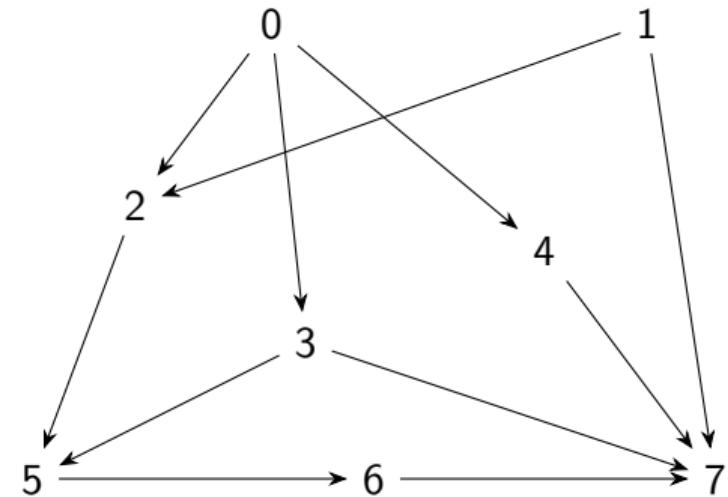


Topological sort algorithm

Fact

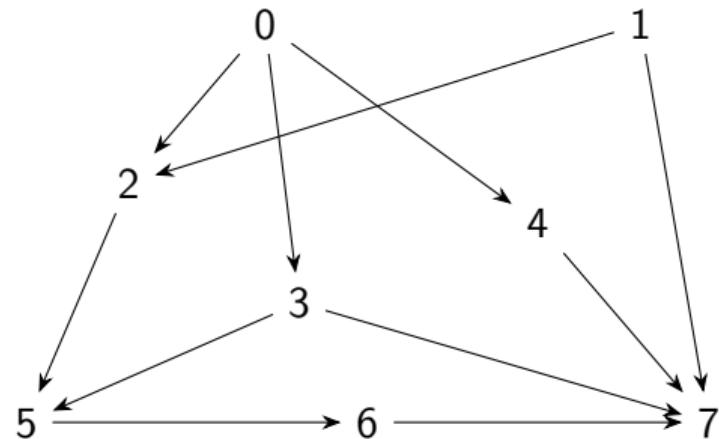
Every DAG has a vertex with indegree 0

- List out a vertex j with indegree = 0
- Delete j and all edges from j
- What remains is again a DAG!
- Can find another vertex with
indegree = 0 to list and eliminate
- Repeat till all vertices are listed



Topological sort algorithm

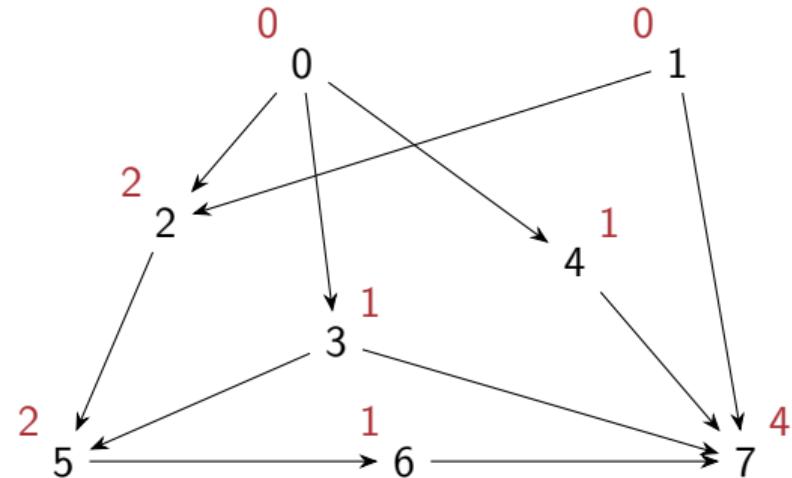
- Compute **indegree** of each vertex



Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix

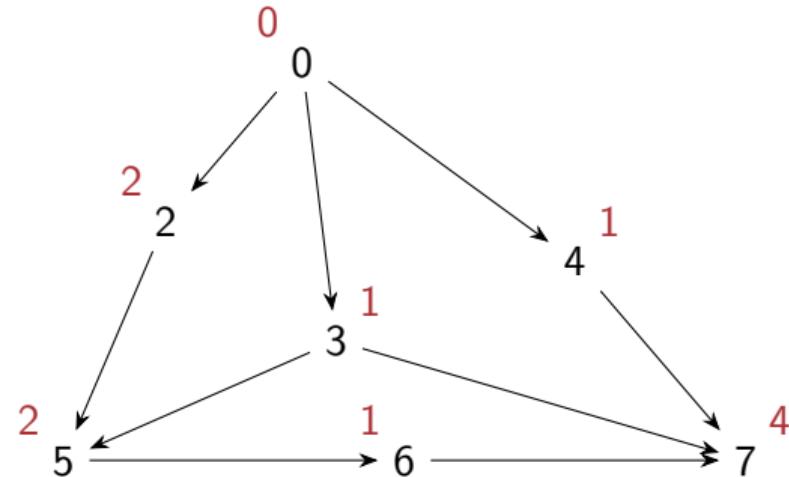
Indegree



Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG

Indegree



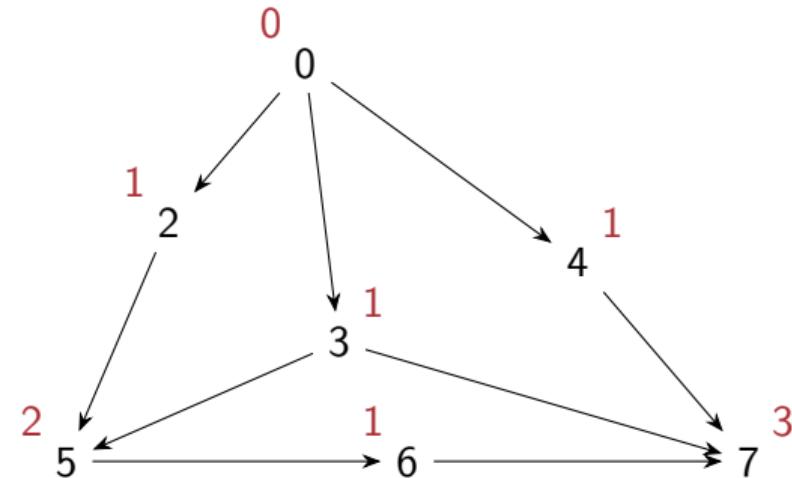
Topologically sorted sequence

1,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees

Indegree



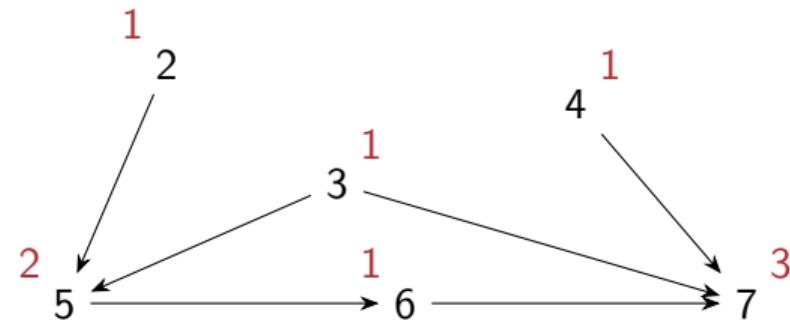
Topologically sorted sequence

1,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate

Indegree



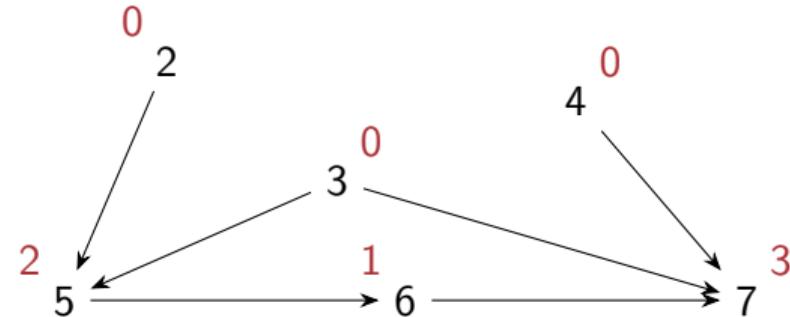
Topologically sorted sequence

1, 0,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate

Indegree



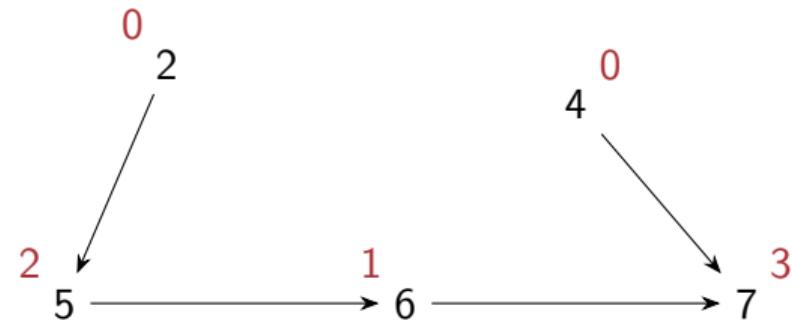
Topologically sorted sequence

1, 0,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree



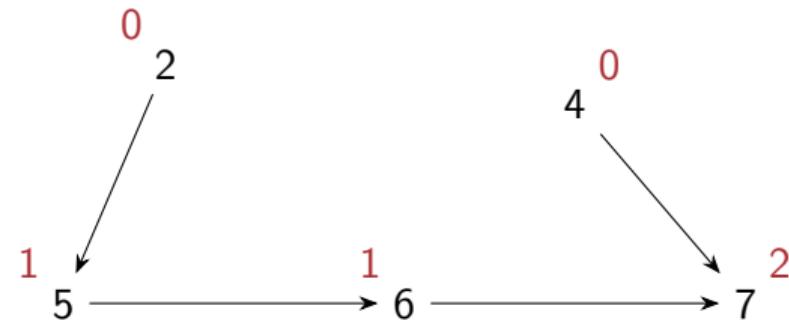
Topologically sorted sequence

1, 0, 3,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree



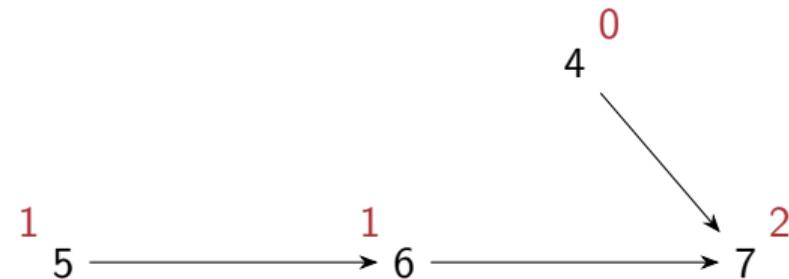
Topologically sorted sequence

1, 0, 3,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree

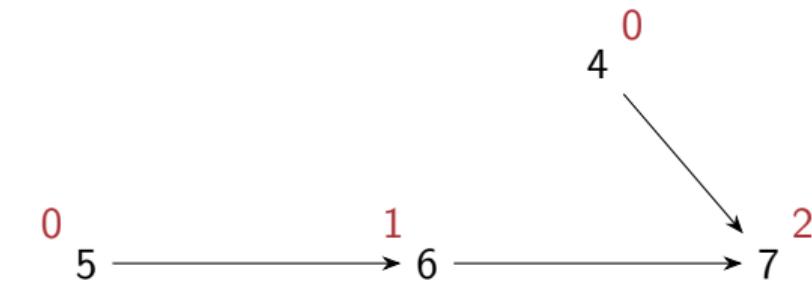


Topologically sorted sequence

1, 0, 3, 2,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed



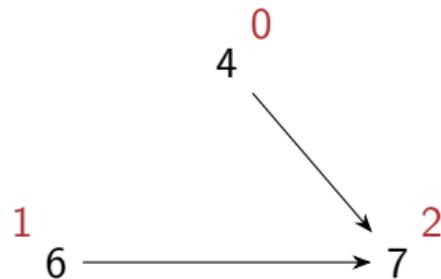
Topologically sorted sequence

1, 0, 3, 2,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree



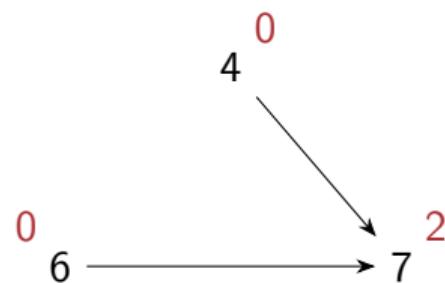
Topologically sorted sequence

1, 0, 3, 2, 5,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree



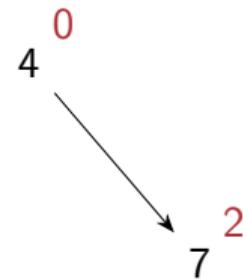
Topologically sorted sequence

1, 0, 3, 2, 5,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree



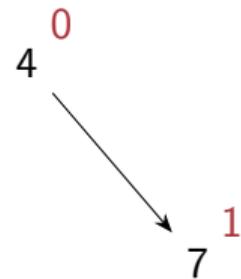
Topologically sorted sequence

1, 0, 3, 2, 5, 6,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree



Topologically sorted sequence

1, 0, 3, 2, 5, 6,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree

1

7

Topologically sorted sequence

1, 0, 3, 2, 5, 6, 4,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree

0

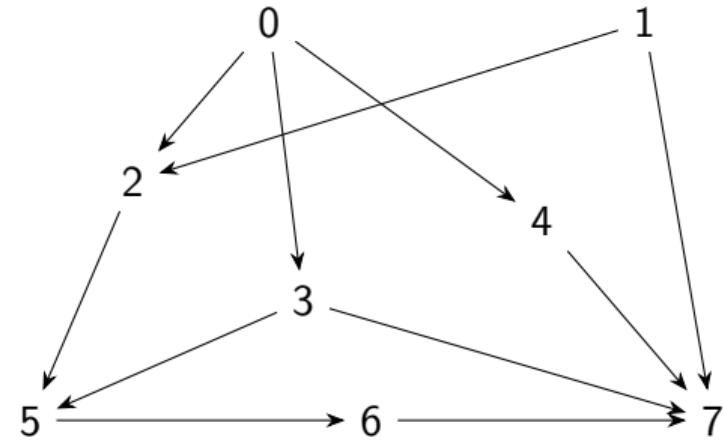
7

Topologically sorted sequence

1, 0, 3, 2, 5, 6, 4,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with **indegree 0** and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed



Topologically sorted sequence

1, 0, 3, 2, 5, 6, 4, 7

An implementation of topological sort

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Repeat till all vertices are listed

```
def toposort(AMat):  
    (rows,cols) = AMat.shape  
    indegree = {}  
    toposortlist = []  
  
    for c in range(cols):  
        indegree[c] = 0  
        for r in range(rows):  
            if AMat[r,c] == 1:  
                indegree[c] = indegree[c] + 1  
  
    for i in range(rows):  
        j = min([k for k in range(cols)  
                 if indegree[k] == 0])  
        toposortlist.append(j)  
        indegree[j] = indegree[j]-1  
        for k in range(cols):  
            if AMat[j,k] == 1:  
                indegree[k] = indegree[k] - 1  
  
    return(toposortlist)
```

An implementation of topological sort

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Repeat till all vertices are listed

Analysis

```
def toposort(AMat):  
    (rows,cols) = AMat.shape  
    indegree = {}  
    toposortlist = []  
  
    for c in range(cols):  
        indegree[c] = 0  
        for r in range(rows):  
            if AMat[r,c] == 1:  
                indegree[c] = indegree[c] + 1  
  
    for i in range(rows):  
        j = min([k for k in range(cols)  
                 if indegree[k] == 0])  
        toposortlist.append(j)  
        indegree[j] = indegree[j]-1  
        for k in range(cols):  
            if AMat[j,k] == 1:  
                indegree[k] = indegree[k] - 1  
  
    return(toposortlist)
```

An implementation of topological sort

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Repeat till all vertices are listed

Analysis

- Initializing indegrees is $O(n^2)$

```
def toposort(AMat):  
    (rows,cols) = AMat.shape  
    indegree = {}  
    toposortlist = []  
  
    for c in range(cols):  
        indegree[c] = 0  
        for r in range(rows):  
            if AMat[r,c] == 1:  
                indegree[c] = indegree[c] + 1  
  
    for i in range(rows):  
        j = min([k for k in range(cols)  
                 if indegree[k] == 0])  
        toposortlist.append(j)  
        indegree[j] = indegree[j]-1  
        for k in range(cols):  
            if AMat[j,k] == 1:  
                indegree[k] = indegree[k] - 1  
  
    return(toposortlist)
```

An implementation of topological sort

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Repeat till all vertices are listed

Analysis

- Initializing indegrees is $O(n^2)$
- Loop to enumerate vertices runs n times
 - Identify next vertex to enumerate: $O(n)$
 - Updating indegrees: $O(n)$

```
def toposort(AMat):  
    (rows,cols) = AMat.shape  
    indegree = {}  
    toposortlist = []  
  
    for c in range(cols):  
        indegree[c] = 0  
    for r in range(rows):  
        if AMat[r,c] == 1:  
            indegree[c] = indegree[c] + 1  
  
    for i in range(rows):  
        j = min([k for k in range(cols)  
                 if indegree[k] == 0])  
        toposortlist.append(j)  
        indegree[j] = indegree[j]-1  
        for k in range(cols):  
            if AMat[j,k] == 1:  
                indegree[k] = indegree[k] - 1  
  
    return(toposortlist)
```

An implementation of topological sort

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Repeat till all vertices are listed

Analysis

- Initializing indegrees is $O(n^2)$
- Loop to enumerate vertices runs n times
 - Identify next vertex to enumerate: $O(n)$
 - Updating indegrees: $O(n)$
- Overall, $O(n^2)$

```
def toposort(AMat):  
    (rows,cols) = AMat.shape  
    indegree = {}  
    toposortlist = []  
  
    for c in range(cols):  
        indegree[c] = 0  
    for r in range(rows):  
        if AMat[r,c] == 1:  
            indegree[c] = indegree[c] + 1  
  
    for i in range(rows):  
        j = min([k for k in range(cols)  
                 if indegree[k] == 0])  
        toposortlist.append(j)  
        indegree[j] = indegree[j]-1  
        for k in range(cols):  
            if AMat[j,k] == 1:  
                indegree[k] = indegree[k] - 1  
  
    return(toposortlist)
```

Using adjacency lists

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Enumerate head of queue, update indegrees, add indegree 0 to queue
- Repeat till queue is empty

```
def toposortlist(AList):  
    (indegree,toposortlist) = ({},[])  
    for u in AList.keys():  
        indegree[u] = 0  
    for u in AList.keys():  
        for v in AList[u]:  
            indegree[v] = indegree[v] + 1  
  
    zerodegreeq = Queue()  
    for u in AList.keys():  
        if indegree[u] == 0:  
            zerodegreeq.addq(u)  
  
    while (not zerodegreeq.isempty()):  
        j = zerodegreeq.delq()  
        toposortlist.append(j)  
        indegree[j] = indegree[j]-1  
        for k in AList[j]:  
            indegree[k] = indegree[k] - 1  
            if indegree[k] == 0:  
                zerodegreeq.addq(k)  
    return(toposortlist)
```

Using adjacency lists

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Enumerate head of queue, update indegrees, add indegree 0 to queue
- Repeat till queue is empty

Analysis

```
def toposortlist(AList):  
    (indegree,toposortlist) = ({},[])  
    for u in AList.keys():  
        indegree[u] = 0  
    for u in AList.keys():  
        for v in AList[u]:  
            indegree[v] = indegree[v] + 1  
  
    zerodegreeq = Queue()  
    for u in AList.keys():  
        if indegree[u] == 0:  
            zerodegreeq.addq(u)  
  
    while (not zerodegreeq.isempty()):  
        j = zerodegreeq.delq()  
        toposortlist.append(j)  
        indegree[j] = indegree[j]-1  
        for k in AList[j]:  
            indegree[k] = indegree[k] - 1  
            if indegree[k] == 0:  
                zerodegreeq.addq(k)  
    return(toposortlist)
```

Using adjacency lists

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Enumerate head of queue, update indegrees, add indegree 0 to queue
- Repeat till queue is empty

Analysis

- Initializing indegrees is $O(m + n)$

```
def toposortlist(AList):
    (indegree,toposortlist) = ({},[])
    for u in AList.keys():
        indegree[u] = 0
    for u in AList.keys():
        for v in AList[u]:
            indegree[v] = indegree[v] + 1

    zerodegreeq = Queue()
    for u in AList.keys():
        if indegree[u] == 0:
            zerodegreeq.addq(u)

    while (not zerodegreeq.isempty()):
        j = zerodegreeq.delq()
        toposortlist.append(j)
        indegree[j] = indegree[j]-1
        for k in AList[j]:
            indegree[k] = indegree[k] - 1
            if indegree[k] == 0:
                zerodegreeq.addq(k)
    return(toposortlist)
```

Using adjacency lists

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Enumerate head of queue, update indegrees, add indegree 0 to queue
- Repeat till queue is empty

Analysis

- Initializing indegrees is $O(m + n)$
- Loop to enumerate vertices runs n times
 - Updating indegrees: amortised $O(m)$

```
def toposortlist(AList):  
    (indegree,toposortlist) = ({},[])  
    for u in AList.keys():  
        indegree[u] = 0  
    for u in AList.keys():  
        for v in AList[u]:  
            indegree[v] = indegree[v] + 1  
  
    zerodegreeq = Queue()  
    for u in AList.keys():  
        if indegree[u] == 0:  
            zerodegreeq.addq(u)  
  
    while (not zerodegreeq.isempty()):  
        j = zerodegreeq.delq()  
        toposortlist.append(j)  
        indegree[j] = indegree[j]-1  
        for k in AList[j]:  
            indegree[k] = indegree[k] - 1  
            if indegree[k] == 0:  
                zerodegreeq.addq(k)  
    return(toposortlist)
```

Using adjacency lists

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Enumerate head of queue, update indegrees, add indegree 0 to queue
- Repeat till queue is empty

Analysis

- Initializing indegrees is $O(m + n)$
- Loop to enumerate vertices runs n times
 - Updating indegrees: amortised $O(m)$
- Overall, $O(m + n)$

```
def toposortlist(AList):
    (indegree,toposortlist) = ({},[])
    for u in AList.keys():
        indegree[u] = 0
    for u in AList.keys():
        for v in AList[u]:
            indegree[v] = indegree[v] + 1

    zerodegreeq = Queue()
    for u in AList.keys():
        if indegree[u] == 0:
            zerodegreeq.addq(u)

    while (not zerodegreeq.isempty()):
        j = zerodegreeq.delq()
        toposortlist.append(j)
        indegree[j] = indegree[j]-1
        for k in AList[j]:
            indegree[k] = indegree[k] - 1
            if indegree[k] == 0:
                zerodegreeq.addq(k)
    return(toposortlist)
```

Summary

- Directed acyclic graphs are a natural way to represent dependencies
- Topological sort gives a feasible schedule that represents dependencies
 - At least one vertex with no dependencies, indegree 0
 - Eliminating such a vertex retains DAG structure
 - Repeat the process till all vertices are listed
- Complexity
 - Using adjacency matrix takes $O(n^2)$
 - Using adjacency list takes $O(m + n)$
- More than one topological sort is possible
 - Choice of which vertex with indegree 0 to list next

Longest Paths in DAGs

Madhavan Mukund

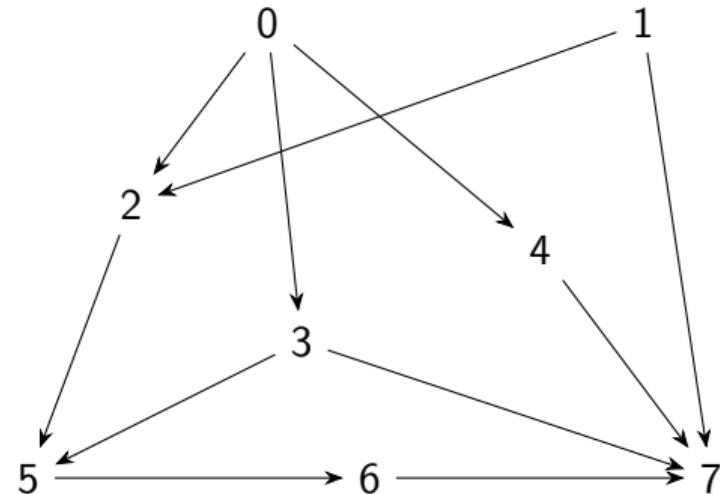
<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 4

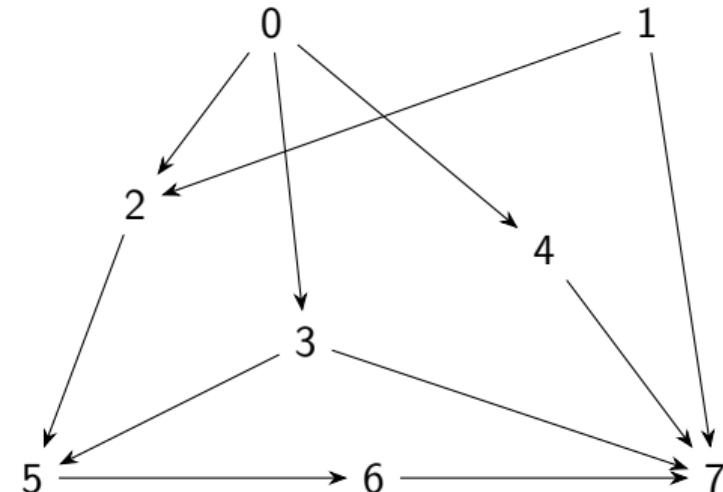
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles



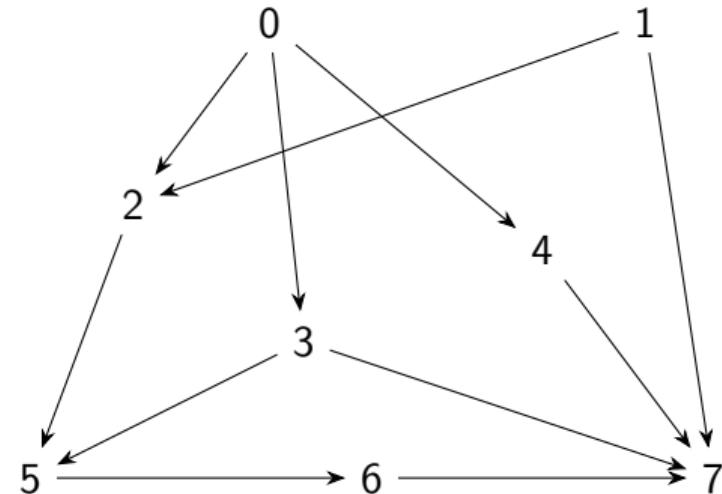
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
 - Feasible schedule



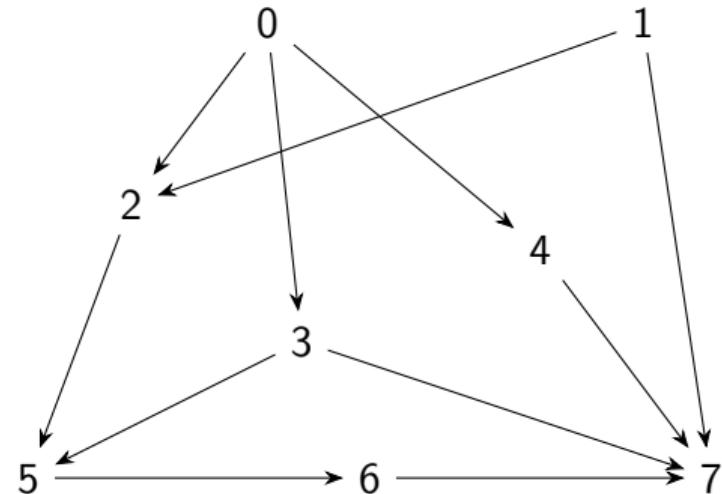
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
 - Feasible schedule
- Imagine the DAG represents prerequisites between courses



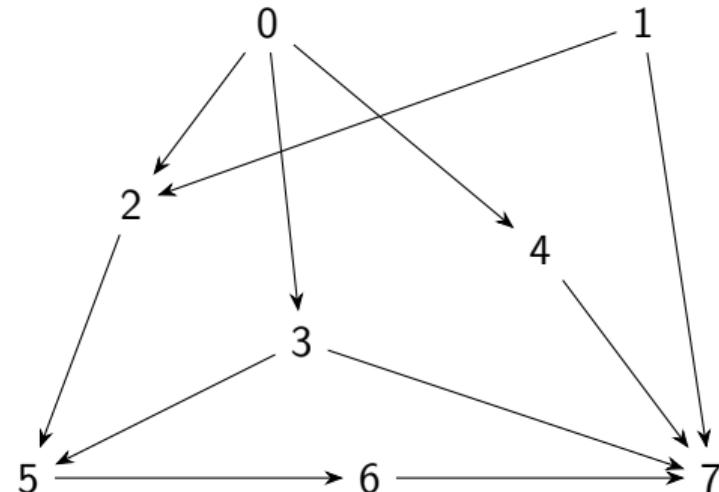
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
 - Feasible schedule
- Imagine the DAG represents prerequisites between courses
- Each course takes a semester



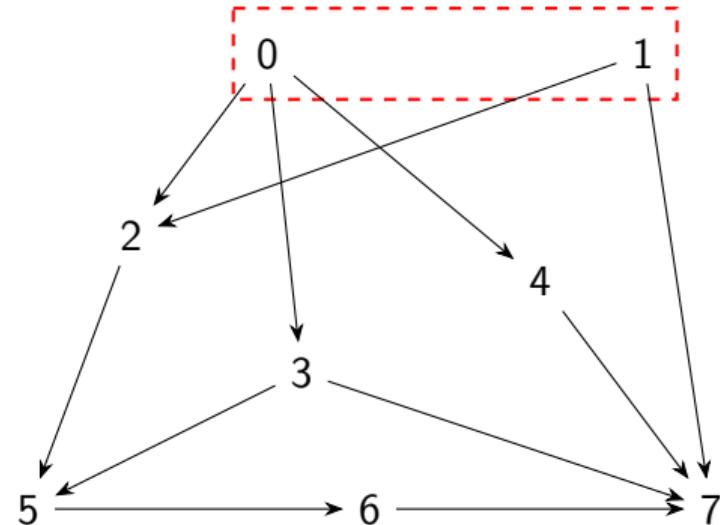
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
 - Feasible schedule
- Imagine the DAG represents prerequisites between courses
- Each course takes a semester
- Minimum number of semesters to complete the programme?



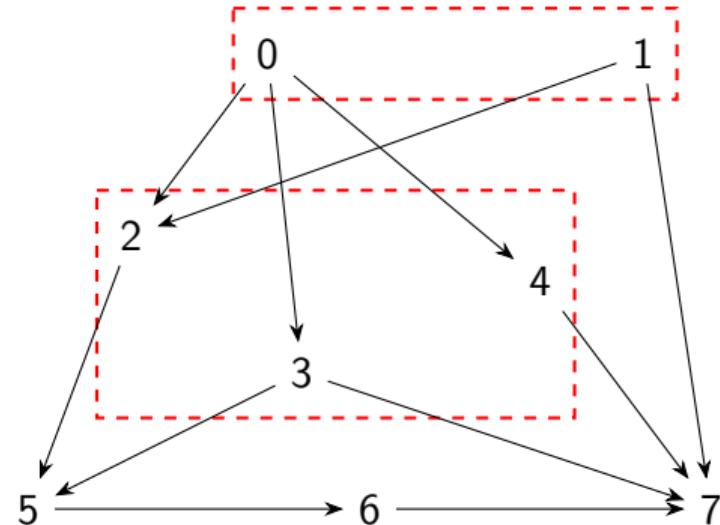
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
 - Feasible schedule
- Imagine the DAG represents prerequisites between courses
- Each course takes a semester
- Minimum number of semesters to complete the programme?



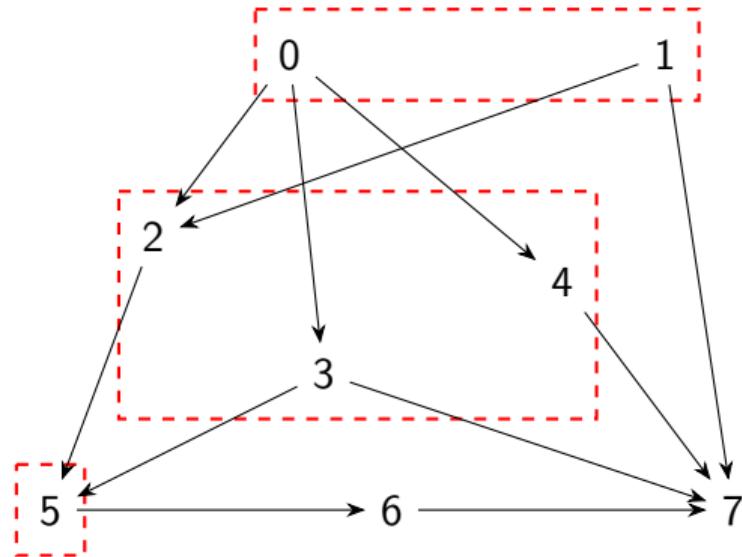
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
 - Feasible schedule
- Imagine the DAG represents prerequisites between courses
- Each course takes a semester
- Minimum number of semesters to complete the programme?



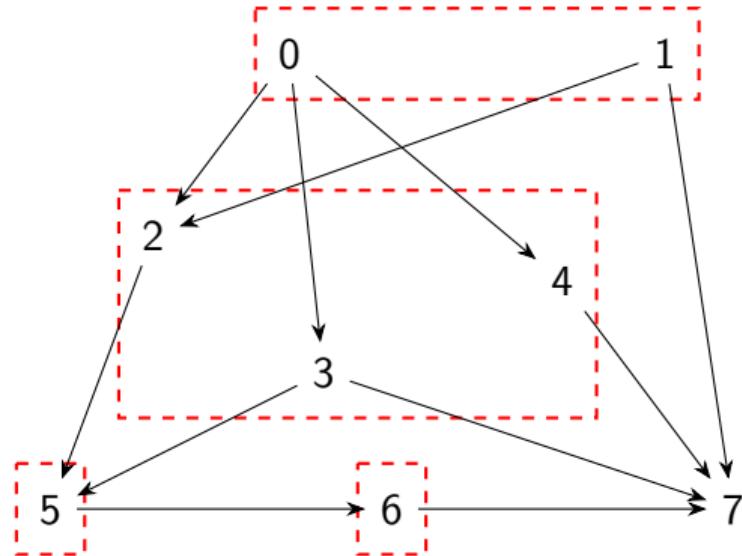
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
 - Feasible schedule
- Imagine the DAG represents prerequisites between courses
- Each course takes a semester
- Minimum number of semesters to complete the programme?



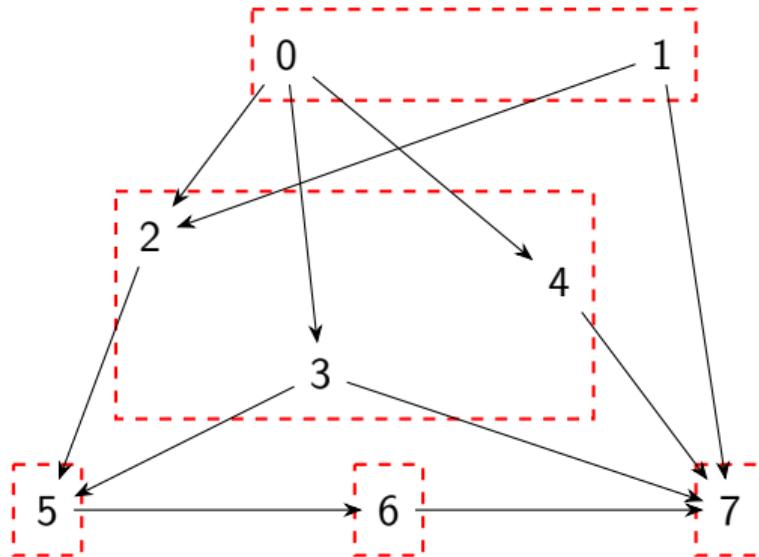
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
 - Feasible schedule
- Imagine the DAG represents prerequisites between courses
- Each course takes a semester
- Minimum number of semesters to complete the programme?



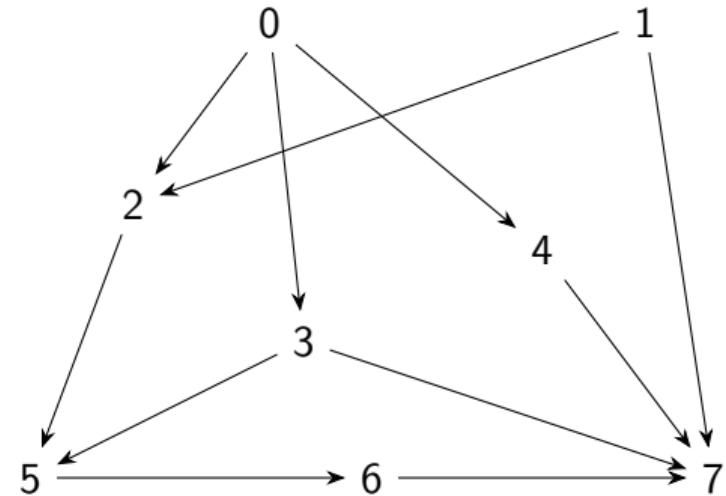
Directed Acyclic Graphs

- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
 - Enumerate $V = \{0, 1, \dots, n - 1\}$ such that for any $(i, j) \in E$, i appears before j
 - Feasible schedule
- Imagine the DAG represents prerequisites between courses
- Each course takes a semester
- Minimum number of semesters to complete the programme?



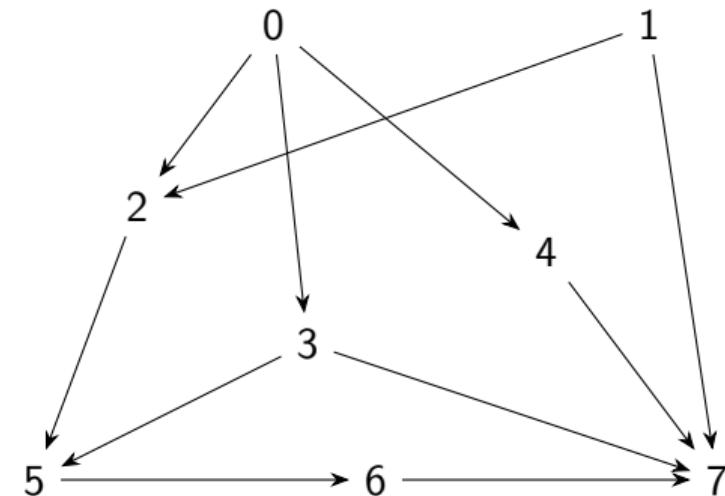
Longest Path

- Find the longest path in a DAG



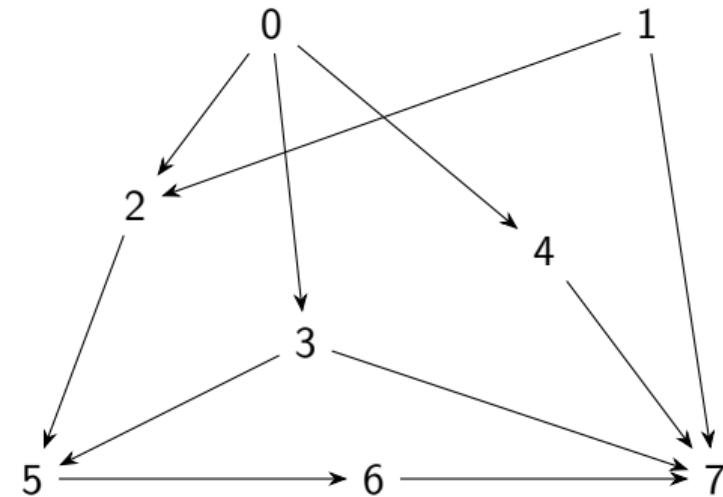
Longest Path

- Find the longest path in a DAG
- If $\text{indegree}(i) = 0$,
 $\text{longest-path-to}(i) = 0$



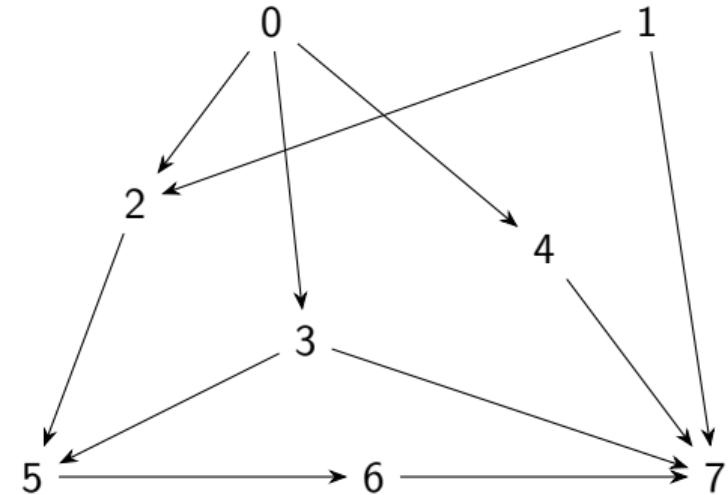
Longest Path

- Find the longest path in a DAG
- If $\text{indegree}(i) = 0$,
 $\text{longest-path-to}(i) = 0$
- If $\text{indegree}(i) > 0$, longest path to i is
1 more than longest path to its
incoming neighbours
$$\text{longest-path-to}(i) = 1 + \max\{\text{longest-path-to}(j) \mid (j, i) \in E\}$$



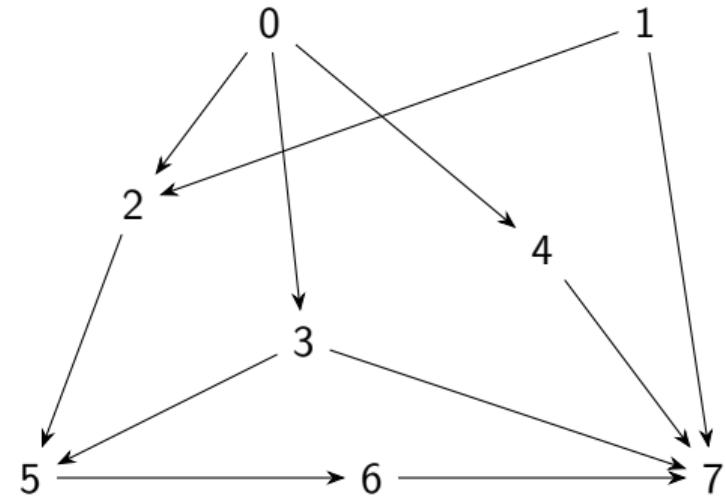
Longest Path

- $\text{longest-path-to}(i) = 1 + \max\{\text{longest-path-to}(j) \mid (j, i) \in E\}$



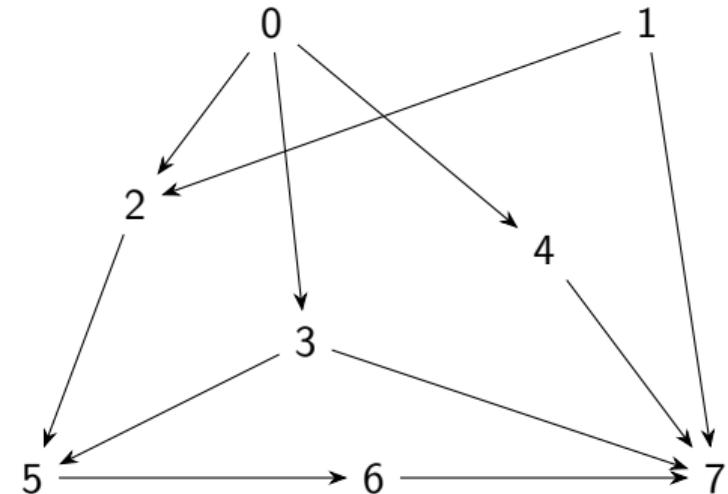
Longest Path

- $\text{longest-path-to}(i) = 1 + \max\{\text{longest-path-to}(j) \mid (j, i) \in E\}$
- To compute $\text{longest-path-to}(i)$, need $\text{longest-path-to}(k)$, for each incoming neighbour k



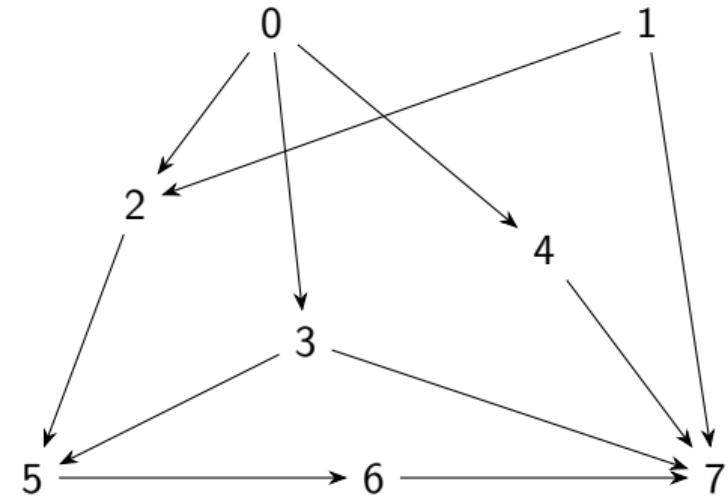
Longest Path

- $\text{longest-path-to}(i) = 1 + \max\{\text{longest-path-to}(j) \mid (j, i) \in E\}$
- To compute $\text{longest-path-to}(i)$, need $\text{longest-path-to}(k)$, for each incoming neighbour k
- If graph is topologically sorted, k is listed before i



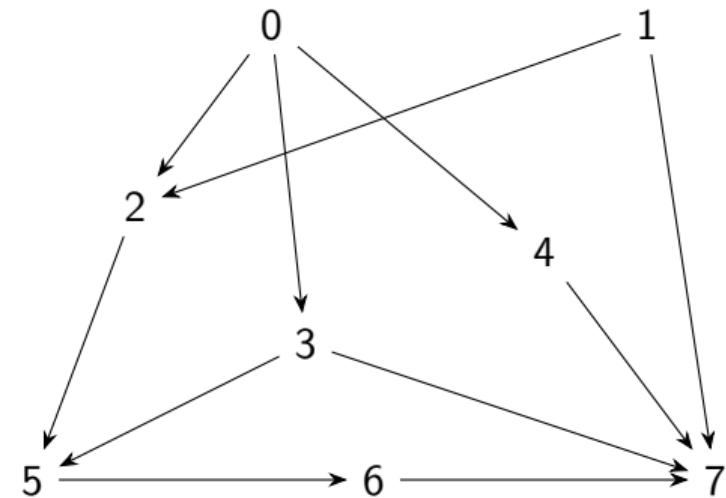
Longest Path

- $\text{longest-path-to}(i) = 1 + \max\{\text{longest-path-to}(j) \mid (j, i) \in E\}$
- To compute $\text{longest-path-to}(i)$, need $\text{longest-path-to}(k)$, for each incoming neighbour k
- If graph is topologically sorted, k is listed before i
- Hence compute $\text{longest-path-to}()$ in topological order



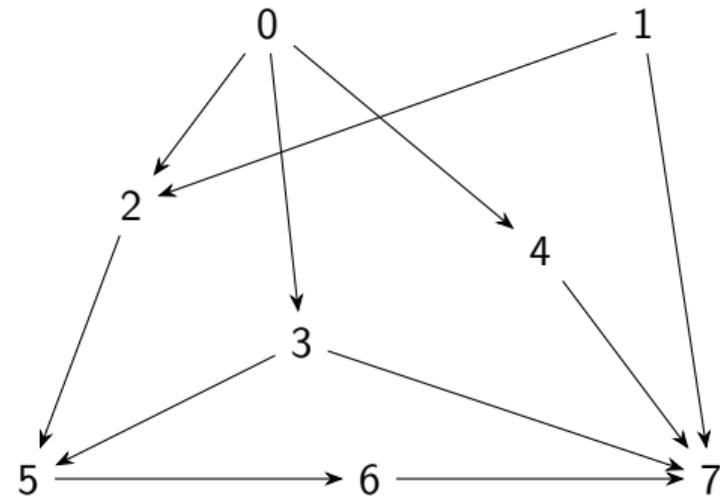
Longest Path

- Let i_0, i_1, \dots, i_{n-1} be a topological ordering of V



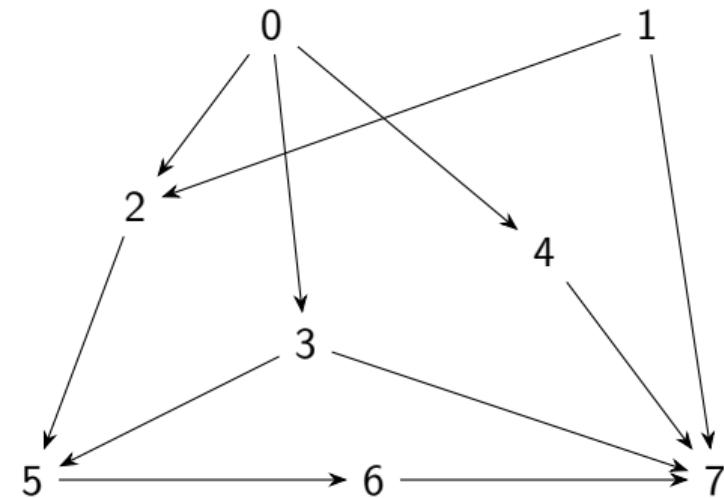
Longest Path

- Let i_0, i_1, \dots, i_{n-1} be a topological ordering of V
- All neighbours of i_k appear before it in this list



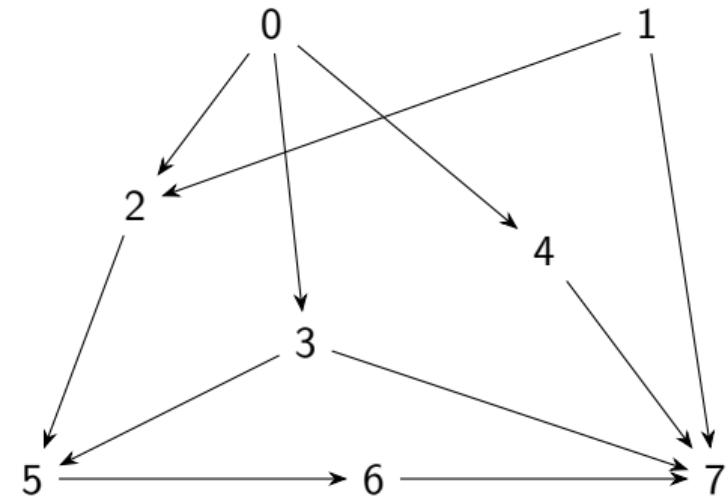
Longest Path

- Let i_0, i_1, \dots, i_{n-1} be a topological ordering of V
- All neighbours of i_k appear before it in this list
- From left to right, compute $\text{longest-path-to}(i_k)$ as $1 + \max\{\text{longest-path-to}(i_j) \mid (i_j, i_k) \in E\}$



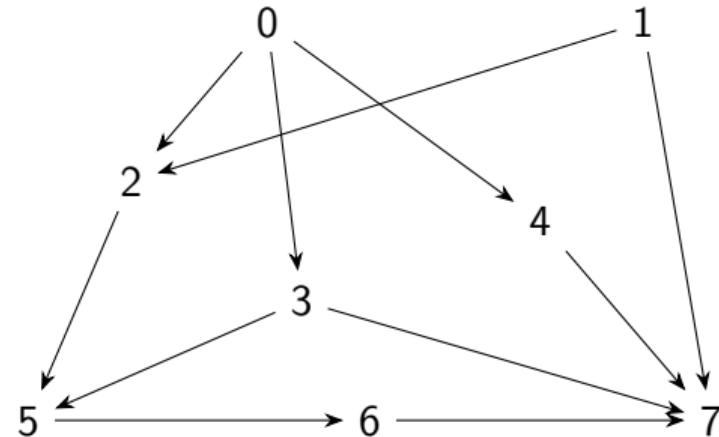
Longest Path

- Let i_0, i_1, \dots, i_{n-1} be a topological ordering of V
- All neighbours of i_k appear before it in this list
- From left to right, compute $\text{longest-path-to}(i_k)$ as $1 + \max\{\text{longest-path-to}(i_j) \mid (i_j, i_k) \in E\}$
- Overlap this computation with topological sorting



Longest path algorithm

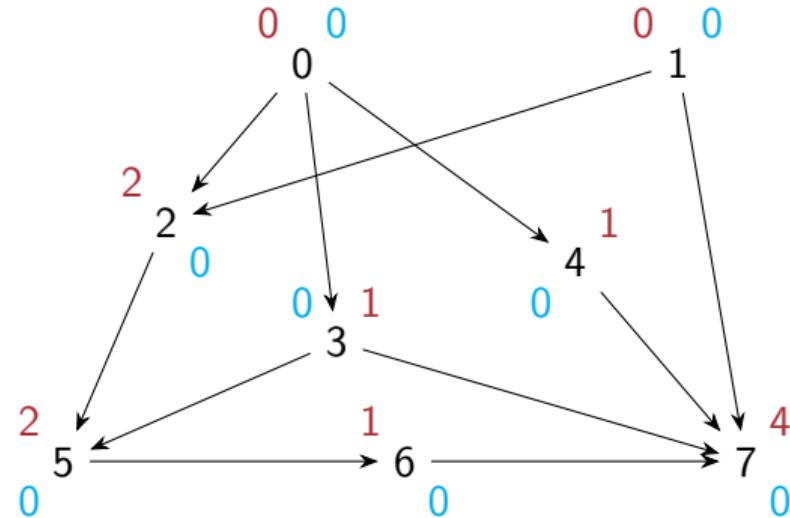
- Compute **indegree** of each vertex



Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest – path – to** to 0 for all vertices

Indegree, Longest path

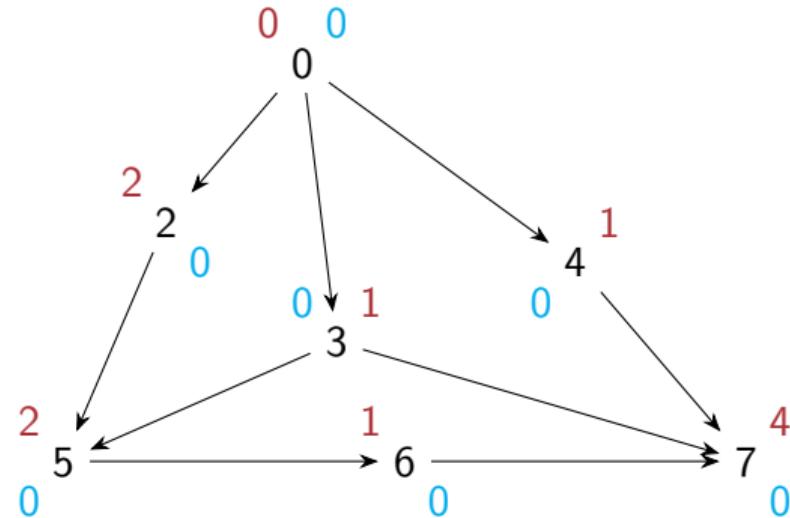


Topological order
Longest path to

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest-path-to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG

Indegree, Longest path



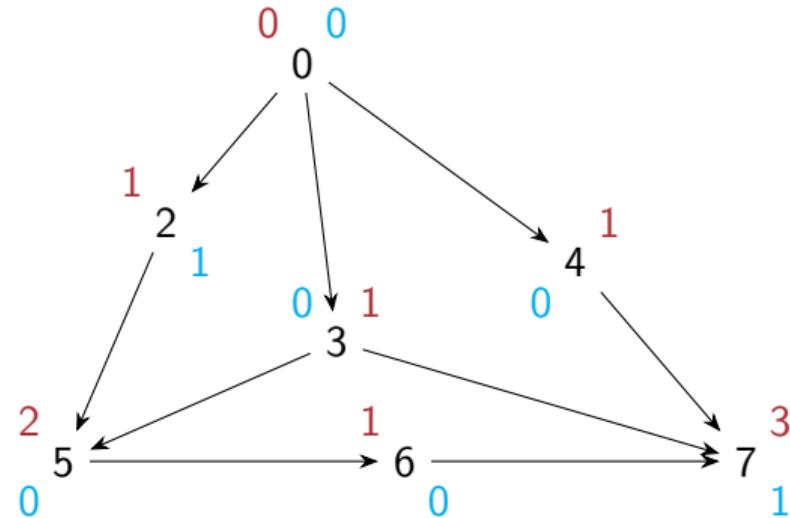
Topological order
Longest path to

1
0

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest-path-to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path

Indegree, Longest path

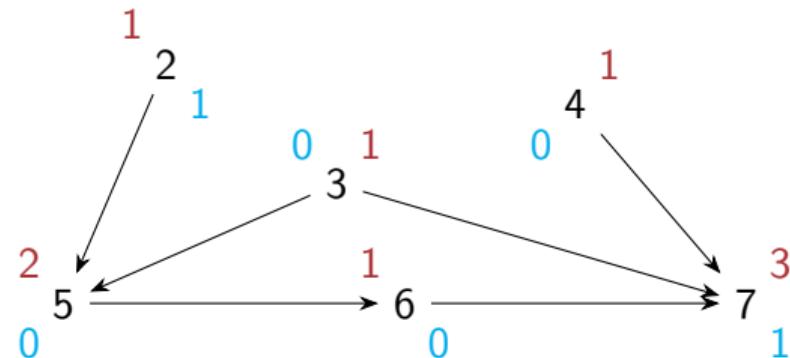


Topological order 1
Longest path to 0

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest-path-to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

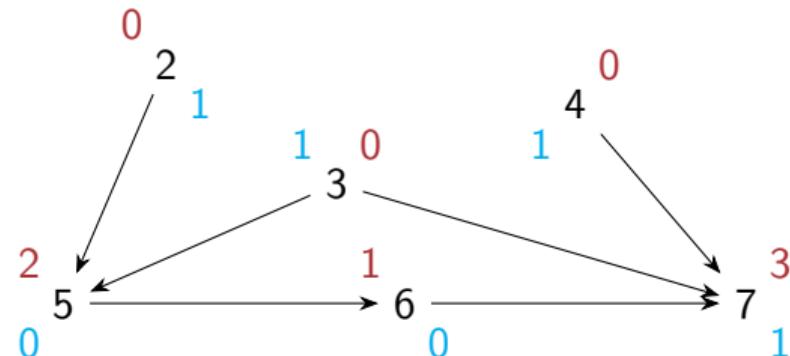


Topological order 1 0
Longest path to 0 0

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest-path-to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

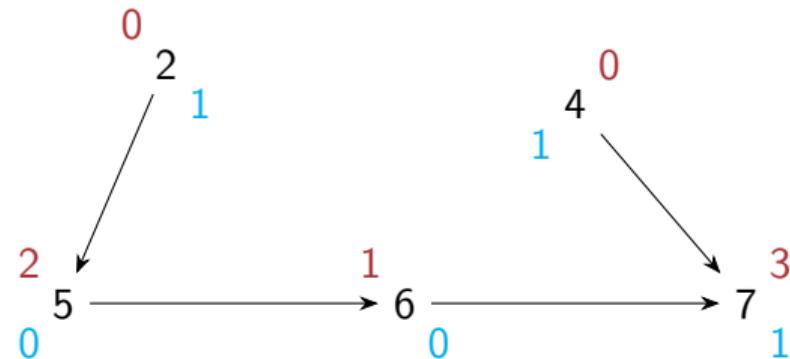


Topological order 1 0
Longest path to 0 0

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest-path-to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

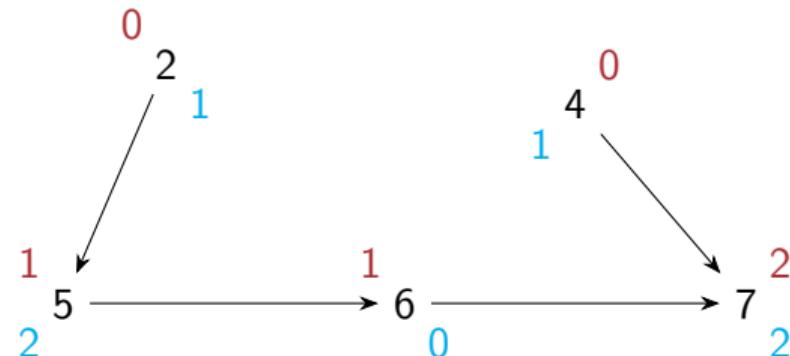


Topological order 1 0 3
Longest path to 0 0 1

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest-path-to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

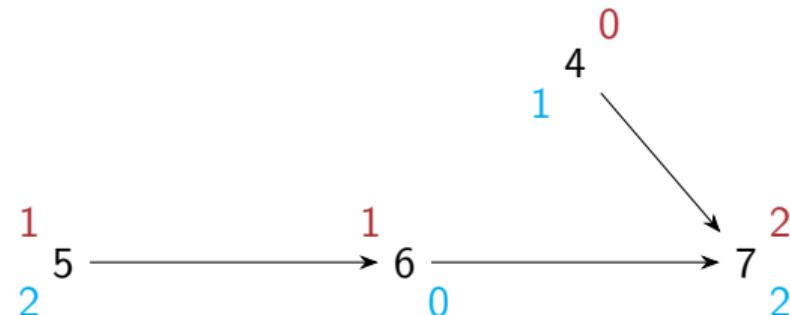


Topological order 1 0 3
Longest path to 0 0 1

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest-path-to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

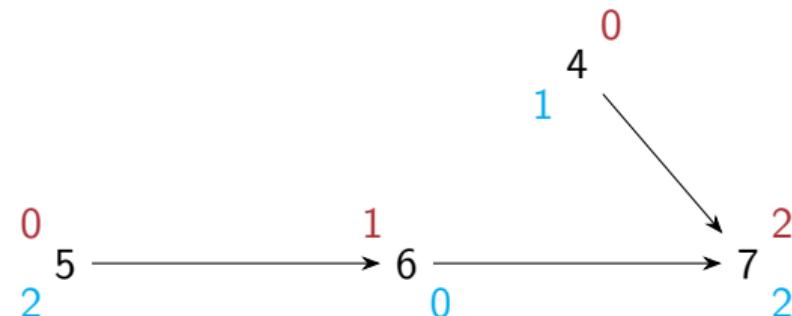


Topological order	1	0	3	2
Longest path to	0	0	1	1

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest-path-to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

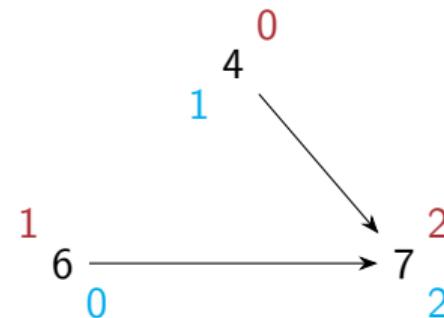


Topological order	1	0	3	2
Longest path to	0	0	1	1

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest-path-to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

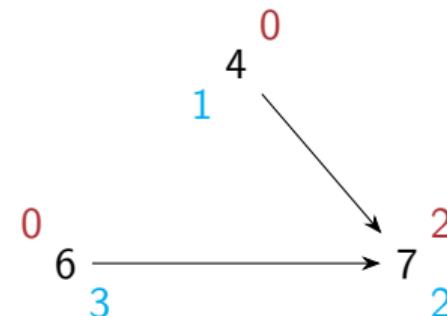


Topological order	1	0	3	2	5
Longest path to	0	0	1	1	2

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest-path-to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

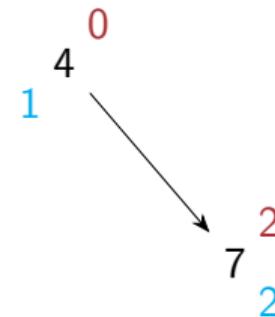


Topological order	1	0	3	2	5
Longest path to	0	0	1	1	2

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest – path – to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

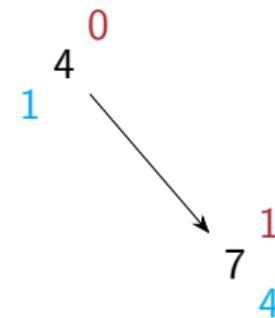


Topological order	1	0	3	2	5	6
Longest path to	0	0	1	1	2	3

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest – path – to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path



Topological order	1	0	3	2	5	6
Longest path to	0	0	1	1	2	3

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize ***longest-path-to*** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

1
7
4

Topological order	1	0	3	2	5	6	4
Longest path to	0	0	1	1	2	3	1

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest – path – to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

0
7
4

Topological order	1	0	3	2	5	6	4
Longest path to	0	0	1	1	2	3	1

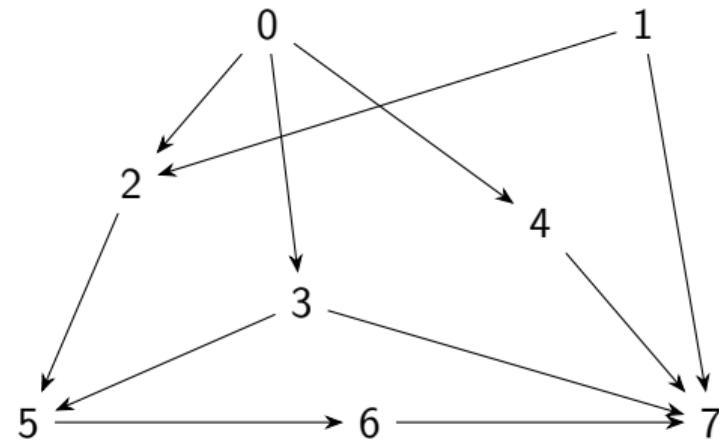
Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize ***longest – path – to*** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Topological order	1	0	3	2	5	6	4	7
Longest path to	0	0	1	1	2	3	1	4

Longest path algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- Initialize **longest – path – to** to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed



Topological order	1	0	3	2	5	6	4	7
Longest path to	0	0	1	1	2	3	1	4

Longest path using adjacency lists

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Process head of queue: update indegrees, update queue, update longest paths
- Repeat till queue is empty

```
def longestpathlist(AList):
    (indegree,lpath) = ({},{})
    for u in AList.keys():
        (indegree[u],lpath[u]) = (0,0)
    for u in AList.keys():
        for v in AList[u]:
            indegree[v] = indegree[v] + 1

    zerodegreeq = Queue()
    for u in AList.keys():
        if indegree[u] == 0:
            zerodegreeq.addq(u)

    while (not zerodegreeq.isempty()):
        j = zerodegreeq.delq()
        indegree[j] = indegree[j]-1
        for k in AList[j]:
            indegree[k] = indegree[k] - 1
            lpath[k] = max(lpath[k],lpath[j]+1)
            if indegree[k] == 0:
                zerodegreeq.addq(k)
    return(lpath)
```

Longest path using adjacency lists

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Process head of queue: update indegrees, update queue, update longest paths
- Repeat till queue is empty

Analysis

```
def longestpathlist(AList):
    (indegree,lpath) = ({},{})
    for u in AList.keys():
        (indegree[u],lpath[u]) = (0,0)
    for u in AList.keys():
        for v in AList[u]:
            indegree[v] = indegree[v] + 1

    zerodegreeq = Queue()
    for u in AList.keys():
        if indegree[u] == 0:
            zerodegreeq.addq(u)

    while (not zerodegreeq.isempty()):
        j = zerodegreeq.delq()
        indegree[j] = indegree[j]-1
        for k in AList[j]:
            indegree[k] = indegree[k] - 1
            lpath[k] = max(lpath[k],lpath[j]+1)
            if indegree[k] == 0:
                zerodegreeq.addq(k)
    return(lpath)
```

Longest path using adjacency lists

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Process head of queue: update indegrees, update queue, update longest paths
- Repeat till queue is empty

Analysis

- Initializing indegrees is $O(m + n)$

```
def longestpathlist(AList):
    (indegree,lpath) = ({},{})
    for u in AList.keys():
        (indegree[u],lpath[u]) = (0,0)
    for u in AList.keys():
        for v in AList[u]:
            indegree[v] = indegree[v] + 1

    zerodegreeq = Queue()
    for u in AList.keys():
        if indegree[u] == 0:
            zerodegreeq.addq(u)

    while (not zerodegreeq.isempty()):
        j = zerodegreeq.delq()
        indegree[j] = indegree[j]-1
        for k in AList[j]:
            indegree[k] = indegree[k] - 1
            lpath[k] = max(lpath[k],lpath[j]+1)
            if indegree[k] == 0:
                zerodegreeq.addq(k)
    return(lpath)
```

Longest path using adjacency lists

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Process head of queue: update indegrees, update queue, update longest paths
- Repeat till queue is empty

Analysis

- Initializing indegrees is $O(m + n)$
- Loop to enumerate vertices runs n times
 - Updating indegrees, longest path: amortised $O(m)$

```
def longestpathlist(AList):
    (indegree,lpath) = ({},{})
    for u in AList.keys():
        (indegree[u],lpath[u]) = (0,0)
    for u in AList.keys():
        for v in AList[u]:
            indegree[v] = indegree[v] + 1

    zerodegreeq = Queue()
    for u in AList.keys():
        if indegree[u] == 0:
            zerodegreeq.addq(u)

    while (not zerodegreeq.isempty()):
        j = zerodegreeq.delq()
        indegree[j] = indegree[j]-1
        for k in AList[j]:
            indegree[k] = indegree[k] - 1
            lpath[k] = max(lpath[k],lpath[j]+1)
            if indegree[k] == 0:
                zerodegreeq.addq(k)
    return(lpath)
```

Longest path using adjacency lists

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Process head of queue: update indegrees, update queue, update longest paths
- Repeat till queue is empty

Analysis

- Initializing indegrees is $O(m + n)$
- Loop to enumerate vertices runs n times
 - Updating indegrees, longest path: amortised $O(m)$
- Overall, $O(m + n)$

```
def longestpathlist(AList):
    (indegree,lpath) = ({}, {})
    for u in AList.keys():
        (indegree[u],lpath[u]) = (0,0)
    for u in AList.keys():
        for v in AList[u]:
            indegree[v] = indegree[v] + 1

    zerodegreeq = Queue()
    for u in AList.keys():
        if indegree[u] == 0:
            zerodegreeq.addq(u)

    while (not zerodegreeq.isempty()):
        j = zerodegreeq.delq()
        indegree[j] = indegree[j]-1
        for k in AList[j]:
            indegree[k] = indegree[k] - 1
            lpath[k] = max(lpath[k],lpath[j]+1)
            if indegree[k] == 0:
                zerodegreeq.addq(k)
    return(lpath)
```

Summary

- Directed acyclic graphs are a natural way to represent dependencies
- Topological sort gives a feasible schedule that represents dependencies
- In parallel with topological sort, we can compute the longest path

Summary

- Directed acyclic graphs are a natural way to represent dependencies
- Topological sort gives a feasible schedule that represents dependencies
- In parallel with topological sort, we can compute the longest path
- Notion of longest path makes sense even for graphs with cycles
 - No repeated vertices in a path, so path has at most $n - 1$ edges

Summary

- Directed acyclic graphs are a natural way to represent dependencies
- Topological sort gives a feasible schedule that represents dependencies
- In parallel with topological sort, we can compute the longest path
- Notion of longest path makes sense even for graphs with cycles
 - No repeated vertices in a path, so path has at most $n - 1$ edges
- However, computing longest paths in arbitrary graphs is much harder than for DAGs
 - No better strategy known than exhaustively enumerating paths

Shortest Paths in Weighted Graphs

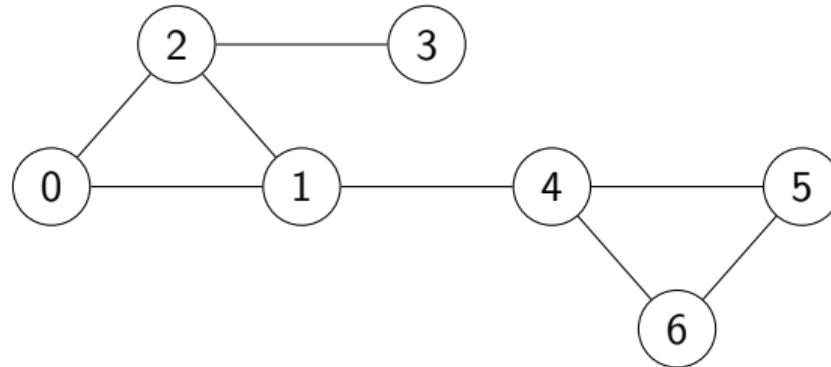
Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 5

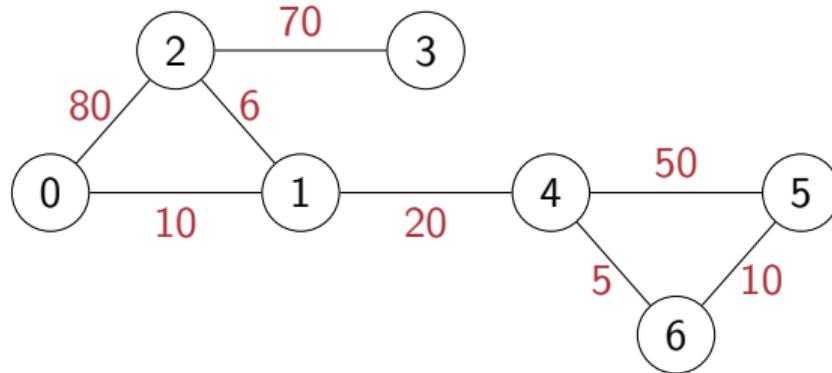
Weighted graphs

- Recall that BFS explores a graph level by level
- BFS computes shortest path, in terms of number of edges, to every reachable vertex



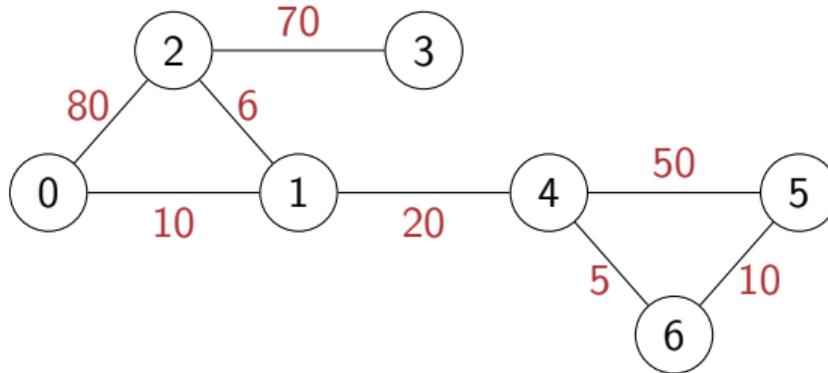
Weighted graphs

- Recall that BFS explores a graph level by level
- BFS computes shortest path, in terms of number of edges, to every reachable vertex
- May assign values to edges
 - Cost, time, distance, ...
 - Weighted graph
- $G = (V, E), W : E \rightarrow \mathbb{R}$



Weighted graphs

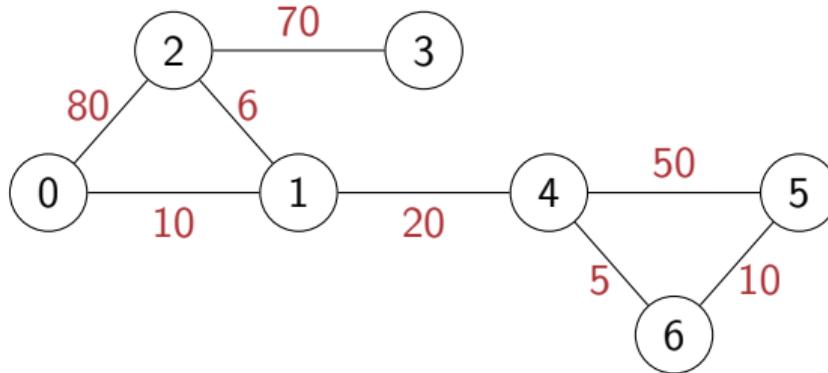
- Recall that BFS explores a graph level by level
- BFS computes shortest path, in terms of number of edges, to every reachable vertex
- May assign values to edges
 - Cost, time, distance, ...
 - Weighted graph
- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Adjacency matrix
Record weights along with edge information — weight is always 0 if no edge



	0	1	2	3	4	5	6
0	(0,0)	(1,10)	(1,80)	(0,0)	(0,0)	(0,0)	(0,0)
1	(1,10)	(0,0)	(1,6)	(0,0)	(1,20)	(0,0)	(0,0)
2	(1,80)	(1,6)	(0,0)	(1,70)	(0,0)	(0,0)	(0,0)
3	(0,0)	(0,0)	(1,70)	(0,0)	(0,0)	(0,0)	(0,0)
4	(0,0)	(1,20)	(0,0)	(0,0)	(0,0)	(1,50)	(1,5)
5	(0,0)	(0,0)	(0,0)	(0,0)	(1,50)	(0,0)	(1,10)
6	(0,0)	(0,0)	(0,0)	(0,0)	(1,5)	(1,10)	(0,0)

Weighted graphs

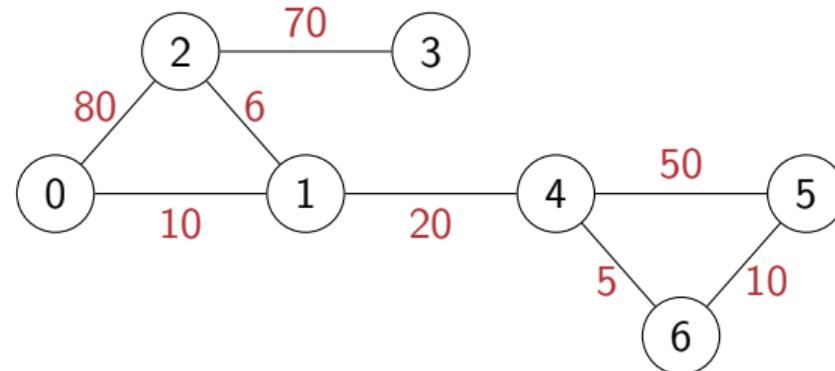
- Recall that BFS explores a graph level by level
- BFS computes shortest path, in terms of number of edges, to every reachable vertex
- May assign values to edges
 - Cost, time, distance, ...
 - Weighted graph
- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Adjacency list
Record weights along with edge information



0	$[(1,10),(2,80)]$
1	$[(0,10),(2,6),(4,20)]$
2	$[(0,80),(1,6),(3,70)]$
3	$[(2,70)]$
4	$[(1,20),(5,50),(6,5)]$
5	$[(4,50),(6,10)]$
6	$[(4,5),(5,10)]$

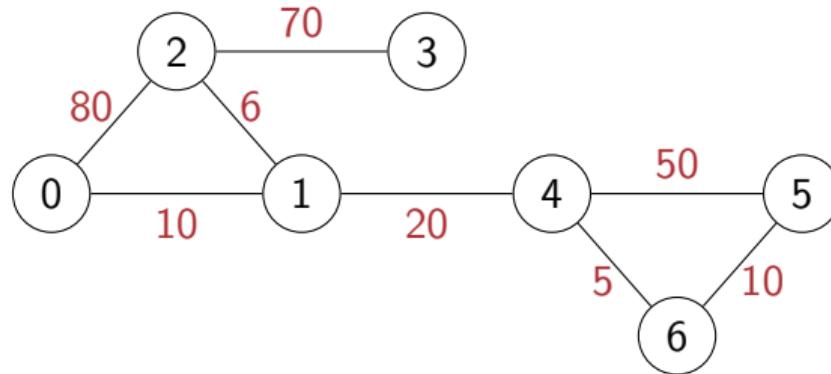
Shortest paths in weighted graphs

- BFS computes shortest path, in terms of number of edges, to every reachable vertex



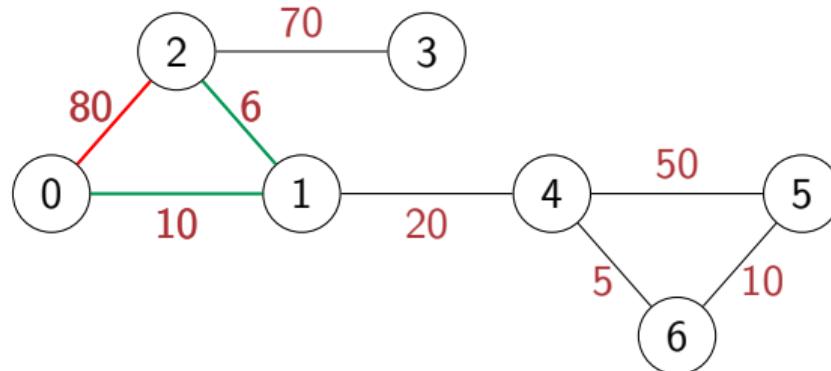
Shortest paths in weighted graphs

- BFS computes shortest path, in terms of number of edges, to every reachable vertex
- In a weighted graph, add up the weights along a path



Shortest paths in weighted graphs

- BFS computes shortest path, in terms of number of edges, to every reachable vertex
- In a weighted graph, add up the weights along a path
- Weighted shortest path need not have minimum number of edges
 - Shortest path from 0 to 2 is via 1



Shortest path problems

Single source shortest paths

- Find shortest paths from a fixed vertex to every other vertex
- Transport finished product from factory (single source) to all retail outlets
- Courier company delivers items from distribution centre (single source) to addressees

Shortest path problems

Single source shortest paths

- Find shortest paths from a fixed vertex to every other vertex
- Transport finished product from factory (single source) to all retail outlets
- Courier company delivers items from distribution centre (single source) to addressees

All pairs shortest paths

- Find shortest paths between every pair of vertices i and j
- Optimal airline, railway, road routes between cities

Negative edge weights

Negative edge weights

- Can negative edge weights be meaningful?
- Taxi driver trying to head home at the end of the day
 - Roads with few customers, drive empty (positive weight)
 - Roads with many customers, make profit (negative weight)
 - Find a route toward home that minimizes the cost

Negative edge weights

Negative edge weights

- Can negative edge weights be meaningful?
- Taxi driver trying to head home at the end of the day
 - Roads with few customers, drive empty (positive weight)
 - Roads with many customers, make profit (negative weight)
 - Find a route toward home that minimizes the cost

Negative cycles

- A negative cycle is one whose weight is negative
 - Sum of the weights of edges that make up the cycle
- By repeatedly traversing a negative cycle, total cost keeps decreasing
- If a graph has a negative cycle, shortest paths are not defined
- Without negative cycles, we can compute shortest paths even if some weights are negative

Summary

- In a weighted graph, each edge has a cost
 - Entries in adjacency matrix capture edge weights
- Length of a path is the sum of the weights
 - Shortest path in a weighted graph need not be minimum in terms of number of edges
- Different shortest path problems
 - Single source — from one designated vertex to all others
 - All-pairs — between every pair of vertices
- Negative edge weights
 - Should not have negative cycles
 - Without negative cycles, shortest paths still well defined

Single Source Shortest Paths

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 5

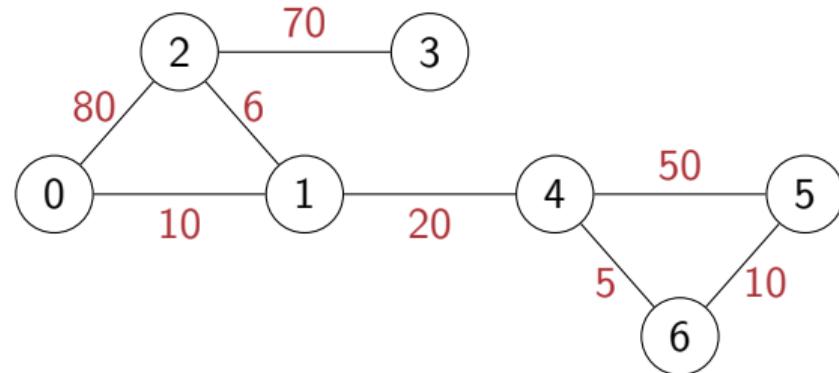
Single source shortest paths

- Weighted graph:

- $G = (V, E)$
- $W : E \rightarrow \mathbb{R}$

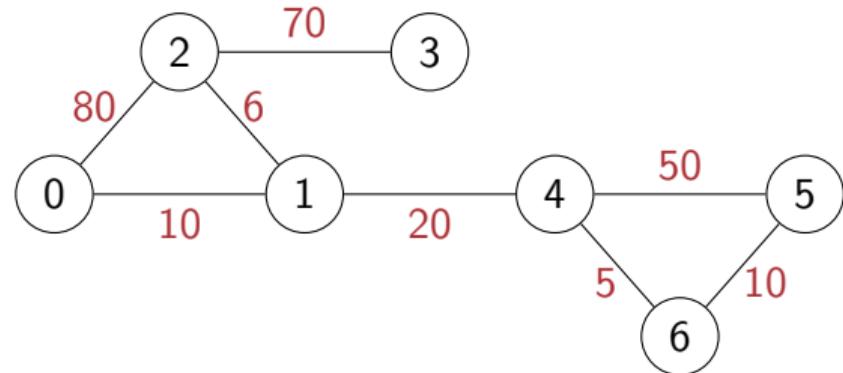
- Single source shortest paths

- Find shortest paths from a fixed vertex to every other vertex



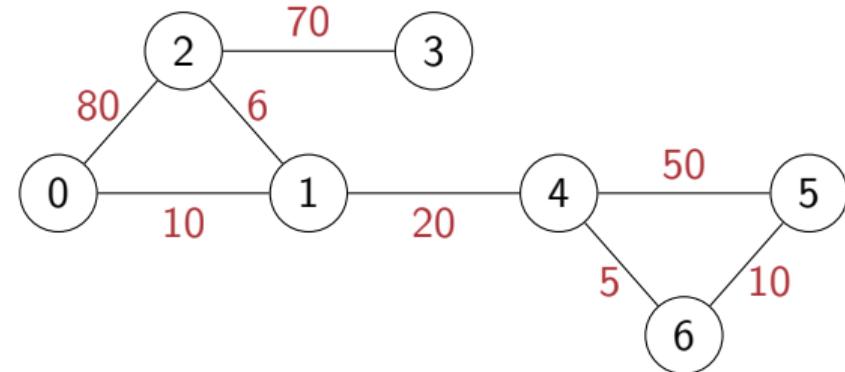
Single source shortest paths

- Weighted graph:
 - $G = (V, E)$
 - $W : E \rightarrow \mathbb{R}$
- Single source shortest paths
 - Find shortest paths from a fixed vertex to every other vertex
- Assume, for now, that edge weights are all non-negative



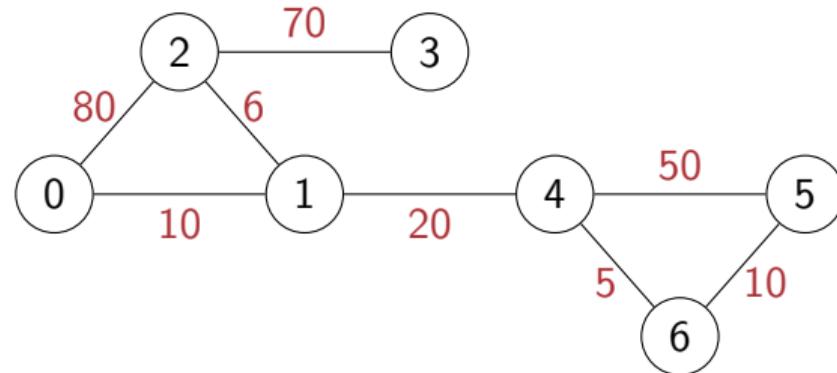
Single source shortest paths

- Compute shortest paths from 0 to all other vertices



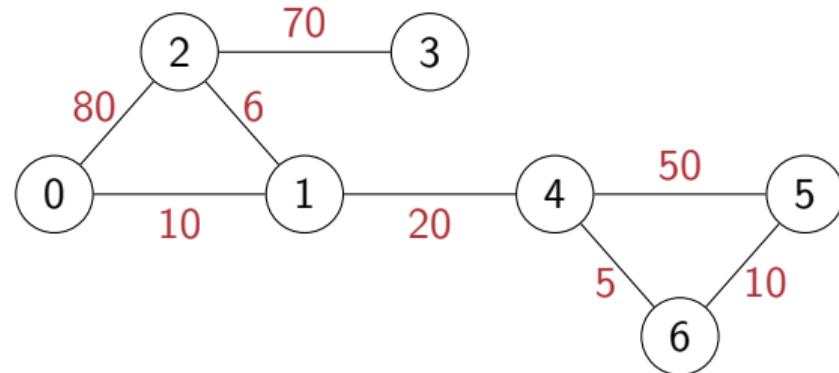
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines



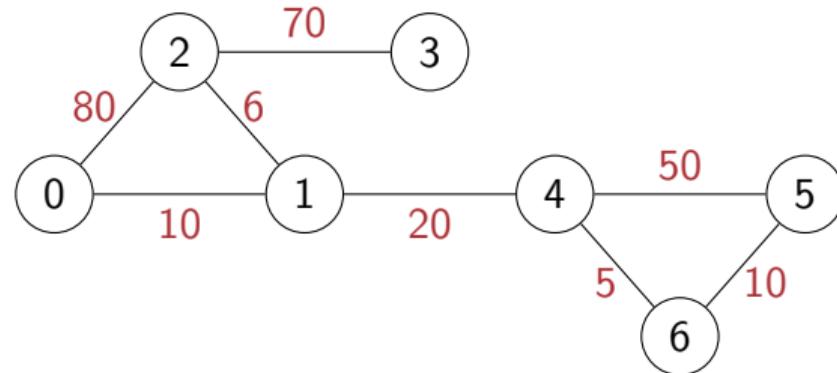
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 0



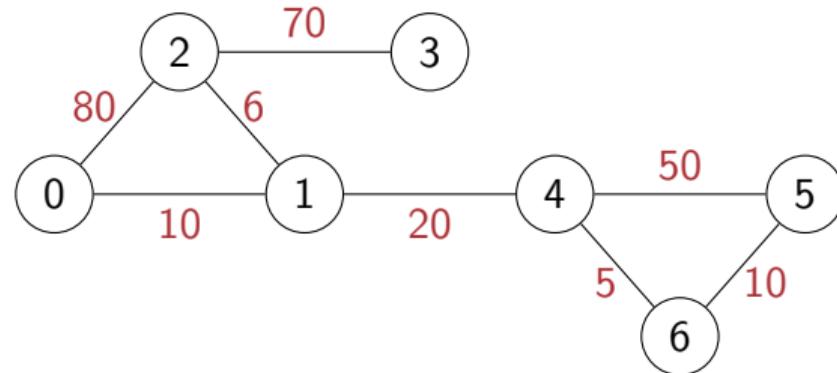
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline



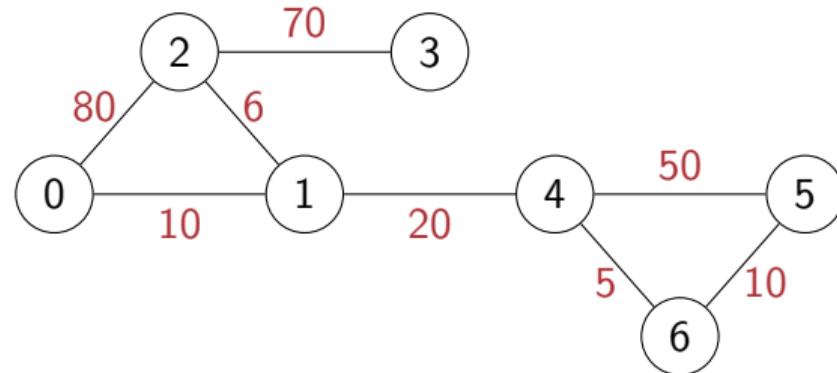
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex



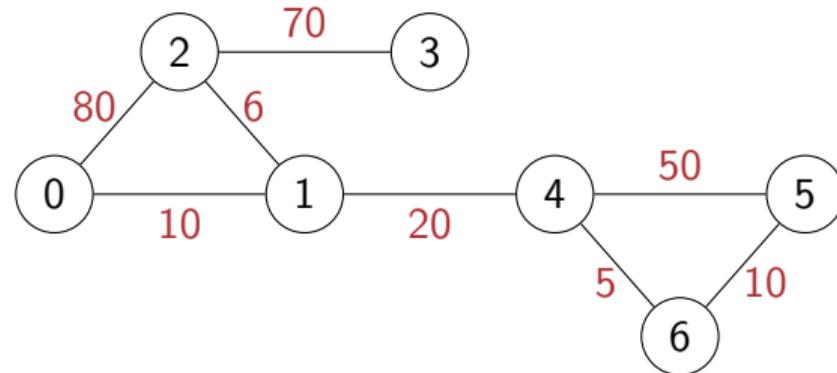
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex



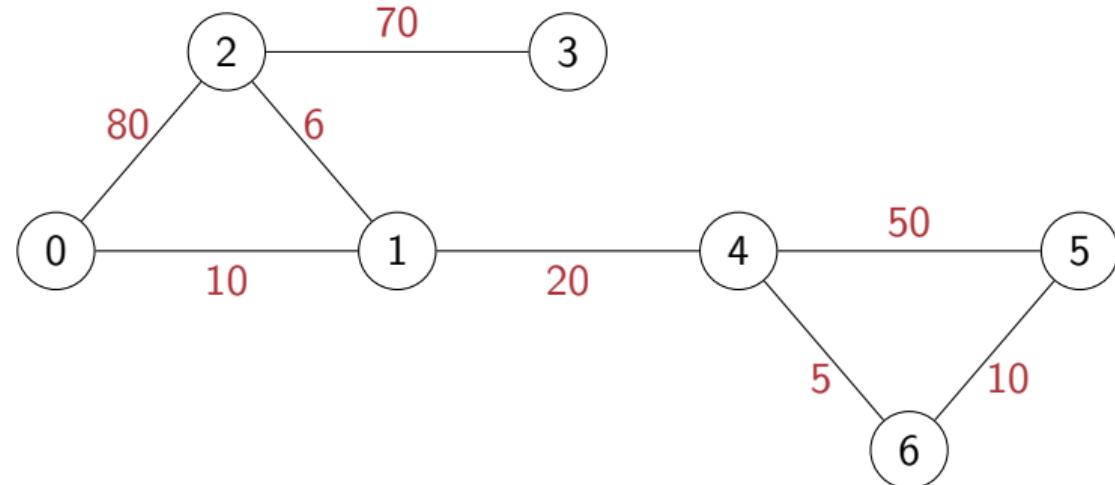
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



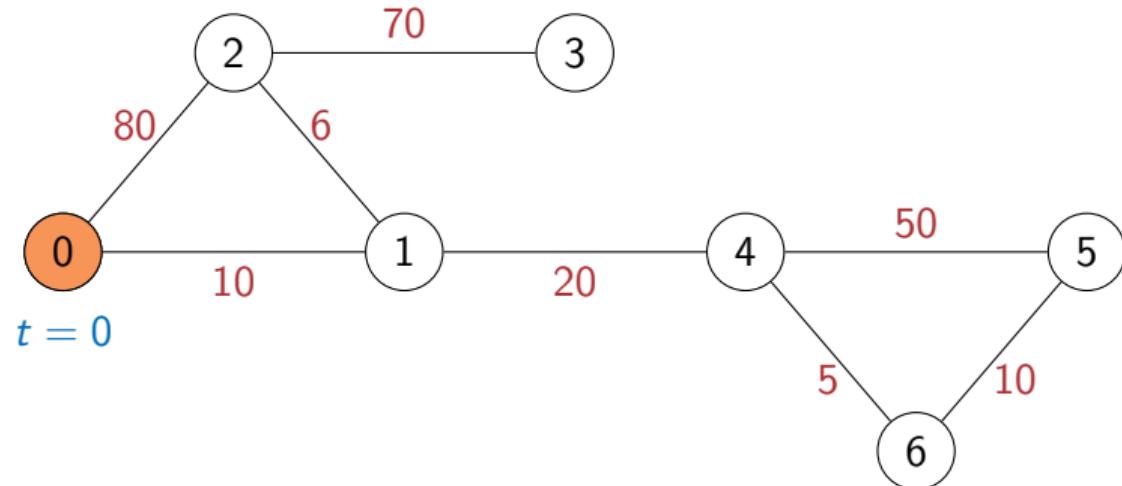
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



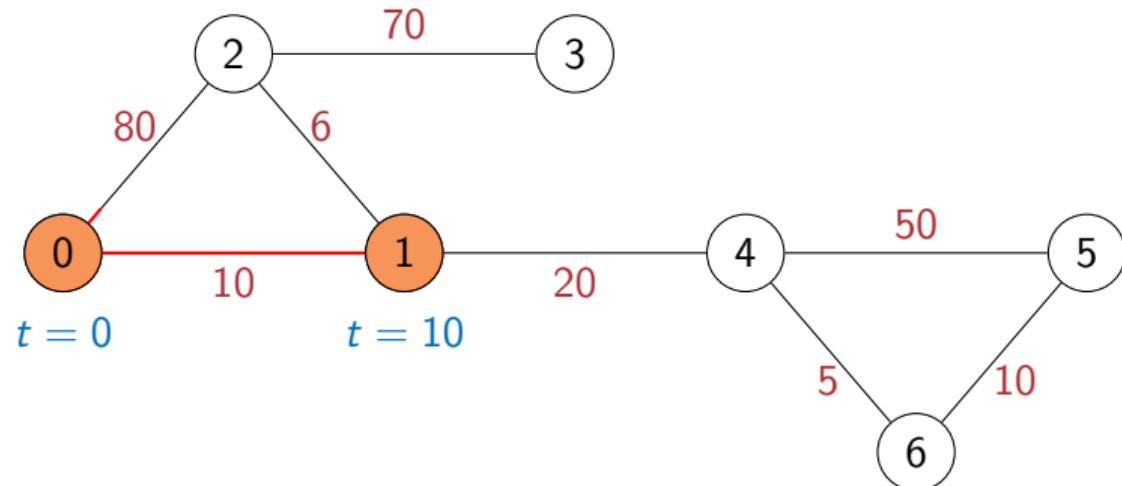
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



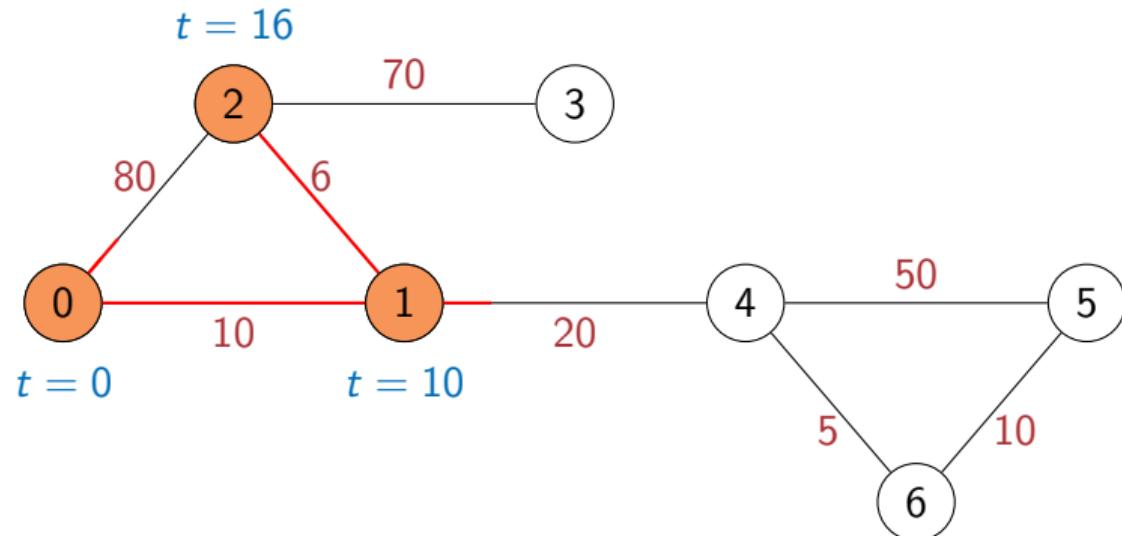
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



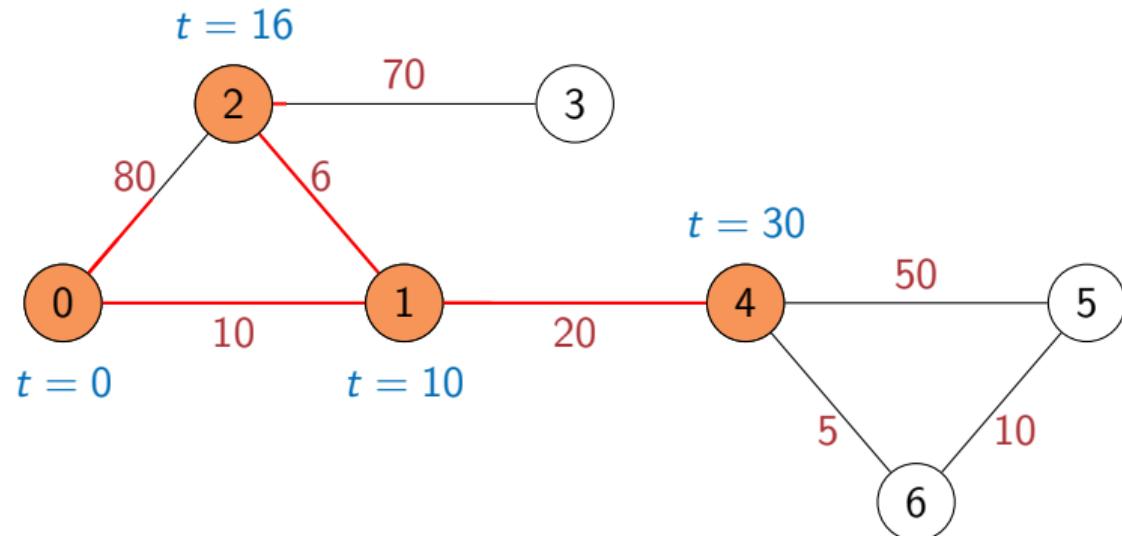
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



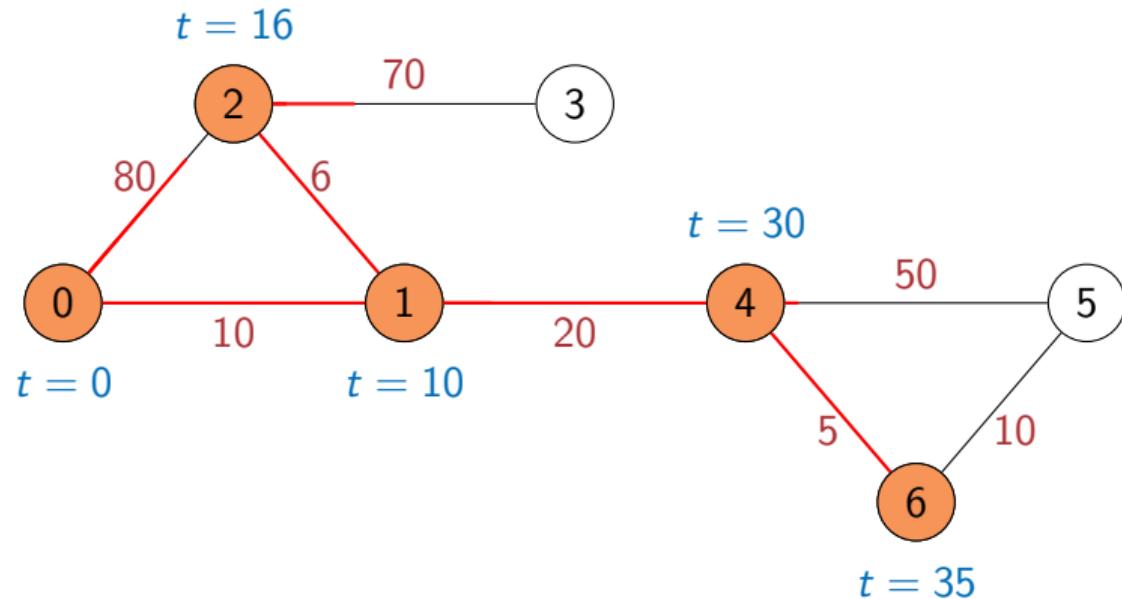
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



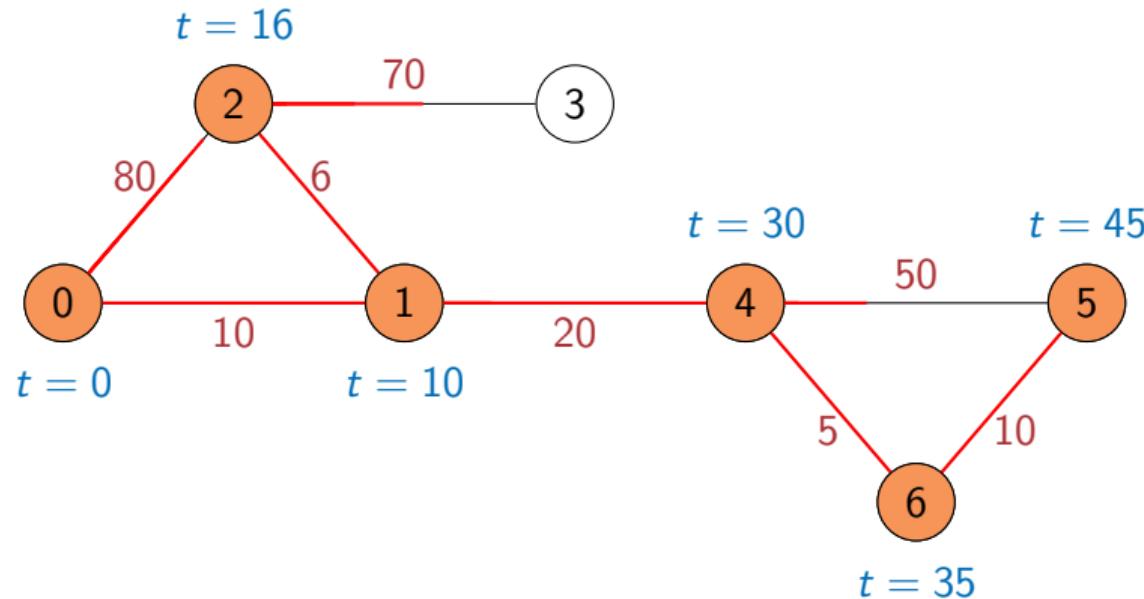
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



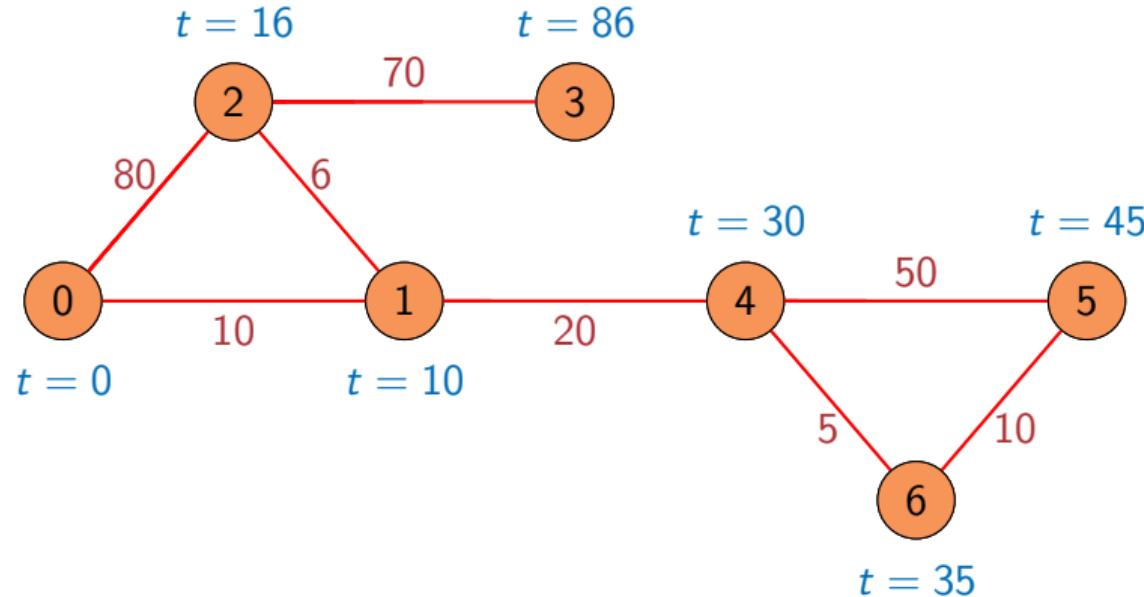
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



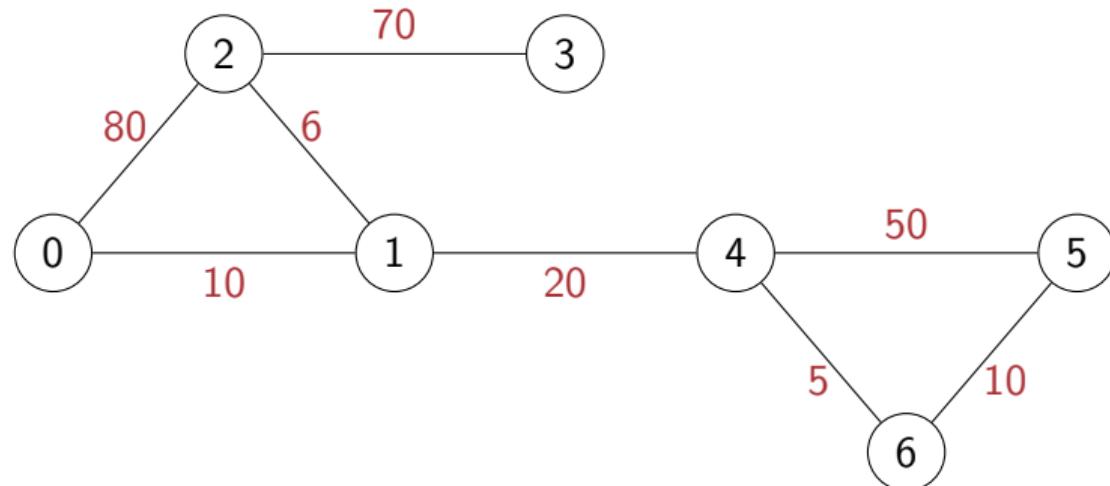
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



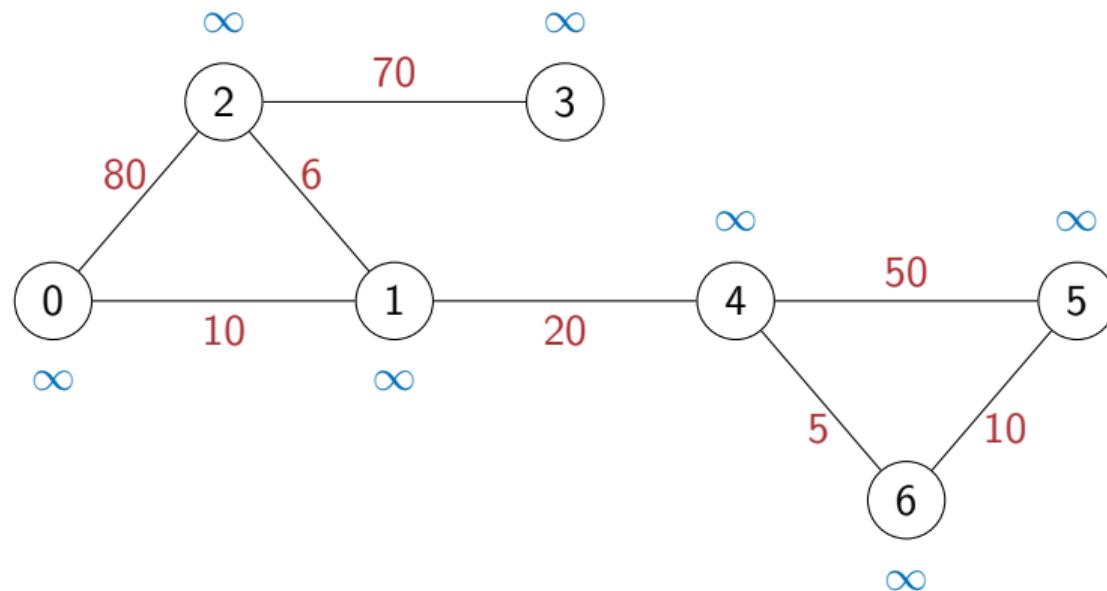
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



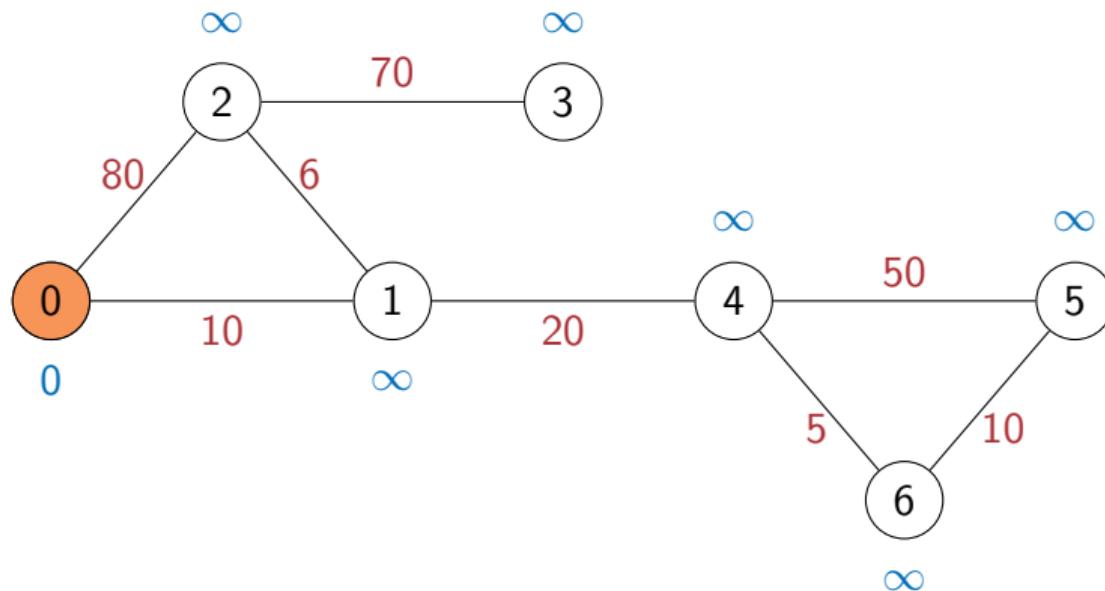
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



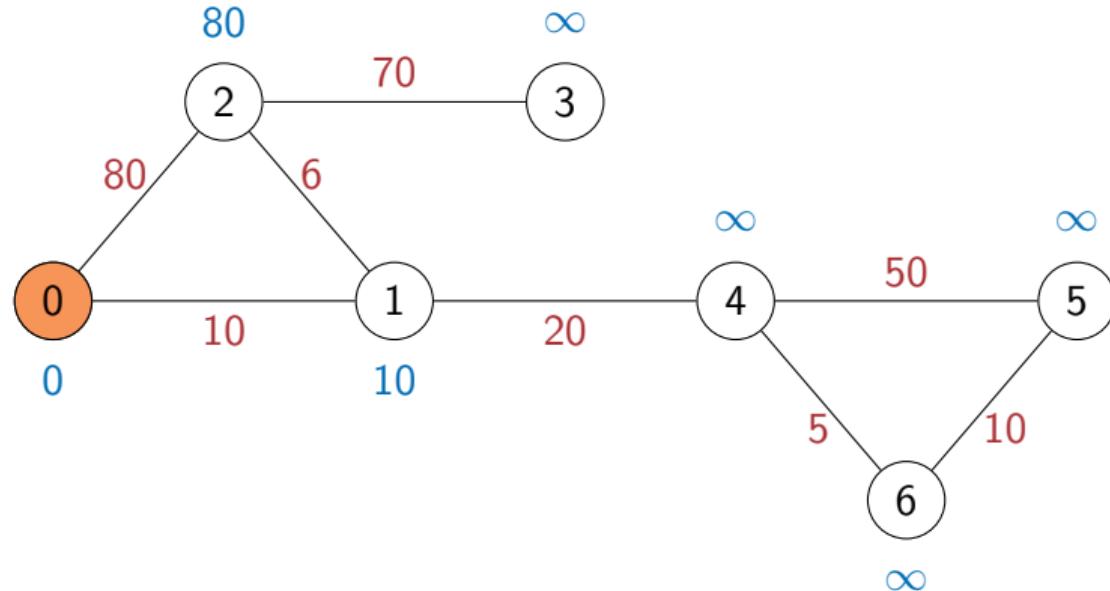
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



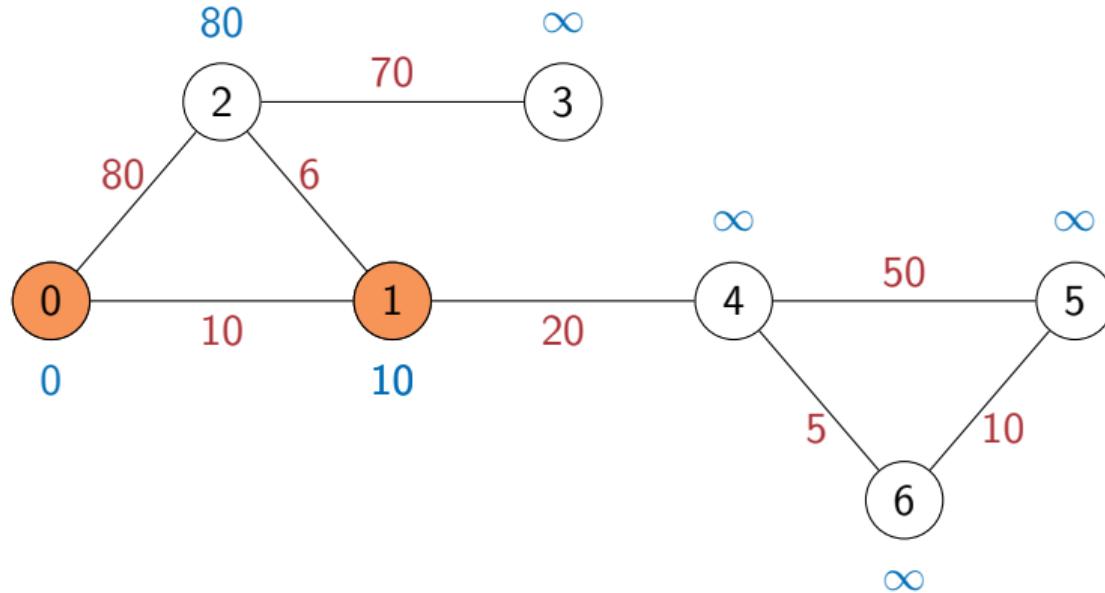
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



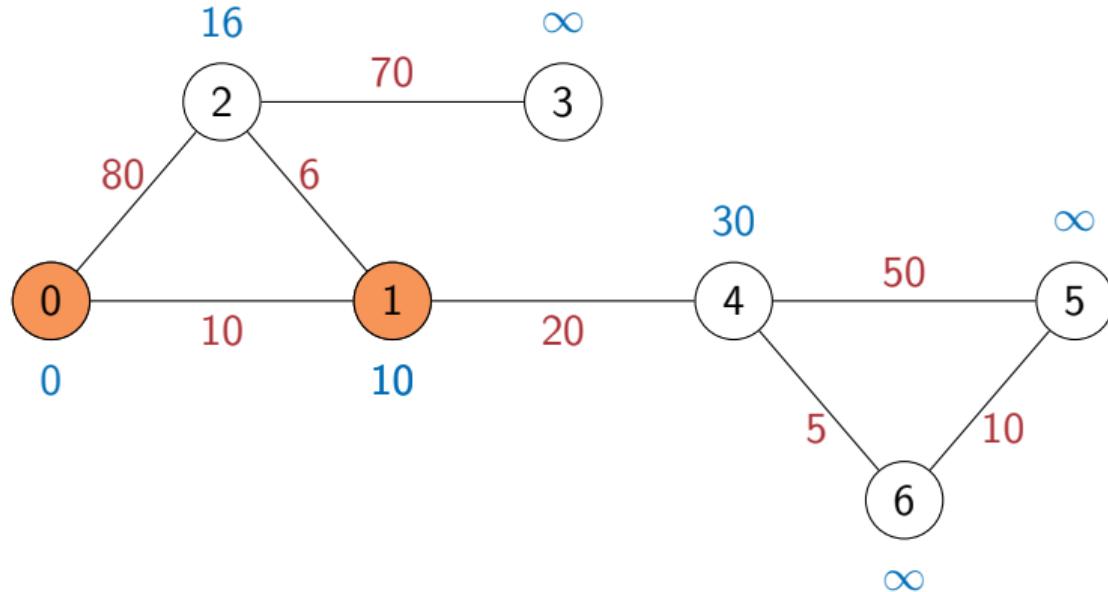
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



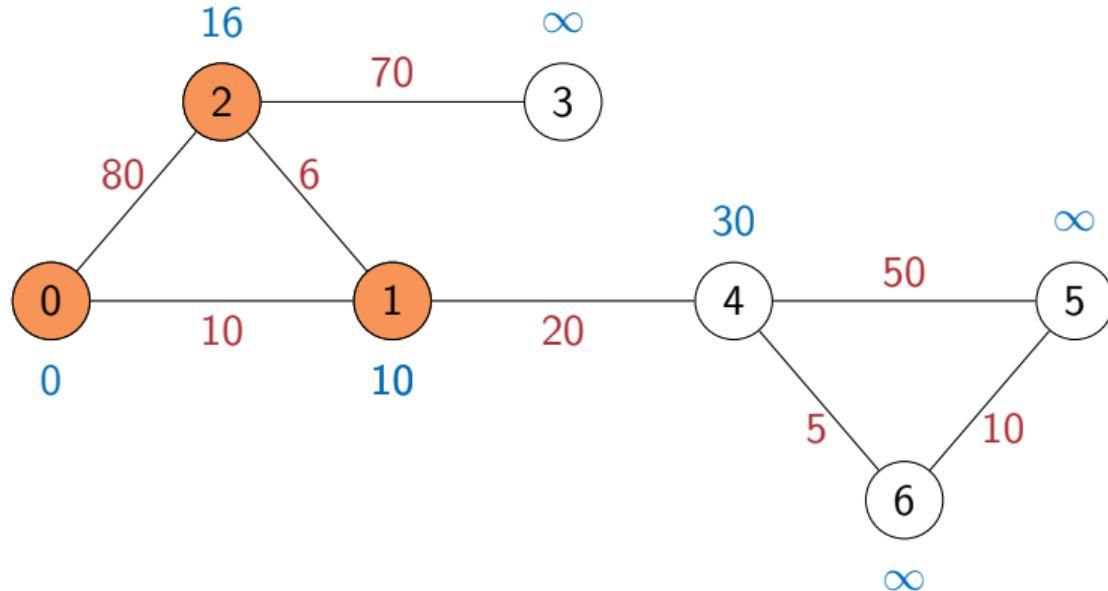
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



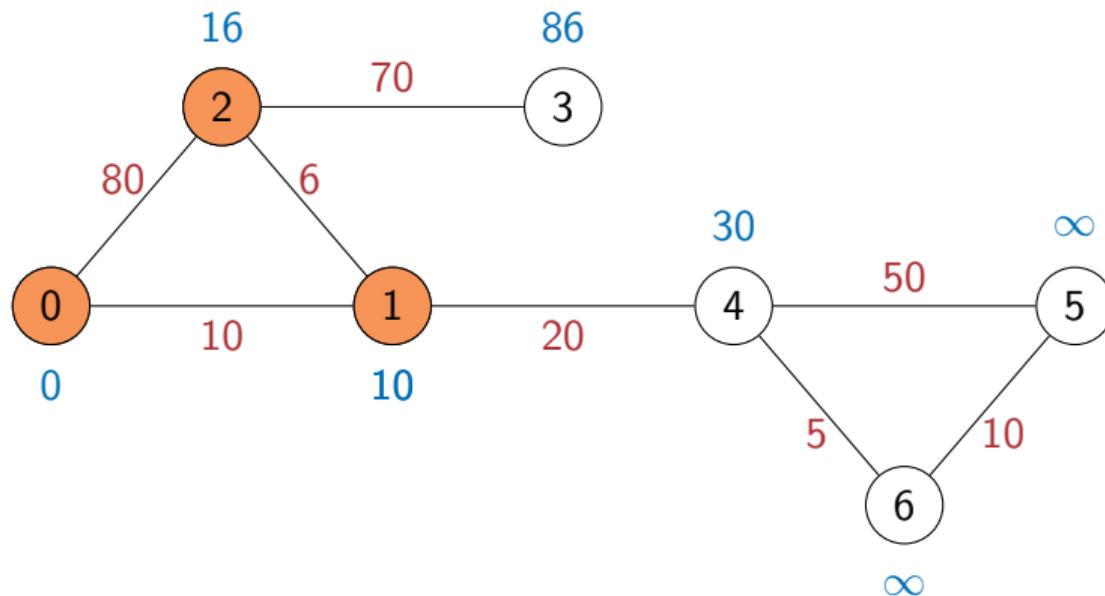
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



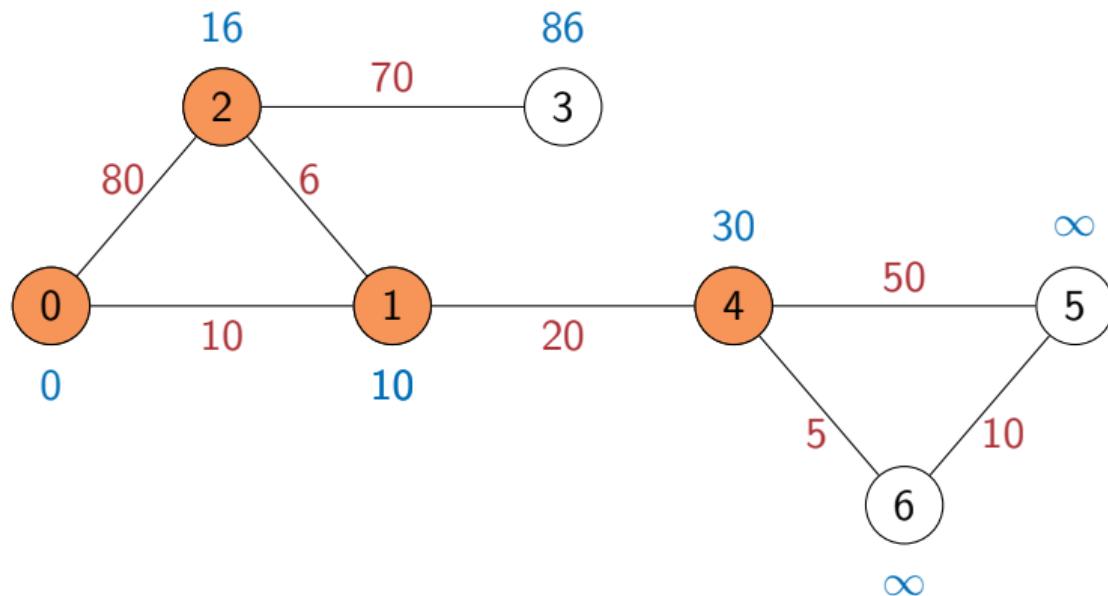
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



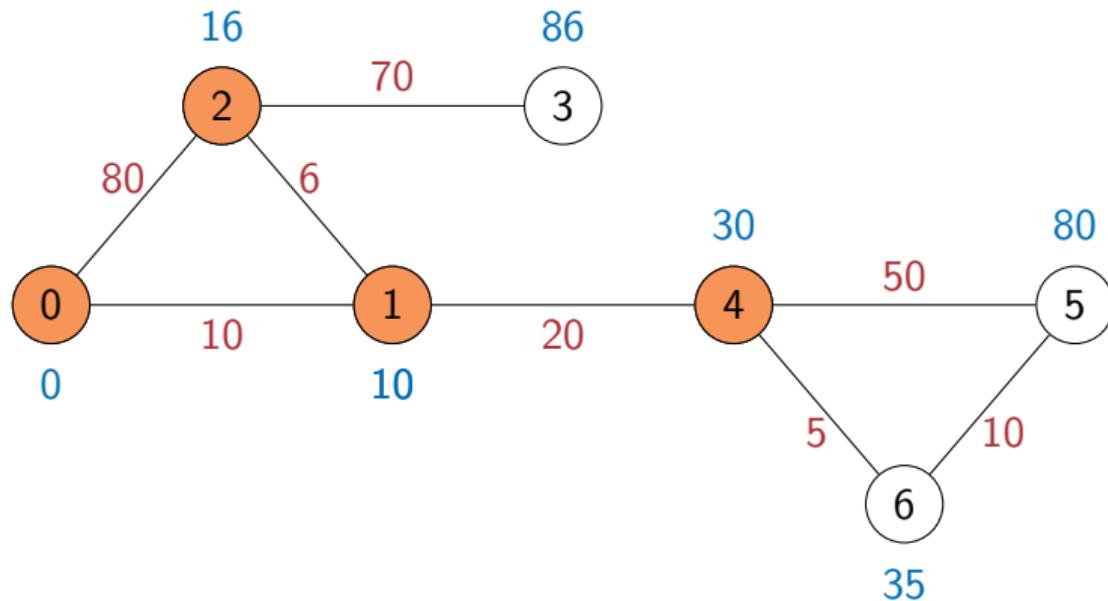
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



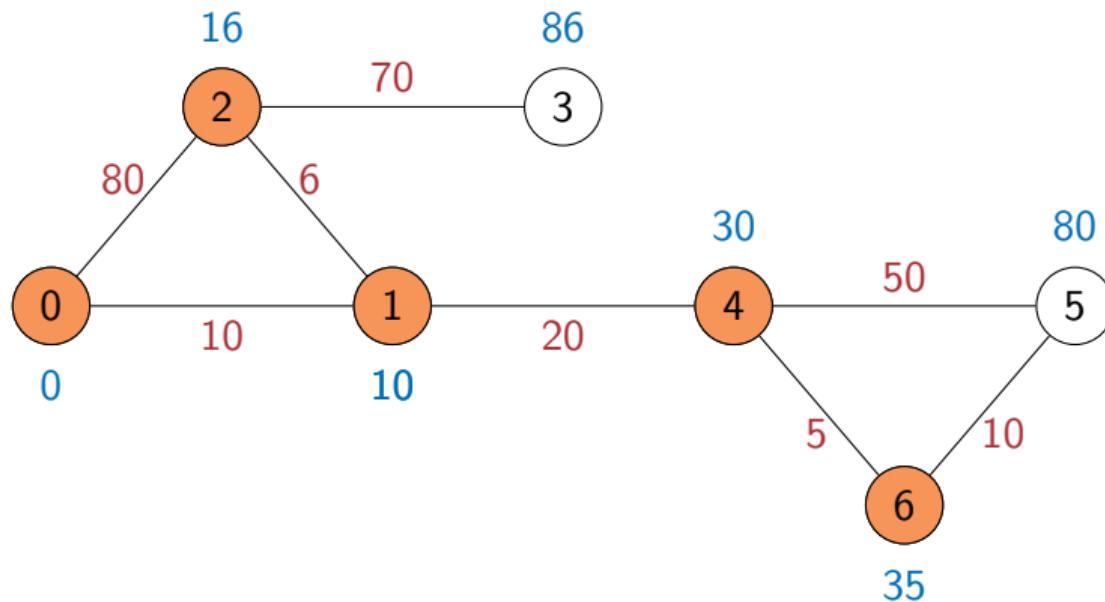
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



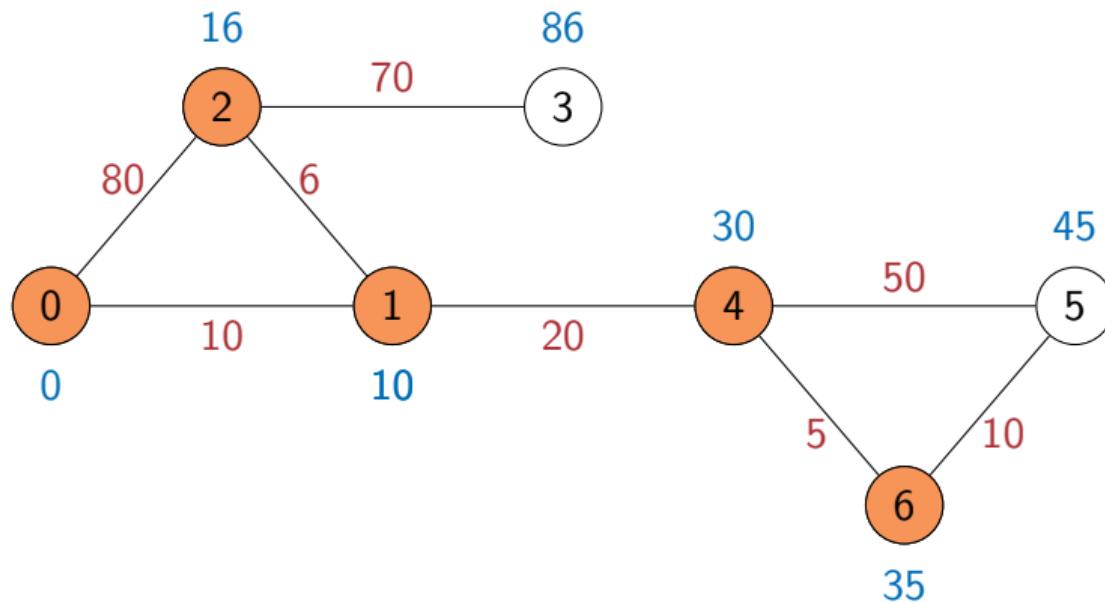
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



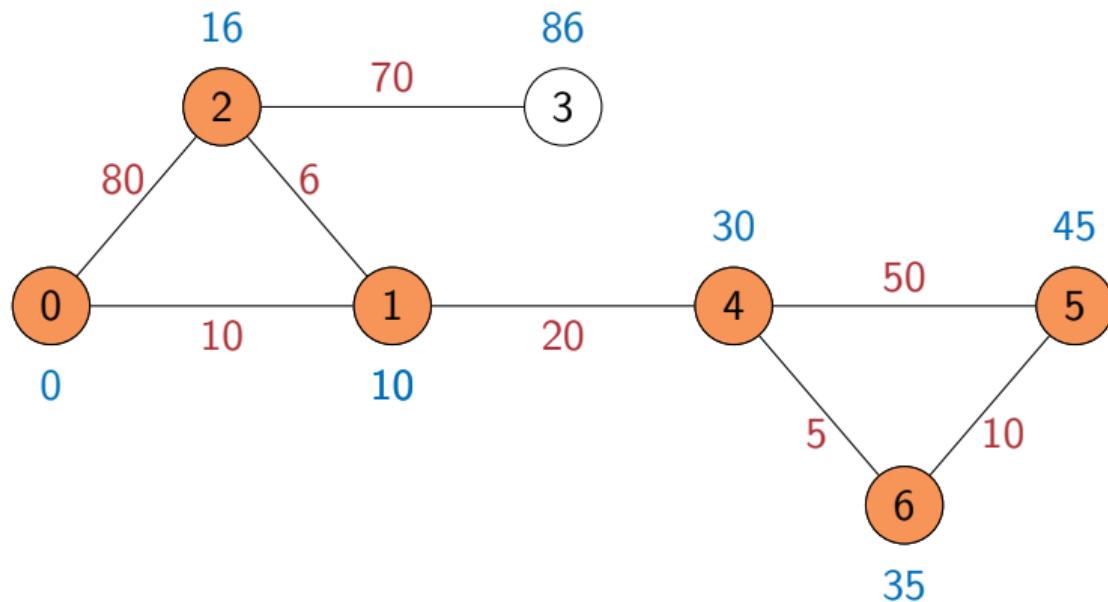
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



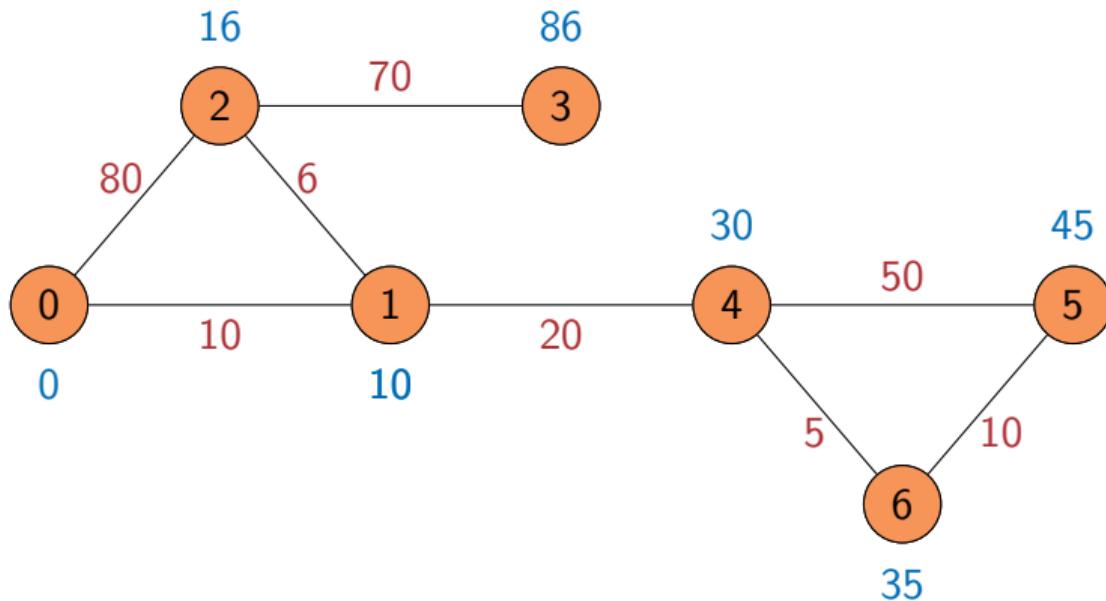
Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



Single source shortest paths

- Compute **expected burn time** for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours
- Algorithm due to Edsger W Dijkstra



Dijkstra's algorithm: Proof of correctness

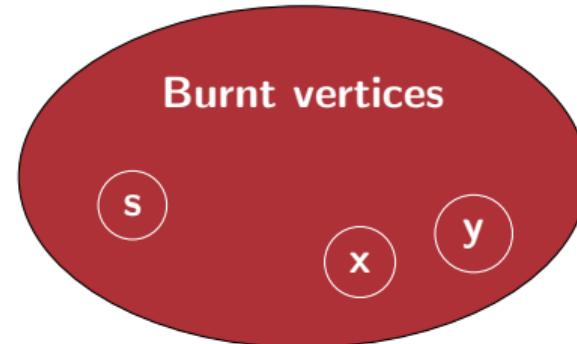
- Each new shortest path we discover extends an earlier one

Dijkstra's algorithm: Proof of correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt

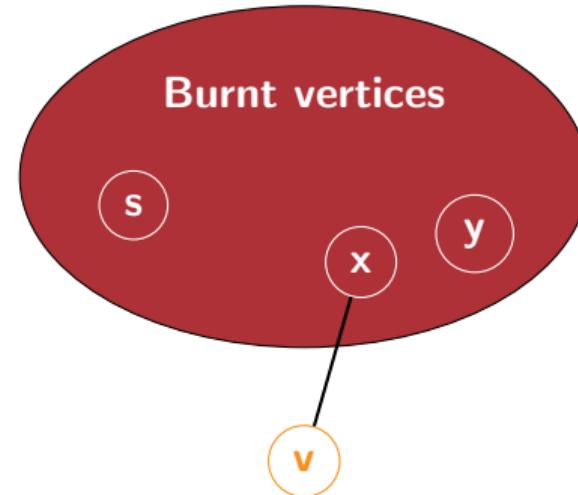
Dijkstra's algorithm: Proof of correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt



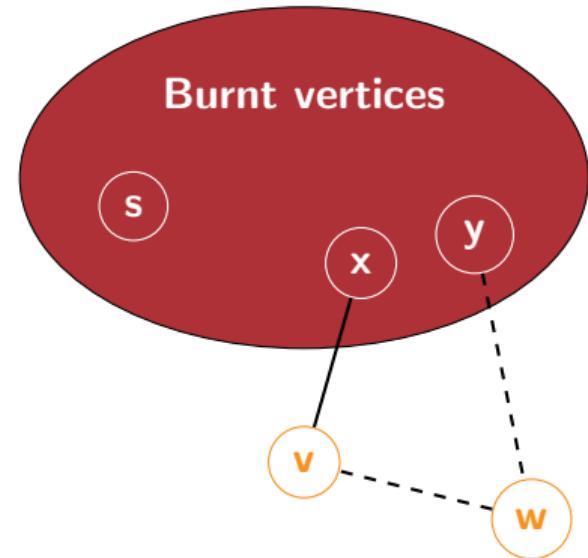
Dijkstra's algorithm: Proof of correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt
- Next vertex to burn is v , via x



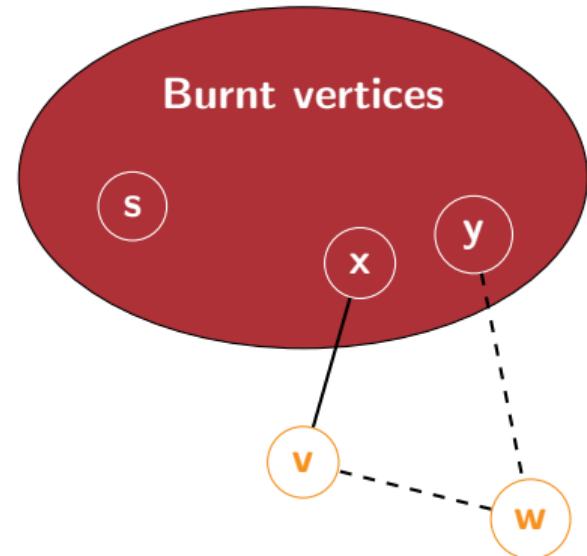
Dijkstra's algorithm: Proof of correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt
- Next vertex to burn is v , via x
- Cannot find a shorter path later from y to v via w
 - Burn time of $w \geq$ burn time of v
 - Edge from w to v has weight ≥ 0



Dijkstra's algorithm: Proof of correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt
- Next vertex to burn is v , via x
- Cannot find a shorter path later from y to v via w
 - Burn time of $w \geq$ burn time of v
 - Edge from w to v has weight ≥ 0
- This argument breaks down if edge (w,v) can have negative weight
 - Can't use Dijkstra's algorithm with negative edge weights



Implementation

- Maintain two dictionaries with vertices as keys
 - `visited`, initially `False` for all `v` (burnt vertices)
 - `distance`, initially `infinity` for all `v` (expected burn time)

```
def dijkstra(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    (visited,distance) = ({},{} )  
    for v in range(rows):  
        (visited[v],distance[v]) = (False,infinity)  
    distance[s] = 0  
    for u in range(rows):  
        nextd = min([distance[v] for v in range(rows)  
                    if not visited[v]])  
        nextvlist = [v for v in range(rows)  
                    if (not visited[v]) and  
                    distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for v in range(cols):  
            if WMat[nextv,v,0] == 1 and (not visited[v]):  
                distance[v] = min(distance[v],distance[nextv]  
                                  +WMat[nextv,v,1])  
    return(distance)
```

Implementation

- Maintain two dictionaries with vertices as keys
 - `visited`, initially `False` for all `v` (burnt vertices)
 - `distance`, initially `infinity` for all `v` (expected burn time)
- Set `distance[s]` to 0

```
def dijkstra(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    (visited,distance) = ({},{} )  
    for v in range(rows):  
        (visited[v],distance[v]) = (False,infinity)  
    distance[s] = 0  
    for u in range(rows):  
        nextd = min([distance[v] for v in range(rows)  
                    if not visited[v]])  
        nextvlist = [v for v in range(rows)  
                    if (not visited[v]) and  
                        distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for v in range(cols):  
            if WMat[nextv,v,0] == 1 and (not visited[v]):  
                distance[v] = min(distance[v],distance[nextv]  
                                +WMat[nextv,v,1])  
    return(distance)
```

Implementation

- Maintain two dictionaries with vertices as keys
 - `visited`, initially `False` for all `v` (burnt vertices)
 - `distance`, initially `infinity` for all `v` (expected burn time)
- Set `distance[s]` to 0
- Repeat, until all reachable vertices are visited
 - Find unvisited vertex `nextv` with minimum distance
 - Set `visited[nextv]` to `True`
 - Recompute `distance[v]` for every neighbour `v` of `nextv`

```
def dijkstra(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    (visited,distance) = ({},{} )  
    for v in range(rows):  
        (visited[v],distance[v]) = (False,infinity)  
    distance[s] = 0  
    for u in range(rows):  
        nextd = min([distance[v] for v in range(rows)  
                    if not visited[v]])  
        nextvlist = [v for v in range(rows)  
                    if (not visited[v]) and  
                        distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for v in range(cols):  
            if WMat[nextv,v,0] == 1 and (not visited[v]):  
                distance[v] = min(distance[v],distance[nextv]  
                                +WMat[nextv,v,1])  
    return(distance)
```

Complexity

- Setting `infinity` takes $O(n^2)$ time

```
def dijkstra(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    (visited,distance) = ({},{} )  
    for v in range(rows):  
        (visited[v],distance[v]) = (False,infinity)  
    distance[s] = 0  
    for u in range(rows):  
        nextd = min([distance[v] for v in range(rows)  
                    if not visited[v]])  
        nextvlist = [v for v in range(rows)  
                    if (not visited[v]) and  
                        distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for v in range(cols):  
            if WMat[nextv,v,0] == 1 and (not visited[v]):  
                distance[v] = min(distance[v],distance[nextv]  
                                +WMat[nextv,v,1])  
    return(distance)
```

Complexity

- Setting `infinity` takes $O(n^2)$ time
- Main loop runs `n` times
 - Each iteration visits one more vertex
 - $O(n)$ to find next vertex to visit
 - $O(n)$ to update `distance[v]` for neighbours

```
def dijkstra(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    (visited,distance) = ({},{} )  
    for v in range(rows):  
        (visited[v],distance[v]) = (False,infinity)  
    distance[s] = 0  
    for u in range(rows):  
        nextd = min([distance[v] for v in range(rows)  
                    if not visited[v]])  
        nextvlist = [v for v in range(rows)  
                    if (not visited[v]) and  
                        distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for v in range(cols):  
            if WMat[nextv,v,0] == 1 and (not visited[v]):  
                distance[v] = min(distance[v],distance[nextv]  
                                +WMat[nextv,v,1])  
    return(distance)
```

Complexity

- Setting `infinity` takes $O(n^2)$ time
- Main loop runs `n` times
 - Each iteration visits one more vertex
 - $O(n)$ to find next vertex to visit
 - $O(n)$ to update `distance[v]` for neighbours
- Overall $O(n^2)$

```
def dijkstra(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    (visited,distance) = ({},{} )  
    for v in range(rows):  
        (visited[v],distance[v]) = (False,infinity)  
    distance[s] = 0  
    for u in range(rows):  
        nextd = min([distance[v] for v in range(rows)  
                    if not visited[v]])  
        nextvlist = [v for v in range(rows)  
                    if (not visited[v]) and  
                        distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for v in range(cols):  
            if WMat[nextv,v,0] == 1 and (not visited[v]):  
                distance[v] = min(distance[v],distance[nextv]  
                                +WMat[nextv,v,1])  
    return(distance)
```

Complexity

- Setting `infinity` takes $O(n^2)$ time
- Main loop runs `n` times
 - Each iteration visits one more vertex
 - $O(n)$ to find next vertex to visit
 - $O(n)$ to update `distance[v]` for neighbours
- Overall $O(n^2)$
- If we use an adjacency list
 - Setting `infinity` and updating distances both $O(m)$, amortised
 - $O(n)$ bottleneck remains to find next vertex to visit
 - Better data structure? Later ...

```
def dijkstralist(WList,s):  
    infinity = 1 + len(WList.keys())*  
              max([d for u in WList.keys()  
                    for (v,d) in WList[u]])  
  
(visited,distance) = ({},{} )  
for v in WList.keys():  
    (visited[v],distance[v]) = (False,infinity)  
distance[s] = 0  
for u in WList.keys():  
    nextd = min([distance[v] for v in WList.keys()  
                  if not visited[v]])  
    nextvlist = [v for v in WList.keys()  
                  if (not visited[v]) and  
                      distance[v] == nextd]  
    if nextvlist == []:  
        break  
    nextv = min(nextvlist)  
    visited[nextv] = True  
    for (v,d) in WList[nextv]:  
        if not visited[v]:  
            distance[v] = min(distance[v],distance[nextv]+d)  
return(distance)
```

Summary

- Dijkstra's algorithm computes single source shortest paths
- Use a **greedy** strategy to identify vertices to visit
 - Next vertex to visit is based on shortest distance computed so far
 - Need to prove that such a strategy is correct
 - Correctness requires edge weights to be non-negative
- Complexity is $O(n^2)$
 - Even with adjacency lists
 - Bottleneck is identifying unvisited vertex with minimum distance
 - Need a better data structure to identify and remove minimum (or maximum) from a collection

Single Source Shortest Paths with Negative Weights

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 5

Dijkstra's algorithm

- Recall the burning pipeline analogy

Dijkstra's algorithm

- Recall the burning pipeline analogy
- We keep track of the following
 - The vertices that have been burnt
 - The expected burn time of vertices

Dijkstra's algorithm

- Recall the burning pipeline analogy
- We keep track of the following
 - The vertices that have been burnt
 - The expected burn time of vertices
- Initially
 - No vertex is burnt
 - Expected burn time of source vertex is 0
 - Expected burn time of rest is ∞

Initialization (assume source vertex 0)

- $B(i) = \text{False}$, for $0 \leq i < n$
- $UB = \{k \mid B(k) = \text{False}\}$
- $EBT(i) = \begin{cases} 0, & \text{if } i = 0 \\ \infty, & \text{otherwise} \end{cases}$

Dijkstra's algorithm

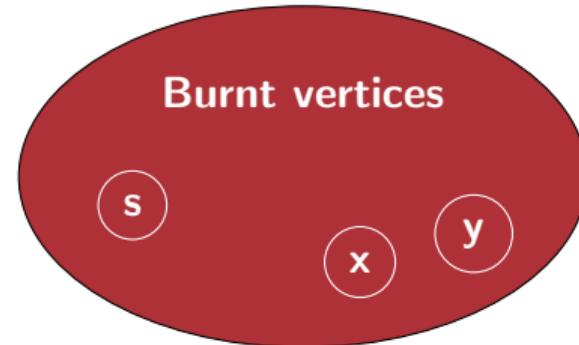
- Recall the burning pipeline analogy
- We keep track of the following
 - The vertices that have been burnt
 - The expected burn time of vertices
- Initially
 - No vertex is burnt
 - Expected burn time of source vertex is 0
 - Expected burn time of rest is ∞
- While there are vertices yet to burn
 - Pick unburnt vertex with minimum expected burn time, mark it as burnt
 - Update the expected burn time of its neighbours

Initialization (assume source vertex 0)

- $B(i) = \text{False}$, for $0 \leq i < n$
 - $UB = \{k \mid B(k) = \text{False}\}$
 - $EBT(i) = \begin{cases} 0, & \text{if } i = 0 \\ \infty, & \text{otherwise} \end{cases}$
- Update, if $UB \neq \emptyset$
- Let $j \in UB$ such that $EBT(j) \leq EBT(k)$ for all $k \in UB$
 - Update $B(j) = \text{True}$, $UB = UB \setminus \{j\}$
 - For each $(j, k) \in E$ such that $k \in UB$,
 $EBT(k) = \min(EBT(k), EBT(j) + W(j, k))$

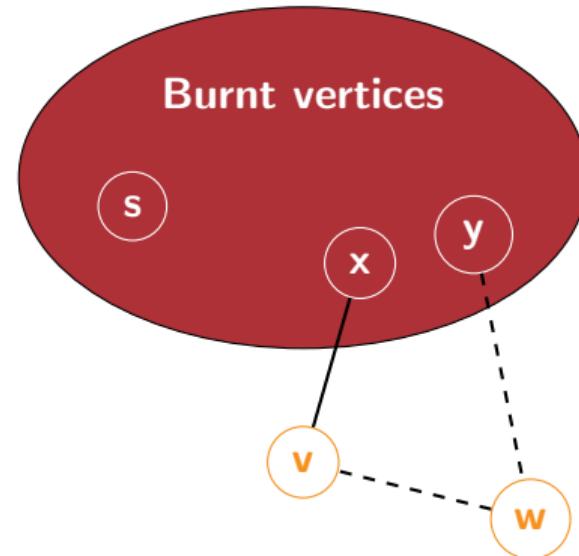
Correctness requires non-negative edge weights

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt



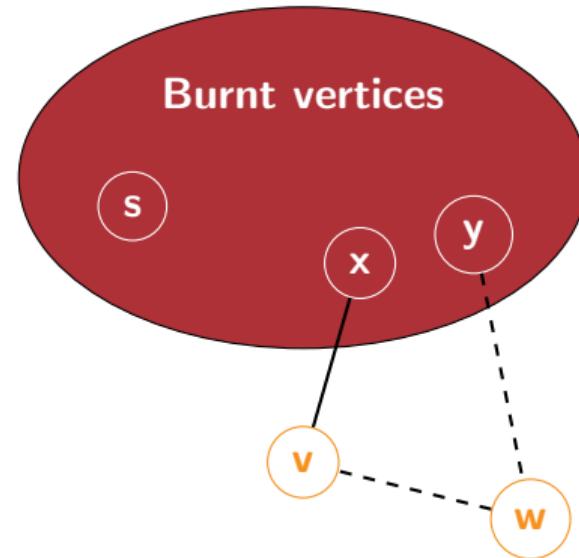
Correctness requires non-negative edge weights

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt
- Next vertex to burn is v , via x
- Cannot find a shorter path later from y to v via w
 - Burn time of $w \geq$ burn time of v
 - Edge from w to v has weight ≥ 0



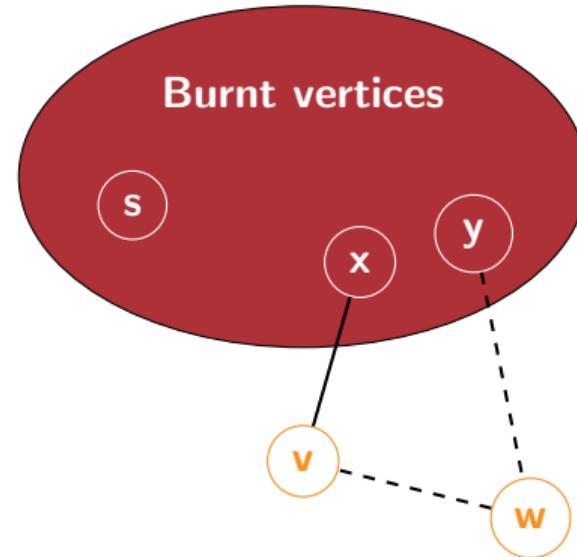
Correctness requires non-negative edge weights

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt
- Next vertex to burn is v , via x
- Cannot find a shorter path later from y to v via w
 - Burn time of $w \geq$ burn time of v
 - Edge from w to v has weight ≥ 0
- This argument breaks down if edge (w,v) can have negative weight



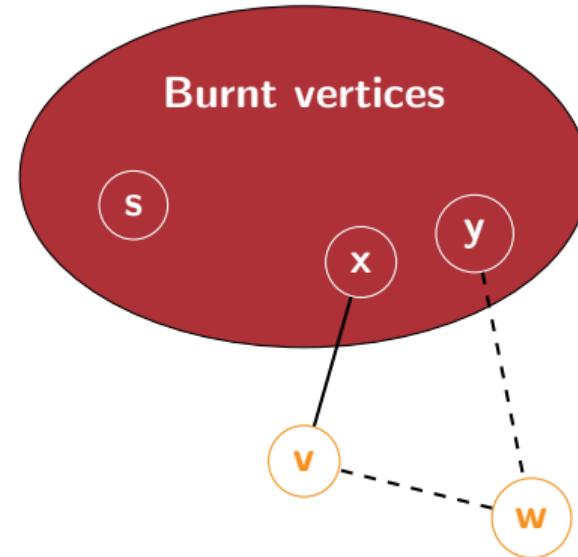
Extending to negative edge weights

- The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt



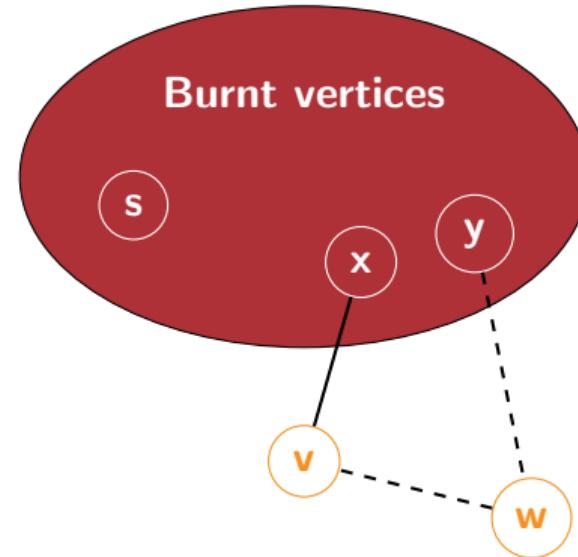
Extending to negative edge weights

- The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt
- What if we allow updates even after a vertex is burnt?



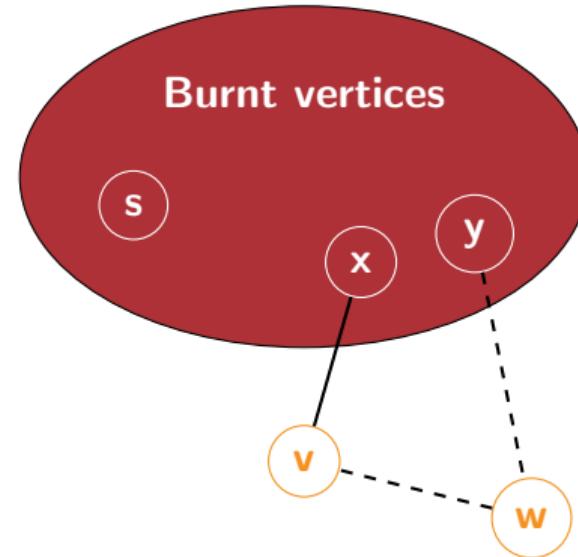
Extending to negative edge weights

- The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt
- What if we allow updates even after a vertex is burnt?
- Recall, negative edge weights are allowed, but no negative cycles



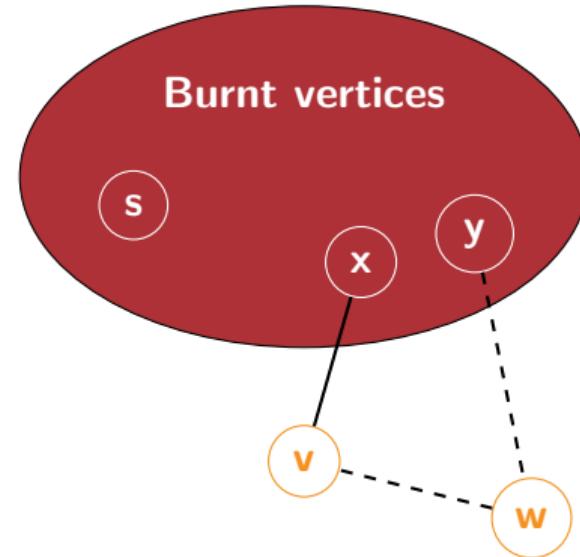
Extending to negative edge weights

- The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt
- What if we allow updates even after a vertex is burnt?
- Recall, negative edge weights are allowed, but no negative cycles
- Going around a cycle can only add to the length



Extending to negative edge weights

- The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt
- What if we allow updates even after a vertex is burnt?
- Recall, negative edge weights are allowed, but no negative cycles
- Going around a cycle can only add to the length
- Shortest route to every vertex is a path, no loops



Extending to negative edge weights

- Suppose minimum weight path from 0 to k is

$$0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{\ell-1}} j_{\ell-1} \xrightarrow{w_\ell} k$$

- Need not be minimum in terms of number of edges

Extending to negative edge weights

- Suppose minimum weight path from 0 to k is

$$0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{\ell-1}} j_{\ell-1} \xrightarrow{w_\ell} k$$

- Need not be minimum in terms of number of edges
- Every prefix of this path must itself be a minimum weight path

- 0 $\xrightarrow{w_1} j_1$
- 0 $\xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2$
- ...
- 0 $\xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{\ell-1}} j_{\ell-1}$

Extending to negative edge weights

- Suppose minimum weight path from 0 to k is

$$0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{\ell-1}} j_{\ell-1} \xrightarrow{w_\ell} k$$

- Need not be minimum in terms of number of edges
- Every prefix of this path must itself be a minimum weight path
 - $0 \xrightarrow{w_1} j_1$
 - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2$
 - \dots
 - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{\ell-1}} j_{\ell-1}$

- Once we discover shortest path to $j_{\ell-1}$, next update will fix shortest path to k

Extending to negative edge weights

- Suppose minimum weight path from 0 to k is

$$0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{\ell-1}} j_{\ell-1} \xrightarrow{w_\ell} k$$

- Need not be minimum in terms of number of edges
- Every prefix of this path must itself be a minimum weight path
 - $0 \xrightarrow{w_1} j_1$
 - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2$
 - \dots
 - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{\ell-1}} j_{\ell-1}$

- Once we discover shortest path to $j_{\ell-1}$, next update will fix shortest path to k
- Repeatedly update shortest distance to each vertex based on shortest distance to its neighbours
- Update cannot push this distance below actual shortest distance

Extending to negative edge weights

- Suppose minimum weight path from 0 to k is
$$0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{\ell-1}} j_{\ell-1} \xrightarrow{w_\ell} k$$
 - Need not be minimum in terms of number of edges
- Every prefix of this path must itself be a minimum weight path
 - $0 \xrightarrow{w_1} j_1$
 - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2$
 - \dots
 - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{\ell-1}} j_{\ell-1}$

- Once we discover shortest path to $j_{\ell-1}$, next update will fix shortest path to k
- Repeatedly update shortest distance to each vertex based on shortest distance to its neighbours
 - Update cannot push this distance below actual shortest distance
- After ℓ updates, all shortest paths using $\leq \ell$ edges have stabilized
 - Minimum weight path to any node has at most $n-1$ edges
 - After $n-1$ updates, all shortest paths have stabilized

Bellman-Ford Algorithm

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

Bellman-Ford Algorithm

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

Repeat $n-1$ times

- For each vertex $j \in \{0, 1, \dots, n-1\}$,
for each edge $(j, k) \in E$,
$$D(k) = \min(D(k), D(j) + W(j, k))$$

Bellman-Ford Algorithm

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

Repeat $n-1$ times

- For each vertex $j \in \{0, 1, \dots, n-1\}$,
for each edge $(j, k) \in E$,
$$D(k) = \min(D(k), D(j) + W(j, k))$$

Works for directed and undirected graphs

Bellman-Ford Algorithm

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

Repeat $n-1$ times

- For each vertex $j \in \{0, 1, \dots, n-1\}$,
for each edge $(j, k) \in E$,
$$D(k) = \min(D(k), D(j) + W(j, k))$$

Works for directed and undirected graphs

```
def bellmanford(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    distance = {}  
    for v in range(rows):  
        distance[v] = infinity  
  
    distance[s] = 0  
  
    for i in range(rows):  
        for u in range(rows):  
            for v in range(cols):  
                if WMat[u,v,0] == 1:  
                    distance[v] = min(distance[v],distance[u]  
                                     +WMat[u,v,1])  
    return(distance)
```

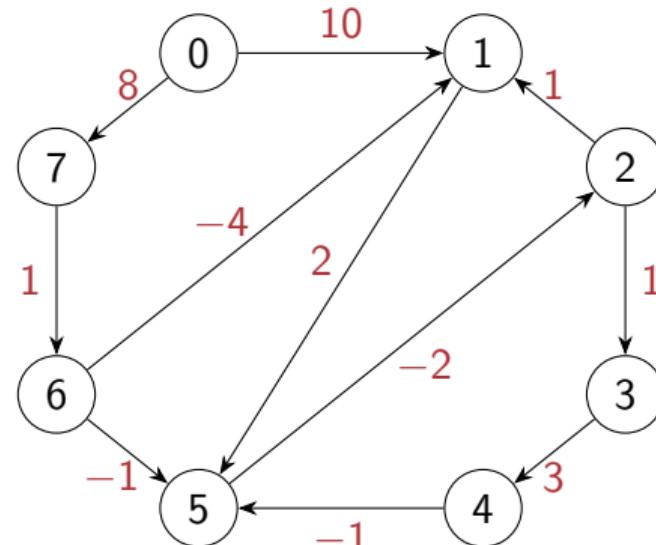
Bellman-Ford Algorithm

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

Repeat $n-1$ times

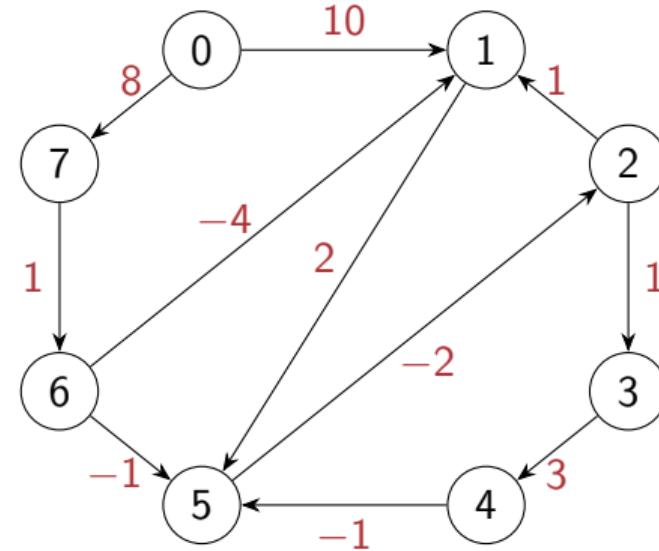
- For each vertex $j \in \{0, 1, \dots, n-1\}$,
for each edge $(j, k) \in E$,
$$D(k) = \min(D(k), D(j) + W(j, k))$$



Works for directed and undirected graphs

Bellman-Ford Algorithm

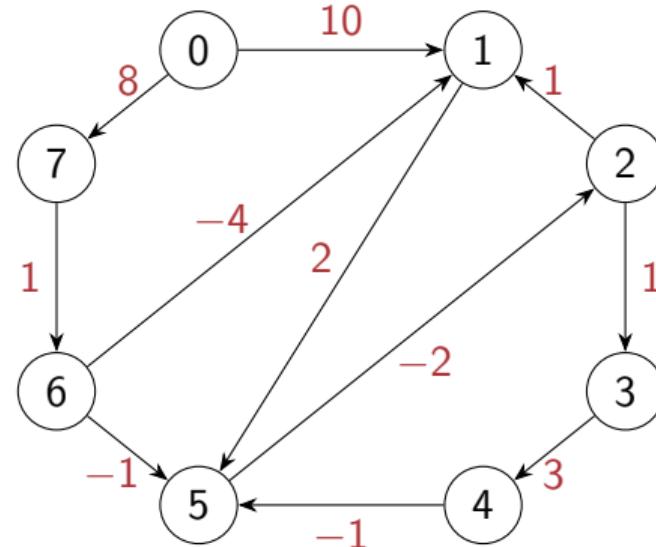
v	$D(v)$
0	
1	
2	
3	
4	
5	
6	
7	



Bellman-Ford Algorithm

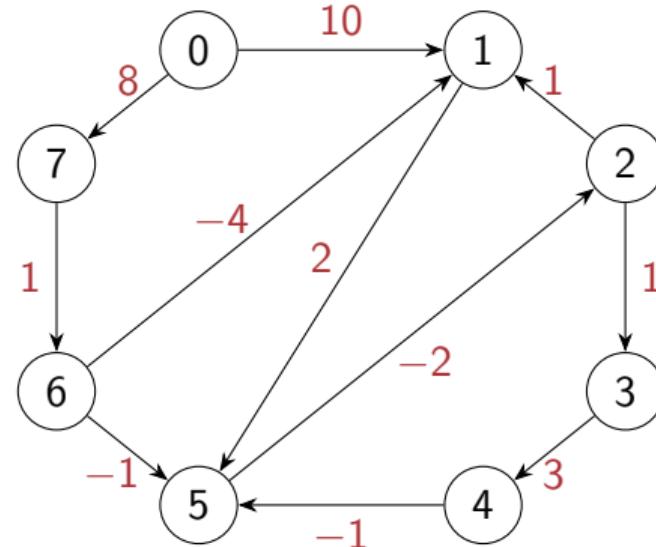
v	$D(v)$
0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

- Initialize $D(0) = 0$



Bellman-Ford Algorithm

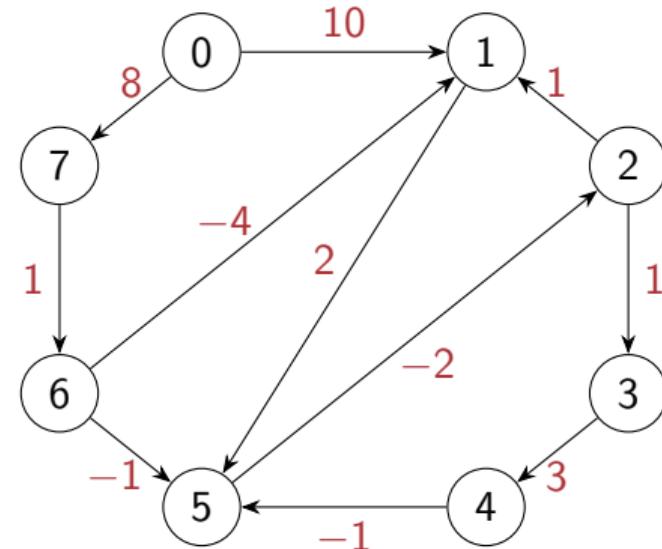
v	$D(v)$	
0	0	0
1	∞	10
2	∞	∞
3	∞	∞
4	∞	∞
5	∞	∞
6	∞	∞
7	∞	8



- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update
$$D(k) = \min(D(k), D(j) + W(j, k))$$

Bellman-Ford Algorithm

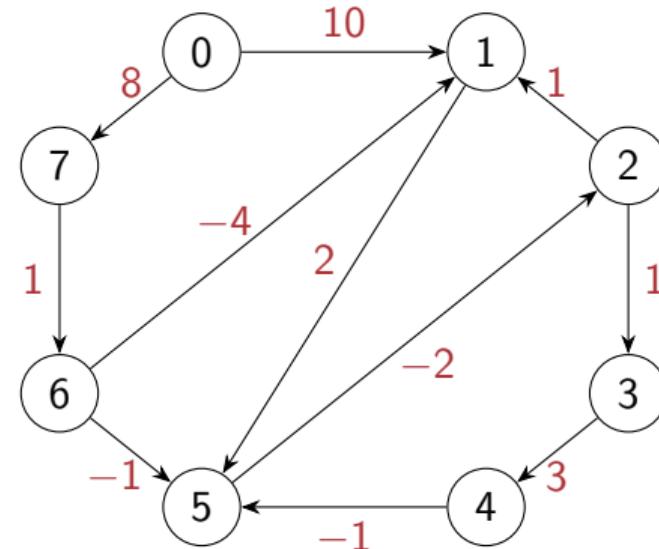
v	$D(v)$		
0	0	0	0
1	∞	10	10
2	∞	∞	∞
3	∞	∞	∞
4	∞	∞	∞
5	∞	∞	12
6	∞	∞	9
7	∞	8	8



- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update
$$D(k) = \min(D(k), D(j) + W(j, k))$$

Bellman-Ford Algorithm

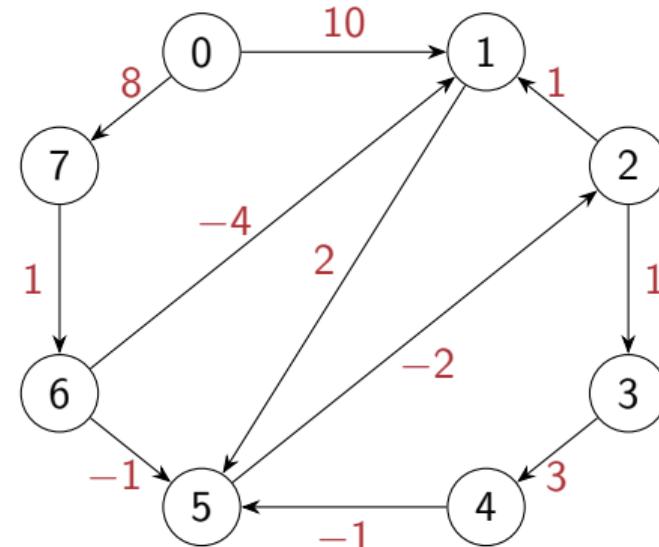
v	$D(v)$			
0	0	0	0	0
1	∞	10	10	5
2	∞	∞	∞	10
3	∞	∞	∞	∞
4	∞	∞	∞	∞
5	∞	∞	12	8
6	∞	∞	9	9
7	∞	8	8	8



- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update
$$D(k) = \min(D(k), D(j) + W(j, k))$$

Bellman-Ford Algorithm

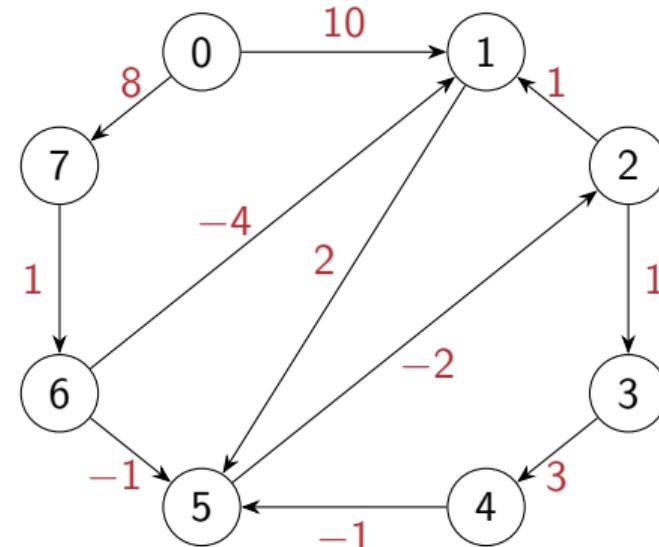
v	$D(v)$				
0	0	0	0	0	0
1	∞	10	10	5	5
2	∞	∞	∞	10	6
3	∞	∞	∞	∞	11
4	∞	∞	∞	∞	∞
5	∞	∞	12	8	7
6	∞	∞	9	9	9
7	∞	8	8	8	8



- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update
$$D(k) = \min(D(k), D(j) + W(j, k))$$

Bellman-Ford Algorithm

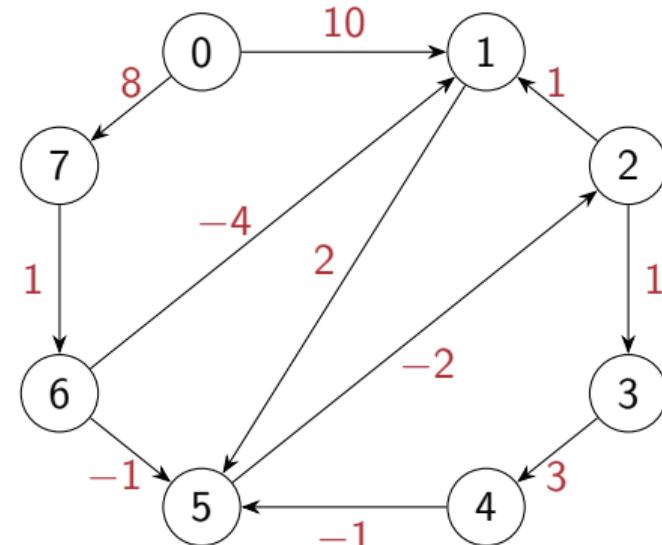
v	$D(v)$					
0	0	0	0	0	0	
1	∞	10	10	5	5	5
2	∞	∞	∞	10	6	5
3	∞	∞	∞	∞	11	7
4	∞	∞	∞	∞	∞	14
5	∞	∞	12	8	7	7
6	∞	∞	9	9	9	9
7	∞	8	8	8	8	



- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update
$$D(k) = \min(D(k), D(j) + W(j, k))$$

Bellman-Ford Algorithm

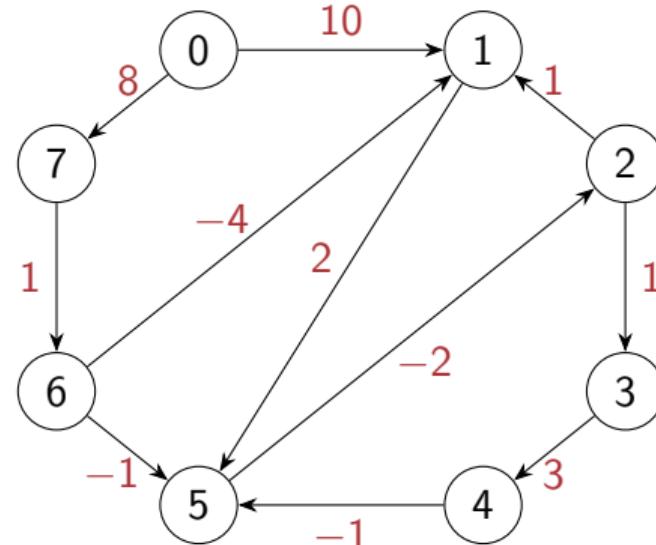
v	$D(v)$						
0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5
2	∞	∞	∞	10	6	5	5
3	∞	∞	∞	∞	11	7	6
4	∞	∞	∞	∞	∞	14	10
5	∞	∞	12	8	7	7	7
6	∞	∞	9	9	9	9	9
7	∞	8	8	8	8	8	8



- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update
$$D(k) = \min(D(k), D(j) + W(j, k))$$

Bellman-Ford Algorithm

v	$D(v)$							
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8

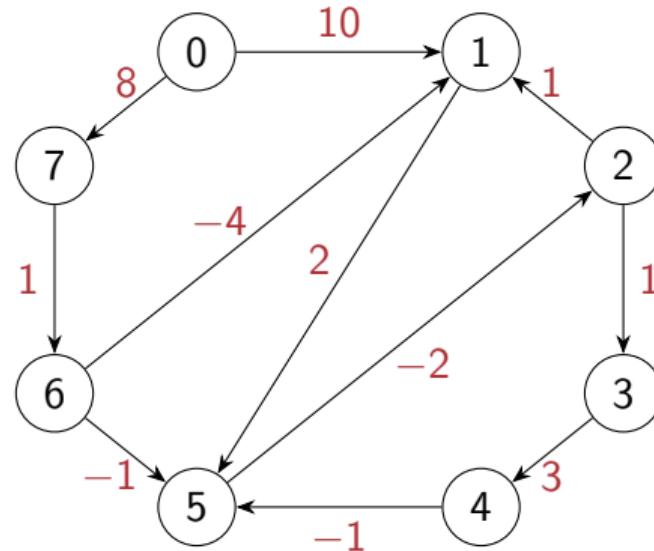


- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update
$$D(k) = \min(D(k), D(j) + W(j, k))$$

Bellman-Ford Algorithm

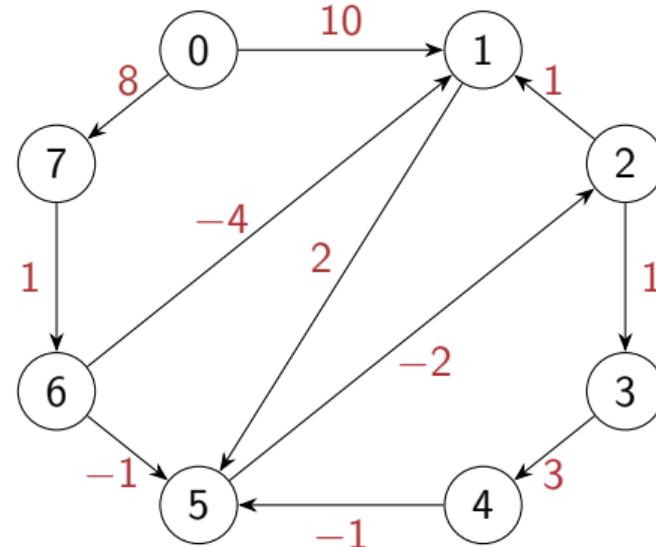
v	$D(v)$							
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8

- What if there was a negative cycle?



Bellman-Ford Algorithm

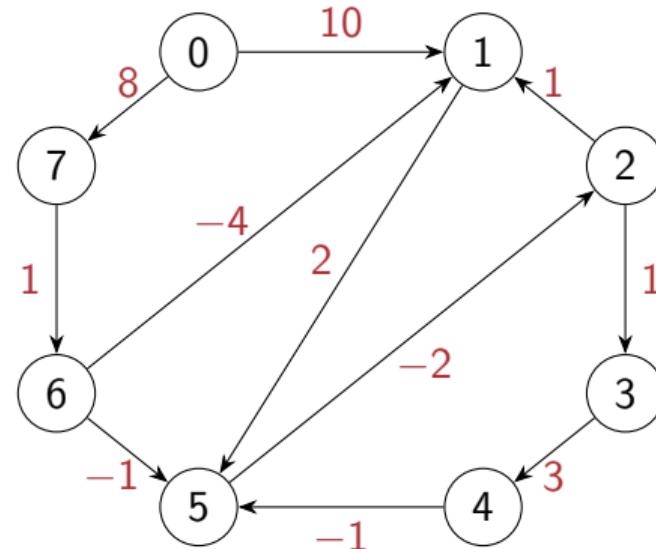
v	$D(v)$							
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8



- What if there was a negative cycle?
- Distance would continue to decrease

Bellman-Ford Algorithm

v	$D(v)$							
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8



- What if there was a negative cycle?
- Distance would continue to decrease
- Check if update n reduces any $D(v)$

Complexity

- Initialing `infinity` takes $O(n^2)$ time

```
def bellmanford(WMat,s):
    (rows,cols,x) = WMat.shape
    infinity = np.max(WMat)*rows+1
    distance = {}
    for v in range(rows):
        distance[v] = infinity

    distance[s] = 0

    for i in range(rows):
        for u in range(rows):
            for v in range(cols):
                if WMat[u,v,0] == 1:
                    distance[v] = min(distance[v],distance[u]
                                       +WMat[u,v,1])
    return(distance)
```

Complexity

- Initializing `infinity` takes $O(n^2)$ time
- The outer update loop runs $O(n)$ times

```
def bellmanford(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    distance = {}  
    for v in range(rows):  
        distance[v] = infinity  
  
    distance[s] = 0  
  
    for i in range(rows):  
        for u in range(rows):  
            for v in range(cols):  
                if WMat[u,v,0] == 1:  
                    distance[v] = min(distance[v],distance[u]  
                                      +WMat[u,v,1])  
    return(distance)
```

Complexity

- Initializing `infinity` takes $O(n^2)$ time
- The outer update loop runs $O(n)$ times
- In each iteration, we have to examine every edge in the graph
 - This take $O(n^2)$ for an adjacency matrix

```
def bellmanford(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    distance = {}  
    for v in range(rows):  
        distance[v] = infinity  
  
    distance[s] = 0  
  
    for i in range(rows):  
        for u in range(rows):  
            for v in range(cols):  
                if WMat[u,v,0] == 1:  
                    distance[v] = min(distance[v],distance[u]  
                                      +WMat[u,v,1])  
  
    return(distance)
```

Complexity

- Initializing `infinity` takes $O(n^2)$ time
- The outer update loop runs $O(n)$ times
- In each iteration, we have to examine every edge in the graph
 - This take $O(n^2)$ for an adjacency matrix
- Overall, $O(n^3)$

```
def bellmanford(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    distance = {}  
    for v in range(rows):  
        distance[v] = infinity  
  
    distance[s] = 0  
  
    for i in range(rows):  
        for u in range(rows):  
            for v in range(cols):  
                if WMat[u,v,0] == 1:  
                    distance[v] = min(distance[v],distance[u]  
                                      +WMat[u,v,1])  
  
    return(distance)
```

Complexity

- Initializing `infinity` takes $O(n^2)$ time
- The outer update loop runs $O(n)$ times
- In each iteration, we have to examine every edge in the graph
 - This take $O(n^2)$ for an adjacency matrix
- Overall, $O(n^3)$
- If we shift to adjacency lists
 - Initializing `infinity` is $O(m)$
 - Scanning all edges in each update iteration is $O(m)$

```
def bellmanfordlist(WList,s):  
    infinity = 1 + len(WList.keys())*  
               max([d for u in WList.keys()  
                     for (v,d) in WList[u]])  
  
    distance = {}  
    for v in WList.keys():  
        distance[v] = infinity  
  
    distance[s] = 0  
  
    for i in WList.keys():  
        for u in WList.keys():  
            for (v,d) in WList[u]:  
                distance[v] = min(distance[v],distance[u] + d)  
    return(distance)
```

Complexity

- Initializing `infinity` takes $O(n^2)$ time
- The outer update loop runs $O(n)$ times
- In each iteration, we have to examine every edge in the graph
 - This take $O(n^2)$ for an adjacency matrix
- Overall, $O(n^3)$
- If we shift to adjacency lists
 - Initializing `infinity` is $O(m)$
 - Scanning all edges in each update iteration is $O(m)$
- Now, overall $O(mn)$

```
def bellmanfordlist(WList,s):  
    infinity = 1 + len(WList.keys())*  
              max([d for u in WList.keys()  
                    for (v,d) in WList[u]])  
  
    distance = {}  
    for v in WList.keys():  
        distance[v] = infinity  
  
    distance[s] = 0  
  
    for i in WList.keys():  
        for u in WList.keys():  
            for (v,d) in WList[u]:  
                distance[v] = min(distance[v],distance[u] + d)  
    return(distance)
```

Summary

- Dijkstra's algorithm assumes non-negative edge weights
 - Final distance is frozen each time a vertex "burns"
 - Should not encounter a shorter route discovered later
- Without negative cycles, every shortest route is a path
- Every prefix of a shortest path is also a shortest path
- Iteratively find shortest paths of length $1, 2, \dots, n-1$
- Update distance to each vertex with every iteration — **Bellman-Ford algorithm**
- $O(n^3)$ time with adjacency matrix, $O(mn)$ time with adjacency list
- If Bellman-Ford algorithm does not converge after $n - 1$ iterations, there is a negative cycle

All-Pairs Shortest Paths

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 5

Shortest paths in weighted graphs

Two types of shortest path problems of interest

Single source shortest paths

- Find shortest paths from a fixed vertex to every other vertex
- Transport finished product from factory (single source) to all retail outlets
- Courier company delivers items from distribution centre (single source) to addressees

All pairs shortest paths

- Find shortest paths between every pair of vertices i and j
- Optimal airline, railway, road routes between cities

Shortest paths in weighted graphs

Two types of shortest path problems of interest

Single source shortest paths

- Find shortest paths from a fixed vertex to every other vertex
- Transport finished product from factory (single source) to all retail outlets
- Courier company delivers items from distribution centre (single source) to addressees
- Dijkstra's algorithm (non-negative weights), Bellman-Ford algorithm (allows negative weights)

All pairs shortest paths

- Find shortest paths between every pair of vertices i and j
- Optimal airline, railway, road routes between cities

Shortest paths in weighted graphs

Two types of shortest path problems of interest

Single source shortest paths

- Find shortest paths from a fixed vertex to every other vertex
- Transport finished product from factory (single source) to all retail outlets
- Courier company delivers items from distribution centre (single source) to addressees
- Dijkstra's algorithm (non-negative weights), Bellman-Ford algorithm (allows negative weights)

All pairs shortest paths

- Find shortest paths between every pair of vertices i and j
- Optimal airline, railway, road routes between cities
- Run Dijkstra or Bellman-Ford from each vertex

Shortest paths in weighted graphs

Two types of shortest path problems of interest

Single source shortest paths

- Find shortest paths from a fixed vertex to every other vertex
- Transport finished product from factory (single source) to all retail outlets
- Courier company delivers items from distribution centre (single source) to addressees
- Dijkstra's algorithm (non-negative weights), Bellman-Ford algorithm (allows negative weights)

All pairs shortest paths

- Find shortest paths between every pair of vertices i and j
- Optimal airline, railway, road routes between cities
- Run Dijkstra or Bellman-Ford from each vertex
- Is there is another way?

Transitive closure

- Adjacency matrix A represents paths of length 1
- Matrix multiplication, $A^2 = A \times A$
 - $A^2[i, j] = 1$ if there is a path of length 2 from i to j
 - For some k , $A[i, k] = A[k, j] = 1$
- In general, $A^{\ell+1} = A^\ell \times A$,
 - $A^{\ell+1}[i, j] = 1$ if there is a path of length $\ell+1$ from i to j
 - For some k , $A^\ell[i, k] = 1$, $A[k, j] = 1$
- $A^+ = A + A^2 + \cdots + A^{n-1}$

Transitive closure

- Adjacency matrix A represents paths of length 1
- Matrix multiplication, $A^2 = A \times A$
 - $A^2[i, j] = 1$ if there is a path of length 2 from i to j
 - For some k , $A[i, k] = A[k, j] = 1$
- In general, $A^{\ell+1} = A^\ell \times A$,
 - $A^{\ell+1}[i, j] = 1$ if there is a path of length $\ell+1$ from i to j
 - For some k , $A^\ell[i, k] = 1$, $A[k, j] = 1$
- $A^+ = A + A^2 + \cdots + A^{n-1}$

An alternative approach

Transitive closure

- Adjacency matrix A represents paths of length 1
- Matrix multiplication, $A^2 = A \times A$
 - $A^2[i,j] = 1$ if there is a path of length 2 from i to j
 - For some k , $A[i,k] = A[k,j] = 1$
- In general, $A^{\ell+1} = A^\ell \times A$,
 - $A^{\ell+1}[i,j] = 1$ if there is a path of length $\ell+1$ from i to j
 - For some k , $A^\ell[i,k] = 1$, $A[k,j] = 1$
- $A^+ = A + A^2 + \cdots + A^{n-1}$

An alternative approach

- $B^k[i,j] = 1$ if there is path from i to j via vertices $\{0, 1, \dots, k-1\}$
- Constraint applies only to intermediate vertices between i and j
- $B^0[i,j] = 1$ if there is a direct edge
- $B^0 = A$

Transitive closure

- Adjacency matrix A represents paths of length 1
- Matrix multiplication, $A^2 = A \times A$
 - $A^2[i,j] = 1$ if there is a path of length 2 from i to j
 - For some k , $A[i,k] = A[k,j] = 1$
- In general, $A^{\ell+1} = A^\ell \times A$,
 - $A^{\ell+1}[i,j] = 1$ if there is a path of length $\ell+1$ from i to j
 - For some k , $A^\ell[i,k] = 1$, $A[k,j] = 1$
- $A^+ = A + A^2 + \cdots + A^{n-1}$

An alternative approach

- $B^k[i,j] = 1$ if there is path from i to j via vertices $\{0, 1, \dots, k-1\}$
 - Constraint applies only to intermediate vertices between i and j
 - $B^0[i,j] = 1$ if there is a direct edge
 - $B^0 = A$
- $B^{k+1}[i,j] = 1$ if
 - $B^k[i,j] = 1$ — can already reach j from i via $\{0, 1, \dots, k-1\}$
 - $B^k[i,k] = 1$ and $B^k[k,j] = 1$ — use $\{0, 1, \dots, k-1\}$ to go from i to k and then from k to j

Warshall's Algorithm

- $B^k[i,j] = 1$ if there is path from i to j via vertices $\{0,1,\dots,k-1\}$
 - $B^0[i,j] = A[i,j]$
 - Direct edges, no intermediate vertices
 - $B^{k+1}[i,j] = 1$ if
 - $B^k[i,j] = 1$, or
 - $B^k[i,k] = 1$ and $B^k[k,j] = 1$
- The algorithm on the left also computes transitive closure — Warshall's algorithm

Warshall's Algorithm

- $B^k[i, j] = 1$ if there is path from i to j via vertices $\{0, 1, \dots, k-1\}$
- $B^0[i, j] = A[i, j]$
 - Direct edges, no intermediate vertices
- $B^{k+1}[i, j] = 1$ if
 - $B^k[i, j] = 1$, or
 - $B^k[i, k] = 1$ and $B^k[k, j] = 1$
- The algorithm on the left also computes transitive closure — **Warshall's algorithm**
- $B^n[i, j] = 1$ if there is some path from i to j with intermediate vertices in $\{0, 1, \dots, n-1\}$

Warshall's Algorithm

- $B^k[i, j] = 1$ if there is path from i to j via vertices $\{0, 1, \dots, k-1\}$
- $B^0[i, j] = A[i, j]$
 - Direct edges, no intermediate vertices
- $B^{k+1}[i, j] = 1$ if
 - $B^k[i, j] = 1$, or
 - $B^k[i, k] = 1$ and $B^k[k, j] = 1$
- The algorithm on the left also computes transitive closure — **Warshall's algorithm**
- $B^n[i, j] = 1$ if there is some path from i to j with intermediate vertices in $\{0, 1, \dots, n-1\}$
- $B^n = A^+$

Warshall's Algorithm

- $B^k[i, j] = 1$ if there is path from i to j via vertices $\{0, 1, \dots, k-1\}$
 - $B^0[i, j] = A[i, j]$
 - Direct edges, no intermediate vertices
 - $B^{k+1}[i, j] = 1$ if
 - $B^k[i, j] = 1$, or
 - $B^k[i, k] = 1$ and $B^k[k, j] = 1$
-
- The algorithm on the left also computes transitive closure — **Warshall's algorithm**
 - $B^n[i, j] = 1$ if there is some path from i to j with intermediate vertices in $\{0, 1, \dots, n-1\}$
 - $B^n = A^+$
 - We adapt Warshall's algorithm to compute all-pairs shortest paths

Floyd-Warshall Algorithm

- Let $SP^k[i, j]$ be the length of the shortest path from i to j via vertices $\{0, 1, \dots, k-1\}$
- $SP^0[i, j] = W[i, j]$
 - No intermediate vertices, shortest path is weight of direct edge
 - Assume $W[i, j] = \infty$ if $(i, j) \notin E$

Floyd-Warshall Algorithm

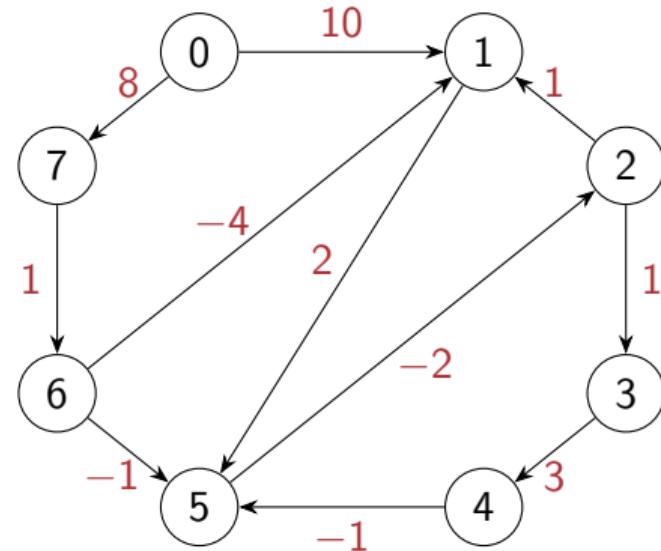
- Let $SP^k[i, j]$ be the length of the shortest path from i to j via vertices $\{0, 1, \dots, k-1\}$
- $SP^0[i, j] = W[i, j]$
 - No intermediate vertices, shortest path is weight of direct edge
 - Assume $W[i, j] = \infty$ if $(i, j) \notin E$
- $SP^{k+1}[i, j]$ is the minimum of
 - $SP^k[i, j]$
Shortest path using only $\{0, 1, \dots, k-1\}$
 - $SP^k[i, k] + SP^k[k, j]$
Combine shortest path from i to k and k to j

Floyd-Warshall Algorithm

- Let $SP^k[i, j]$ be the length of the shortest path from i to j via vertices $\{0, 1, \dots, k-1\}$
- $SP^0[i, j] = W[i, j]$
 - No intermediate vertices, shortest path is weight of direct edge
 - Assume $W[i, j] = \infty$ if $(i, j) \notin E$
- $SP^{k+1}[i, j]$ is the minimum of
 - $SP^k[i, j]$
Shortest path using only $\{0, 1, \dots, k-1\}$
 - $SP^k[i, k] + SP^k[k, j]$
Combine shortest path from i to k and k to j
- $SP^n[i, j] = 1$ is the length of the shortest path overall from i to j
 - Intermediate vertices lie in $\{0, 1, \dots, n-1\}$

Floyd-Warshall Algorithm

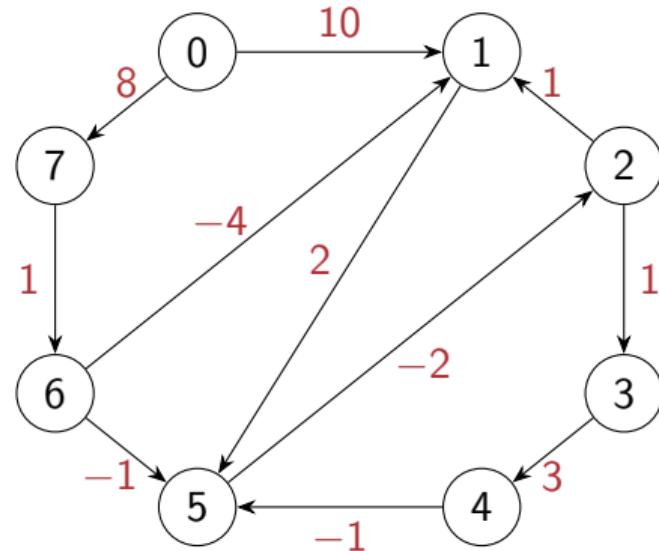
SP^0	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	∞	∞	8
1	∞	∞	∞	∞	∞	2	∞	∞
2	∞	1	∞	1	∞	∞	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	∞	-2	∞	∞	∞	∞	∞
6	∞	-4	∞	∞	∞	-1	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞



Floyd-Warshall Algorithm

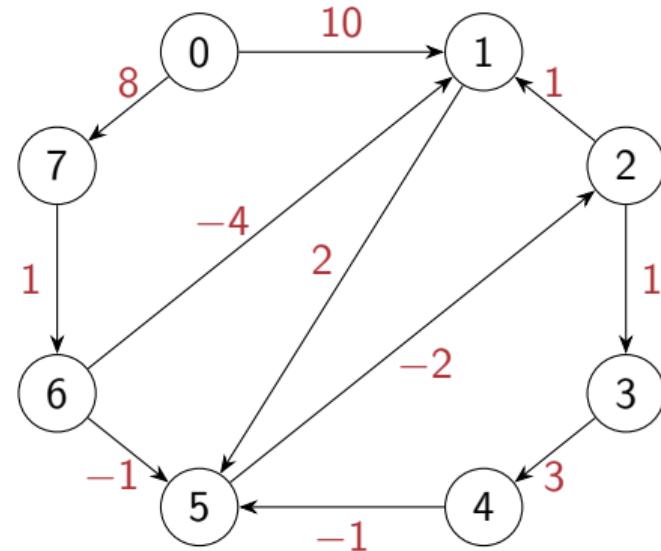
SP^0	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	∞	∞	8
1	∞	∞	∞	∞	∞	2	∞	∞
2	∞	1	∞	1	∞	∞	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	∞	-2	∞	∞	∞	∞	∞
6	∞	-4	∞	∞	∞	-1	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞

SP^1	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	∞	∞	8
1	∞	∞	∞	∞	∞	2	∞	∞
2	∞	1	∞	1	∞	∞	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	∞	-2	∞	∞	∞	∞	∞
6	∞	-4	∞	∞	∞	-1	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞



Floyd-Warshall Algorithm

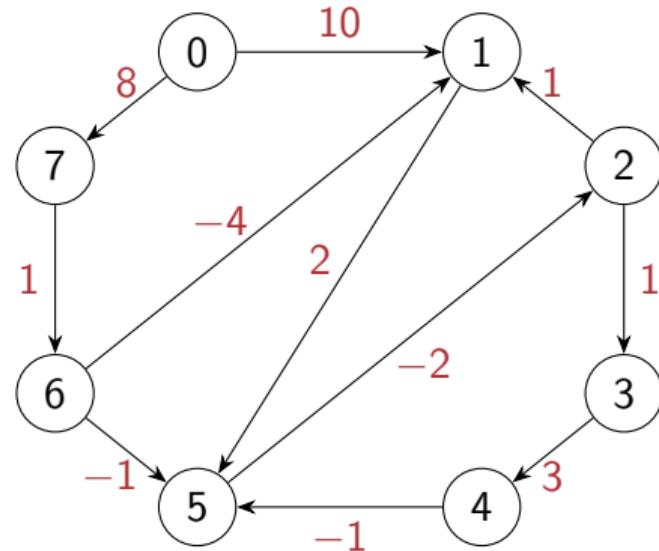
SP^1	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	∞	∞	8
1	∞	∞	∞	∞	2	∞	∞	∞
2	∞	1	∞	1	∞	∞	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	∞	-2	∞	∞	∞	∞	∞
6	∞	-4	∞	∞	∞	-1	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞



Floyd-Warshall Algorithm

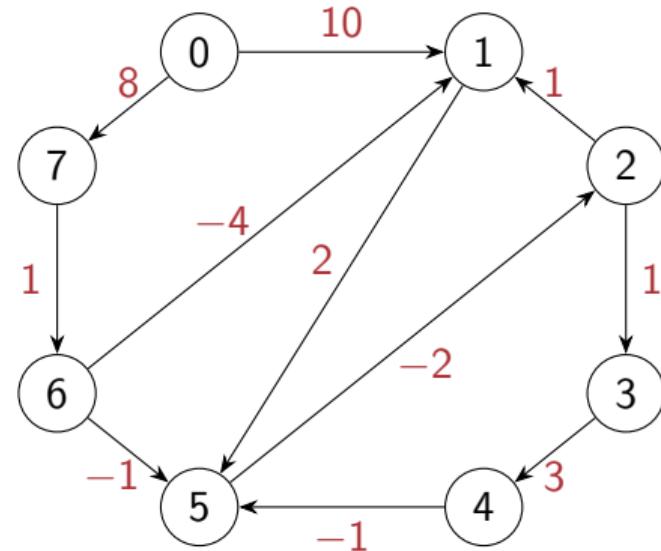
SP^1	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	∞	∞	8
1	∞	∞	∞	∞	∞	2	∞	∞
2	∞	1	∞	1	∞	∞	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	∞	-2	∞	∞	∞	∞	∞
6	∞	-4	∞	∞	∞	-1	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞

SP^2	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	12	∞	8
1	∞	∞	∞	∞	∞	2	∞	∞
2	∞	1	∞	1	∞	3	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	∞	-2	∞	∞	∞	∞	∞
6	∞	-4	∞	∞	∞	-2	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞



Floyd-Warshall Algorithm

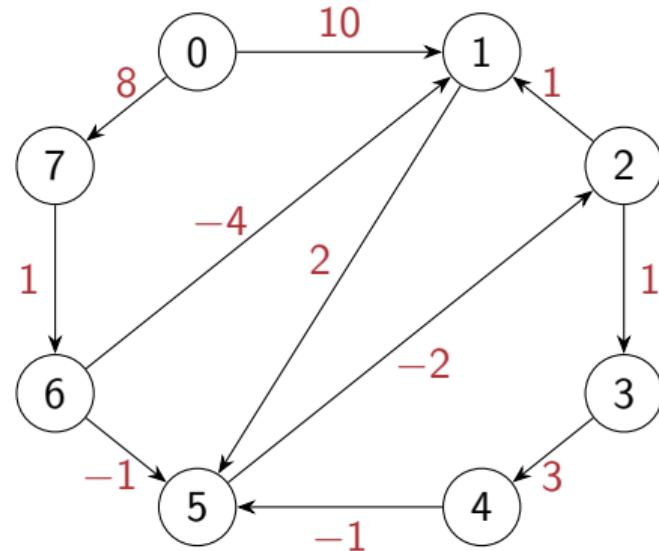
SP^2	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	12	∞	8
1	∞	∞	∞	∞	2	∞	∞	
2	∞	1	∞	1	∞	3	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	∞	-2	∞	∞	∞	∞	∞
6	∞	-4	∞	∞	∞	-2	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞



Floyd-Warshall Algorithm

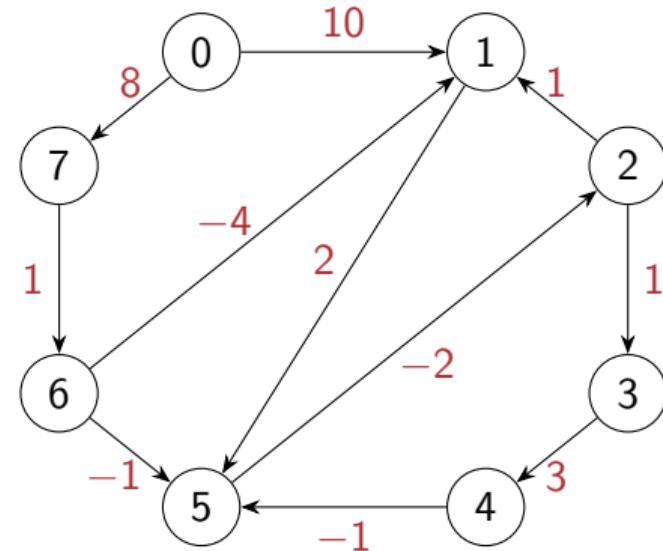
SP^2	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	12	∞	8
1	∞	∞	∞	∞	∞	2	∞	∞
2	∞	1	∞	1	∞	3	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	∞	-2	∞	∞	∞	∞	∞
6	∞	-4	∞	∞	∞	-2	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞

SP^3	0	1	2	3	4	5	6	7
0	∞	10	∞	∞	∞	12	∞	8
1	∞	∞	∞	∞	∞	2	∞	∞
2	∞	1	∞	1	∞	3	∞	∞
3	∞	∞	∞	∞	3	∞	∞	∞
4	∞	∞	∞	∞	∞	-1	∞	∞
5	∞	-1	-2	-1	∞	1	∞	∞
6	∞	-4	∞	∞	∞	-2	∞	∞
7	∞	∞	∞	∞	∞	∞	1	∞



Floyd-Warshall Algorithm

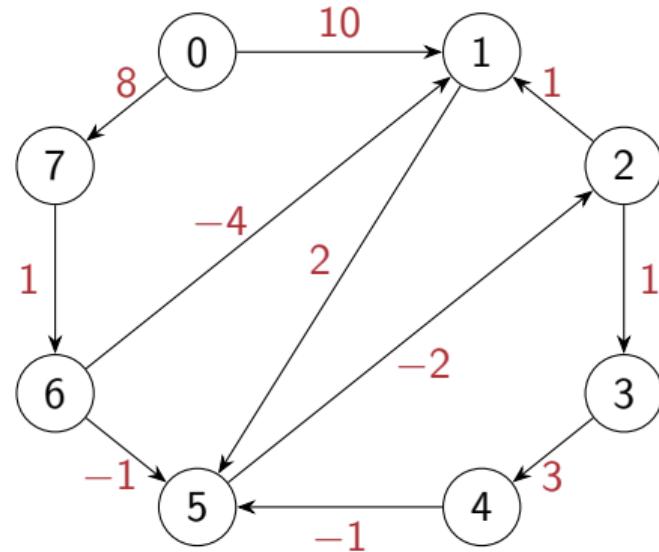
SP^T	0	1	2	3	4	5	6	7
0	∞	10	10	11	14	12	∞	8
1	∞	1	0	1	4	2	∞	∞
2	∞	1	1	1	4	3	∞	∞
3	∞	1	0	1	3	2	∞	∞
4	∞	-2	-3	-2	1	-1	∞	∞
5	∞	-1	-2	-1	2	1	∞	∞
6	∞	-4	-4	-3	0	-2	∞	∞
7	∞	-3	-3	-2	1	-1	1	∞



Floyd-Warshall Algorithm

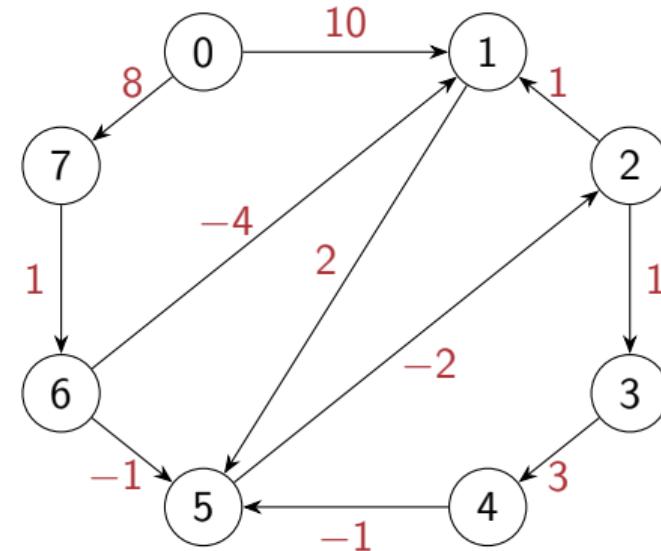
SP'	0	1	2	3	4	5	6	7
0	∞	10	10	11	14	12	∞	8
1	∞	1	0	1	4	2	∞	∞
2	∞	1	1	1	4	3	∞	∞
3	∞	1	0	1	3	2	∞	∞
4	∞	-2	-3	-2	1	-1	∞	∞
5	∞	-1	-2	-1	2	1	∞	∞
6	∞	-4	-4	-3	0	-2	∞	∞
7	∞	-3	-3	-2	1	-1	1	∞

SP^8	0	1	2	3	4	5	6	7
0	∞	5	5	6	9	7	9	8
1	∞	1	0	1	4	2	∞	∞
2	∞	1	1	1	4	3	∞	∞
3	∞	1	0	1	3	2	∞	∞
4	∞	-2	-3	-2	1	-1	∞	∞
5	∞	-1	-2	-1	2	1	∞	∞
6	∞	-4	-4	-3	0	-2	∞	∞
7	∞	-3	-3	-2	1	-1	1	∞



Floyd-Warshall Algorithm

SP^8	0	1	2	3	4	5	6	7
0	∞	5	5	6	9	7	9	8
1	∞	1	0	1	4	2	∞	∞
2	∞	1	1	1	4	3	∞	∞
3	∞	1	0	1	3	2	∞	∞
4	∞	-2	-3	-2	1	-1	∞	∞
5	∞	-1	-2	-1	2	1	∞	∞
6	∞	-4	-4	-3	0	-2	∞	∞
7	∞	-3	-3	-2	1	-1	1	∞



Implementation

- Shortest path matrix SP is
 $n \times n \times (n + 1)$

```
def floydwarshall(WMat):
    (rows,cols,x) = WMat.shape
    infinity = np.max(WMat)*rows*rows+1
    SP = np.zeros(shape=(rows,cols,cols+1))
    for i in range(rows):
        for j in range(cols):
            SP[i,j,0] = infinity

    for i in range(rows):
        for j in range(cols):
            if WMat[i,j,0] == 1:
                SP[i,j,0] = WMat[i,j,1]

    for k in range(1,cols+1):
        for i in range(rows):
            for j in range(cols):
                SP[i,j,k] = min(SP[i,j,k-1],
                                SP[i,k-1,k-1]+SP[k-1,j,k-1])
    return(SP[:, :, cols])
```

Implementation

- Shortest path matrix SP is $n \times n \times (n + 1)$
- Initialize $SP[i, j, 0]$ to edge weight $W(i, j)$, or ∞ if no edge

```
def floydwarshall(WMat):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows*rows+1  
    SP = np.zeros(shape=(rows,cols,cols+1))  
    for i in range(rows):  
        for j in range(cols):  
            SP[i,j,0] = infinity  
  
    for i in range(rows):  
        for j in range(cols):  
            if WMat[i,j,0] == 1:  
                SP[i,j,0] = WMat[i,j,1]  
  
    for k in range(1,cols+1):  
        for i in range(rows):  
            for j in range(cols):  
                SP[i,j,k] = min(SP[i,j,k-1],  
                                SP[i,k-1,k-1]+SP[k-1,j,k-1])  
    return(SP[:, :, cols])
```

Implementation

- Shortest path matrix SP is $n \times n \times (n + 1)$
- Initialize $SP[i, j, 0]$ to edge weight $W(i, j)$, or ∞ if no edge
- Update $SP[i, j, k]$ from $SP[i, j, k - 1]$ using the Floyd-Warshall update rule

```
def floydwarshall(WMat):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows*rows+1  
    SP = np.zeros(shape=(rows,cols,cols+1))  
    for i in range(rows):  
        for j in range(cols):  
            SP[i,j,0] = infinity  
  
    for i in range(rows):  
        for j in range(cols):  
            if WMat[i,j,0] == 1:  
                SP[i,j,0] = WMat[i,j,1]  
  
    for k in range(1,cols+1):  
        for i in range(rows):  
            for j in range(cols):  
                SP[i,j,k] = min(SP[i,j,k-1],  
                                SP[i,k-1,k-1]+SP[k-1,j,k-1])  
    return(SP[:, :, cols])
```

Implementation

- Shortest path matrix SP is $n \times n \times (n + 1)$
- Initialize $SP[i, j, 0]$ to edge weight $W(i, j)$, or ∞ if no edge
- Update $SP[i, j, k]$ from $SP[i, j, k - 1]$ using the Floyd-Warshall update rule
- Time complexity is $O(n^3)$

```
def floydwarshall(WMat):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows*rows+1  
    SP = np.zeros(shape=(rows,cols,cols+1))  
    for i in range(rows):  
        for j in range(cols):  
            SP[i,j,0] = infinity  
  
    for i in range(rows):  
        for j in range(cols):  
            if WMat[i,j,0] == 1:  
                SP[i,j,0] = WMat[i,j,1]  
  
    for k in range(1,cols+1):  
        for i in range(rows):  
            for j in range(cols):  
                SP[i,j,k] = min(SP[i,j,k-1],  
                                SP[i,k-1,k-1]+SP[k-1,j,k-1])  
    return(SP[:, :, cols])
```

Implementation

- Shortest path matrix SP is $n \times n \times (n + 1)$
- Initialize $SP[i, j, 0]$ to edge weight $W(i, j)$, or ∞ if no edge
- Update $SP[i, j, k]$ from $SP[i, j, k - 1]$ using the Floyd-Warshall update rule
- Time complexity is $O(n^3)$
- We only need $SP[i, j, k - 1]$ to compute $SP[i, j, k]$

```
def floydwarshall(WMat):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows*rows+1  
    SP = np.zeros(shape=(rows,cols,cols+1))  
    for i in range(rows):  
        for j in range(cols):  
            SP[i,j,0] = infinity  
  
    for i in range(rows):  
        for j in range(cols):  
            if WMat[i,j,0] == 1:  
                SP[i,j,0] = WMat[i,j,1]  
  
    for k in range(1,cols+1):  
        for i in range(rows):  
            for j in range(cols):  
                SP[i,j,k] = min(SP[i,j,k-1],  
                                SP[i,k-1,k-1]+SP[k-1,j,k-1])  
    return(SP[:, :, cols])
```

Implementation

- Shortest path matrix SP is $n \times n \times (n + 1)$
- Initialize $SP[i, j, 0]$ to edge weight $W(i, j)$, or ∞ if no edge
- Update $SP[i, j, k]$ from $SP[i, j, k - 1]$ using the Floyd-Warshall update rule
- Time complexity is $O(n^3)$
- We only need $SP[i, j, k - 1]$ to compute $SP[i, j, k]$
- Maintain two “slices” $SP[i, j]$, $SP'[i, j]$, compute SP' from SP , copy SP' to SP , save space

```
def floydwarshall(WMat):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows*rows+1  
    SP = np.zeros(shape=(rows,cols,cols+1))  
    for i in range(rows):  
        for j in range(cols):  
            SP[i,j,0] = infinity  
  
    for i in range(rows):  
        for j in range(cols):  
            if WMat[i,j,0] == 1:  
                SP[i,j,0] = WMat[i,j,1]  
  
    for k in range(1,cols+1):  
        for i in range(rows):  
            for j in range(cols):  
                SP[i,j,k] = min(SP[i,j,k-1],  
                                SP[i,k-1,k-1]+SP[k-1,j,k-1])  
    return(SP[:, :, cols])
```

Summary

- Warshall's algorithm is an alternative way to compute transitive closure
 - $B^k[i, j] = 1$ if we can reach j from i using vertices in $\{0, 1, \dots, k-1\}$
- Adapt Warshall's algorithm to compute all pairs shortest paths
 - $SP^k[i, j]$ is the length of the shortest path from i to j using vertices in $\{0, 1, \dots, k-1\}$
 - $SP^n[i, j]$ is the length of the overall shortest path
 - Floyd-Warshall algorithm
- Works with negative edge weights, assuming no negative cycles
- Simple nested loop implementation, time $O(n^3)$
- Space can be limited to $O(n^2)$ by reusing two “slices” SP and SP'

Minimum Cost Spanning Trees

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 5

Examples

Roads

- District hit by cyclone, roads are damaged
- Government sets to work to restore roads
- Priority is to ensure that all parts of the district can be reached
- What set of roads should be restored first?

Examples

Roads

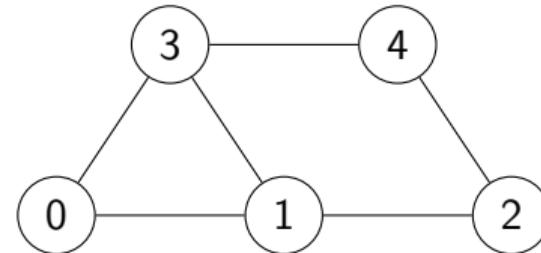
- District hit by cyclone, roads are damaged
- Government sets to work to restore roads
- Priority is to ensure that all parts of the district can be reached
- What set of roads should be restored first?

Fibre optic cables

- Internet service provider has a network of fibre optic cables
- Wants to ensure redundancy against cable faults
- Lay secondary cables in parallel to first
- What is the minimum number of cables to be doubled up so that entire network is connected via redundant links?

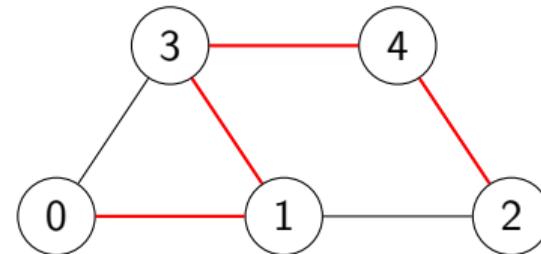
Spanning trees

- Retain a minimal set of edges so that graph remains connected



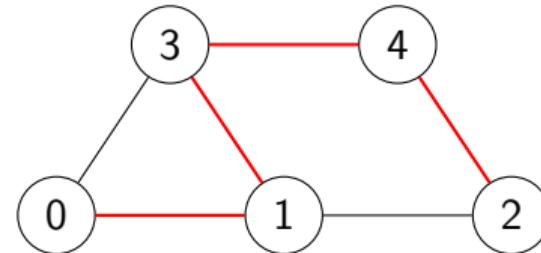
Spanning trees

- Retain a minimal set of edges so that graph remains connected
- Recall that a minimally connected graph is a **tree**
 - Adding an edge to a tree creates a loop
 - Removing an edge disconnects the graph



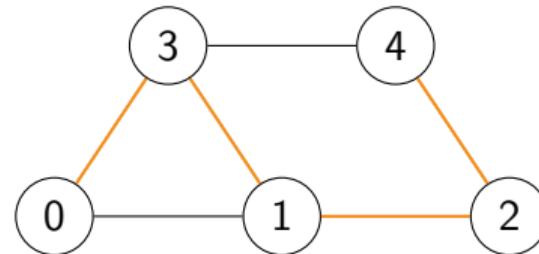
Spanning trees

- Retain a minimal set of edges so that graph remains connected
- Recall that a minimally connected graph is a **tree**
 - Adding an edge to a tree creates a loop
 - Removing an edge disconnects the graph
- Want a tree that connects all the vertices — **spanning tree**



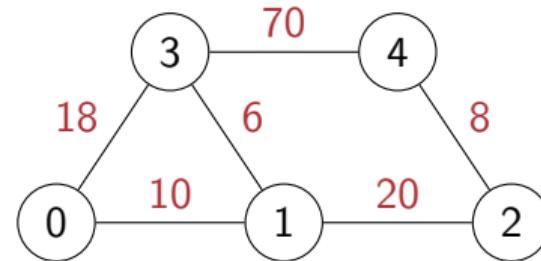
Spanning trees

- Retain a minimal set of edges so that graph remains connected
- Recall that a minimally connected graph is a **tree**
 - Adding an edge to a tree creates a loop
 - Removing an edge disconnects the graph
- Want a tree that connects all the vertices — **spanning tree**
- More than one spanning tree, in general



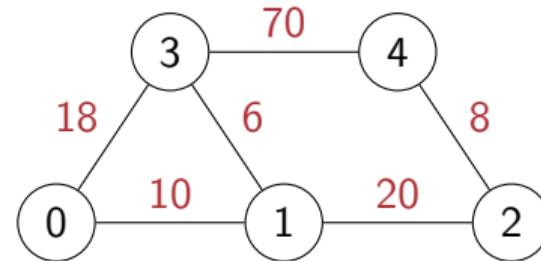
Spanning trees with costs

- Restoring a road or laying a fibre optic cable has a cost



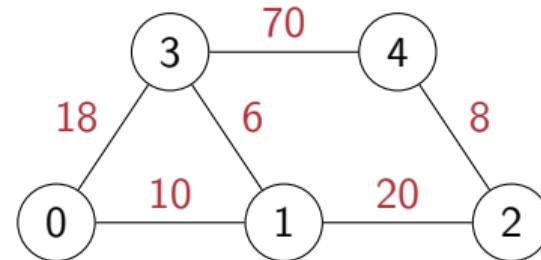
Spanning trees with costs

- Restoring a road or laying a fibre optic cable has a cost
- Minimum cost spanning tree
 - Add the cost of all the edges in the tree
 - Among the different spanning trees, choose one with minimum cost



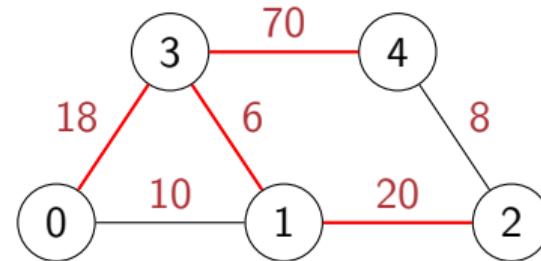
Spanning trees with costs

- Restoring a road or laying a fibre optic cable has a cost
- Minimum cost spanning tree
 - Add the cost of all the edges in the tree
 - Among the different spanning trees, choose one with minimum cost
- Example



Spanning trees with costs

- Restoring a road or laying a fibre optic cable has a cost
- Minimum cost spanning tree
 - Add the cost of all the edges in the tree
 - Among the different spanning trees, choose one with minimum cost
- Example
 - Spanning tree, Cost is 114 — not minimum cost spanning tree



Spanning trees with costs

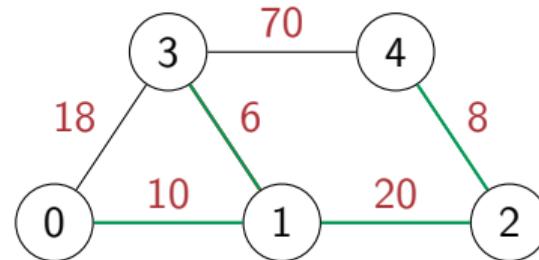
- Restoring a road or laying a fibre optic cable has a cost

- Minimum cost spanning tree

- Add the cost of all the edges in the tree
 - Among the different spanning trees, choose one with minimum cost

- Example

- Spanning tree, Cost is 114 — not minimum cost spanning tree
 - Another spanning tree, Cost is 44 — minimum cost spanning tree



Some facts about trees

Definition A tree is a connected acyclic graph.

Fact 1

A tree on n vertices has exactly $n - 1$ edges

- Initially, one single component
- Deleting edge (i, j) must split component
 - Otherwise, there is still a path from i to j , combine with (i, j) to form cycle
- Each edge deletion creates one more component
- Deleting $n - 1$ edges creates n components, each an isolated vertex

Some facts about trees

Definition A tree is a connected acyclic graph.

Fact 1

A tree on n vertices has exactly $n - 1$ edges

- Initially, one single component
- Deleting edge (i,j) must split component
 - Otherwise, there is still a path from i to j , combine with (i,j) to form cycle
- Each edge deletion creates one more component
- Deleting $n - 1$ edges creates n components, each an isolated vertex

Fact 2

Adding an edge to a tree must create a cycle.

- Suppose we add an edge (i,j)
- Tree is connected, so there is already a path from i to j
- The new edge (i,j) combined with this path from i to j forms a cycle

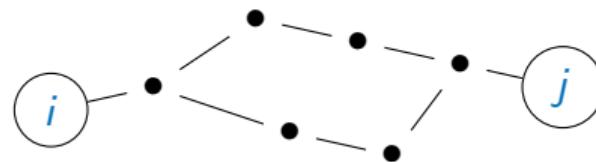
Some facts about trees

Definition A tree is a connected acyclic graph.

Fact 3

In a tree, every pair of vertices is connected by a unique path.

- If there are two paths from i to j , there must be a cycle



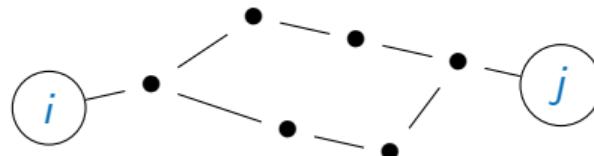
Some facts about trees

Definition A tree is a connected acyclic graph.

Fact 3

In a tree, every pair of vertices is connected by a unique path.

- If there are two paths from i to j , there must be a cycle



Observation

Any two of the following facts about a graph G implies the third

- G is connected
- G is acyclic
- G has $n - 1$ edges

Building minimum cost spanning trees

- We will use these facts about trees to build minimum cost spanning trees

Building minimum cost spanning trees

- We will use these facts about trees to build minimum cost spanning trees
- Two natural strategies

Building minimum cost spanning trees

- We will use these facts about trees to build minimum cost spanning trees
- Two natural strategies
- Start with the smallest edge and “grow” a tree
 - Prim’s algorithm

Building minimum cost spanning trees

- We will use these facts about trees to build minimum cost spanning trees
- Two natural strategies
- Start with the smallest edge and “grow” a tree
 - Prim’s algorithm
- Scan the edges in ascending order of weight to connect components without forming cycles
 - Kruskal’s algorithm

Minimum Cost Spanning Trees: Prim's Algorithm

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 5

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V

Minimum cost spanning tree (MCST)

- Weighted undirected graph,

$$G = (V, E), W : E \rightarrow \mathbb{R}$$

- G assumed to be connected

- Find a minimum cost **spanning tree**

- Tree connecting all vertices in V

- **Strategy**

- Incrementally grow the minimum cost spanning tree

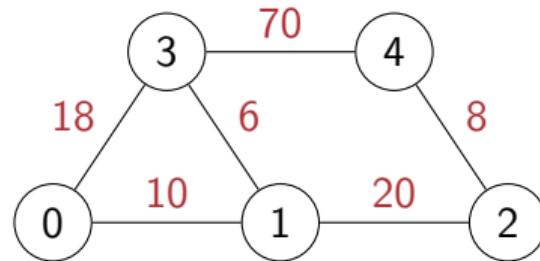
- Start with a smallest weight edge overall

- Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- Strategy
 - Incrementally grow the minimum cost spanning tree
 - Start with a smallest weight edge overall
 - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

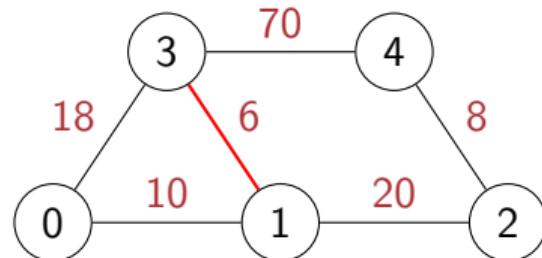
Example



Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- Strategy
 - Incrementally grow the minimum cost spanning tree
 - Start with a smallest weight edge overall
 - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Example

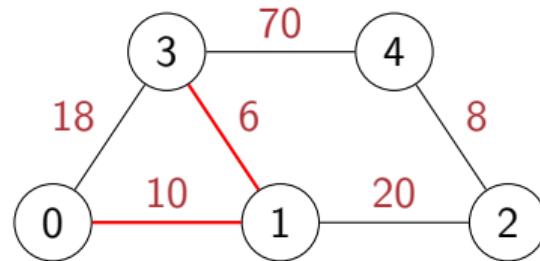


- Start with smallest edge, $(1, 3)$

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- Strategy
 - Incrementally grow the minimum cost spanning tree
 - Start with a smallest weight edge overall
 - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Example

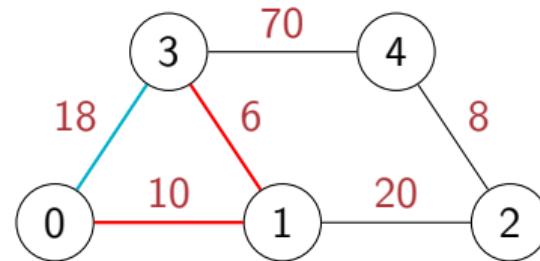


- Start with smallest edge, $(1, 3)$
- Extend the tree with $(1, 0)$

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- Strategy
 - Incrementally grow the minimum cost spanning tree
 - Start with a smallest weight edge overall
 - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Example

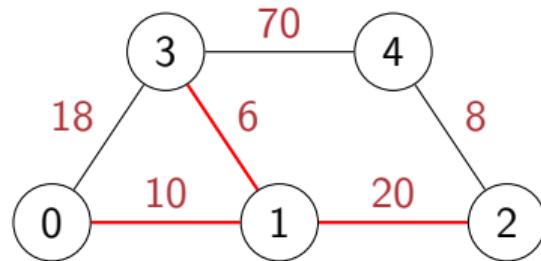


- Start with smallest edge, $(1, 3)$
- Extend the tree with $(1, 0)$
- Can't add $(0, 3)$, forms a cycle

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- Strategy
 - Incrementally grow the minimum cost spanning tree
 - Start with a smallest weight edge overall
 - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Example

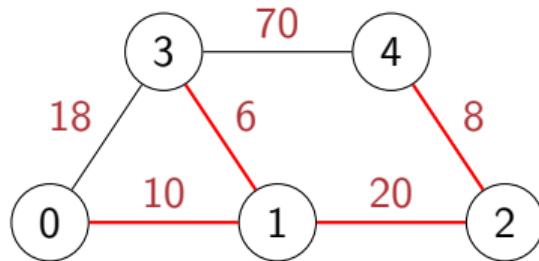


- Start with smallest edge, $(1, 3)$
- Extend the tree with $(1, 0)$
- Can't add $(0, 3)$, forms a cycle
- Instead, extend the tree with $(1, 2)$

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- Strategy
 - Incrementally grow the minimum cost spanning tree
 - Start with a smallest weight edge overall
 - Extend the current tree by adding the smallest edge from the tree to a vertex not yet in the tree

Example



- Start with smallest edge, $(1, 3)$
- Extend the tree with $(1, 0)$
- Can't add $(0, 3)$, forms a cycle
- Instead, extend the tree with $(1, 2)$
- Extend the tree with $(2, 4)$

Prim's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$

Prim's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Incrementally build an MCST
 - $TV \subseteq V$: tree vertices, already added to MCST
 - $TE \subseteq E$: tree edges, already added to MCST

Prim's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Incrementally build an MCST
 - $TV \subseteq V$: tree vertices, already added to MCST
 - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$

Prim's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Incrementally build an MCST
 - $TV \subseteq V$: tree vertices, already added to MCST
 - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
 - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST

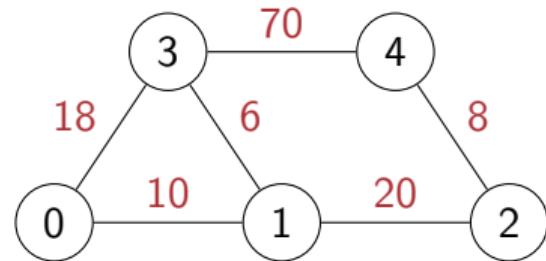
Prim's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Incrementally build an MCST
 - $TV \subseteq V$: tree vertices, already added to MCST
 - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
 - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
 - Choose minimum weight edge $f = (u, v)$ such that $u \in TV, v \notin TV$
 - Add v to TV , f to TE

Prim's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Incrementally build an MCST
 - $TV \subseteq V$: tree vertices, already added to MCST
 - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
 - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
 - Choose minimum weight edge $f = (u, v)$ such that $u \in TV, v \notin TV$
 - Add v to TV , f to TE

Example



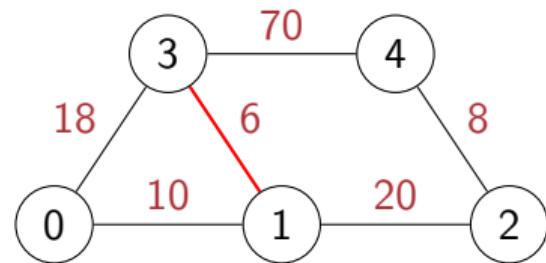
$$TV = \emptyset$$

$$TE = \emptyset$$

Prim's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Incrementally build an MCST
 - $TV \subseteq V$: tree vertices, already added to MCST
 - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
 - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
 - Choose minimum weight edge $f = (u, v)$ such that $u \in TV, v \notin TV$
 - Add v to TV , f to TE

Example



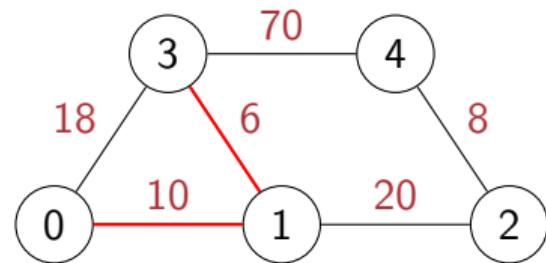
$$TV = \{1, 3\}$$

$$TE = \{(1, 3)\}$$

Prim's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Incrementally build an MCST
 - $TV \subseteq V$: tree vertices, already added to MCST
 - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
 - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
 - Choose minimum weight edge $f = (u, v)$ such that $u \in TV, v \notin TV$
 - Add v to TV , f to TE

Example



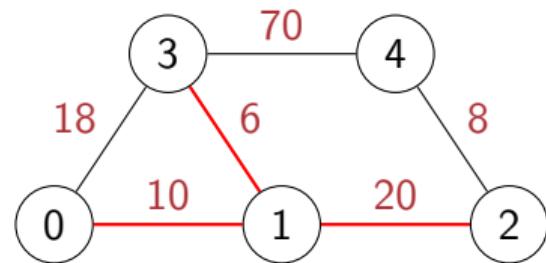
$$TV = \{1, 3, 0\}$$

$$TE = \{(1, 3), (1, 0)\}$$

Prim's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Incrementally build an MCST
 - $TV \subseteq V$: tree vertices, already added to MCST
 - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
 - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
 - Choose minimum weight edge $f = (u, v)$ such that $u \in TV$, $v \notin TV$
 - Add v to TV , f to TE

Example



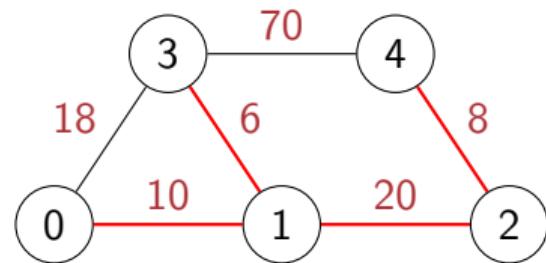
$$TV = \{1, 3, 0, 2\}$$

$$TE = \{(1, 3), (1, 0), (1, 2)\}$$

Prim's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Incrementally build an MCST
 - $TV \subseteq V$: tree vertices, already added to MCST
 - $TE \subseteq E$: tree edges, already added to MCST
- Initially, $TV = TE = \emptyset$
- Choose minimum weight edge $e = (i, j)$
 - Set $TV = \{i, j\}$, $TE = \{e\}$ MCST
- Repeat $n - 2$ times
 - Choose minimum weight edge $f = (u, v)$ such that $u \in TV$, $v \notin TV$
 - Add v to TV , f to TE

Example



$$TV = \{1, 3, 0, 2, 4\}$$

$$TE = \{(1, 3), (1, 0), (1, 2), (2, 4)\}$$

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
- Every MCST must include e

Correctness of Prim's algorithm

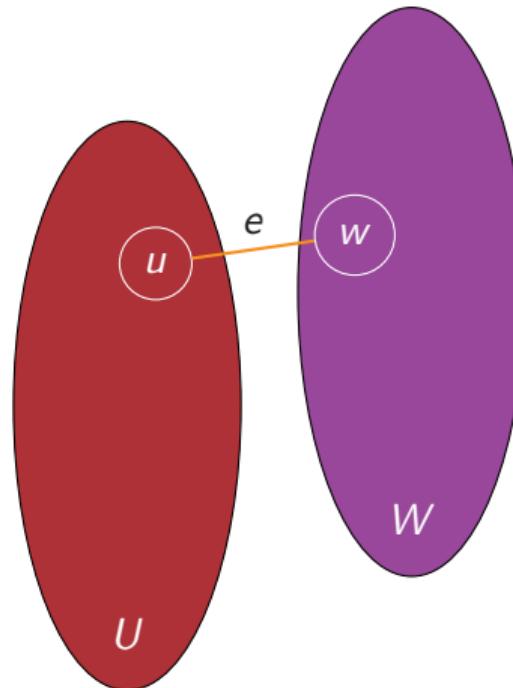
Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
 - Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
 - Every MCST must include e
-
- Assume for now, all edge weights distinct

Correctness of Prim's algorithm

Minimum Separator Lemma

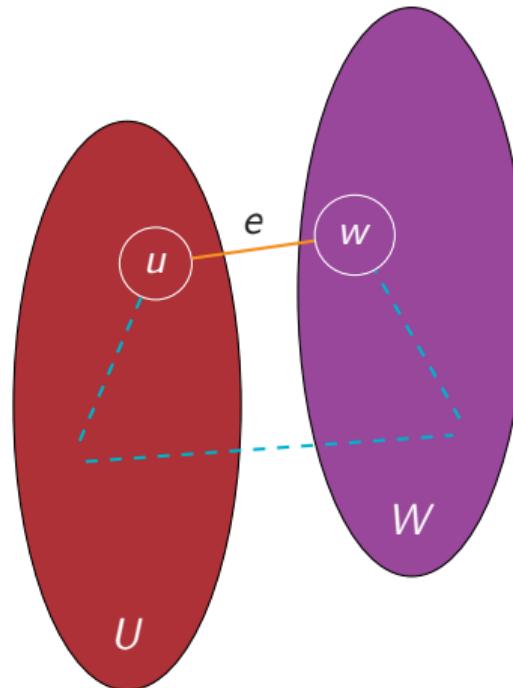
- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
 - Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
 - Every MCST must include e
-
- Assume for now, all edge weights distinct
 - Let T be an MCST, $e \notin T$



Correctness of Prim's algorithm

Minimum Separator Lemma

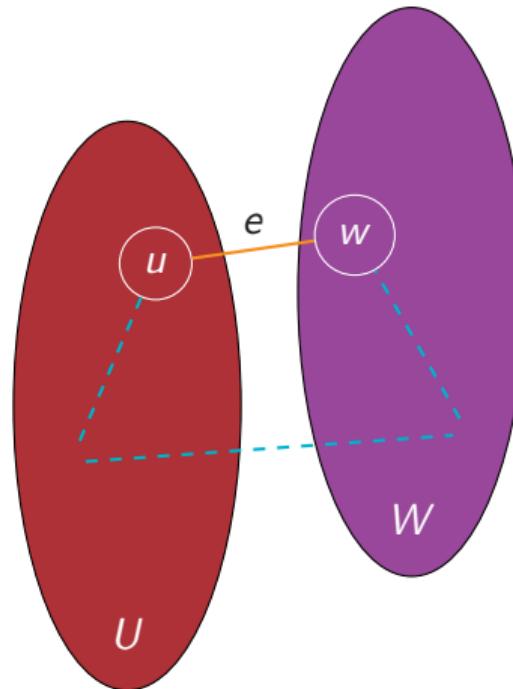
- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
 - Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
 - Every MCST must include e
-
- Assume for now, all edge weights distinct
 - Let T be an MCST, $e \notin T$
 - T contains a path p from u to w



Correctness of Prim's algorithm

Minimum Separator Lemma

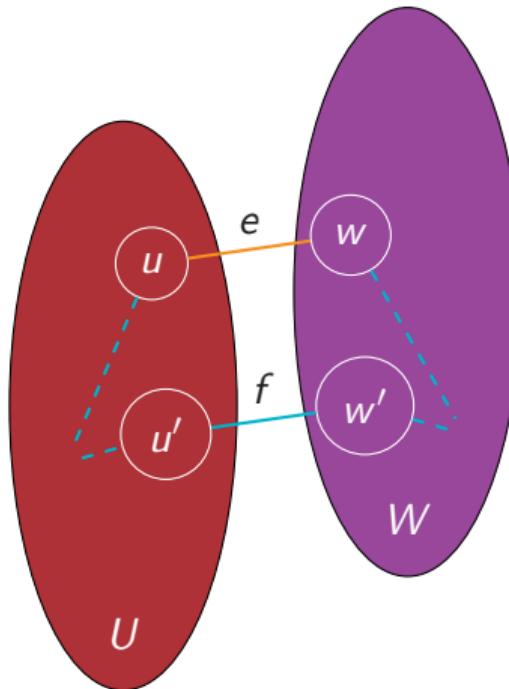
- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
 - Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
 - Every MCST must include e
-
- Assume for now, all edge weights distinct
 - Let T be an MCST, $e \notin T$
 - T contains a path p from u to w
 - p starts in U , ends in W



Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
 - Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
 - Every MCST must include e
-
- Assume for now, all edge weights distinct
 - Let T be an MCST, $e \notin T$
 - T contains a path p from u to w
 - p starts in U , ends in W
 - Let $f = (u', w')$ be the first edge on p crossing from U to W

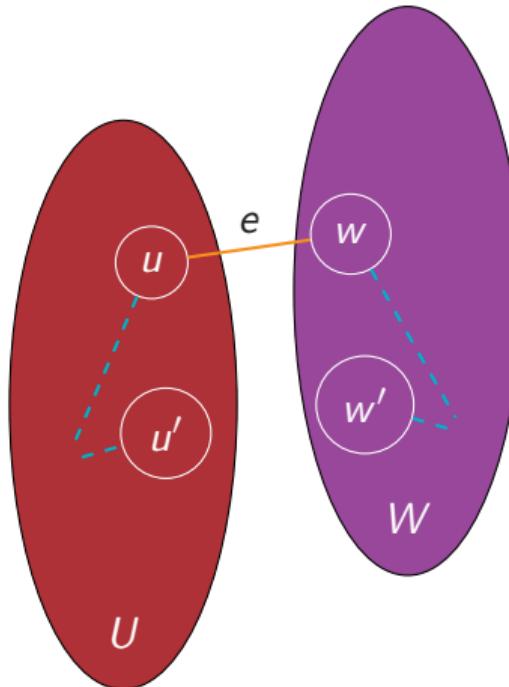


Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
- Every MCST must include e

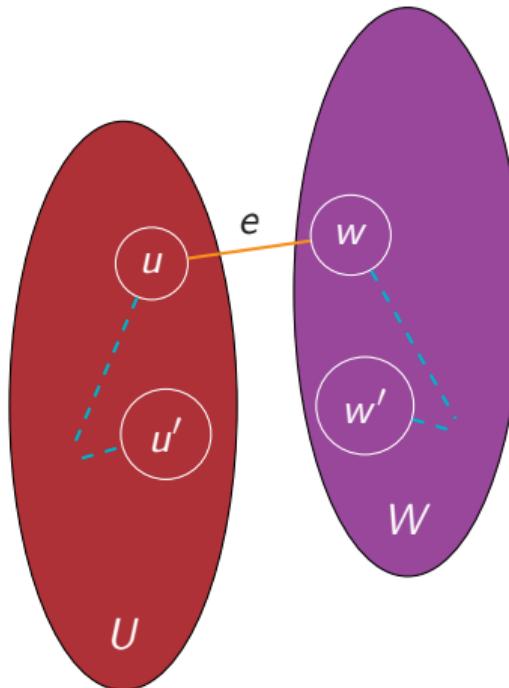
- Assume for now, all edge weights distinct
- Let T be an MCST, $e \notin T$
- T contains a path p from u to w
 - p starts in U , ends in W
 - Let $f = (u', w')$ be the first edge on p crossing from U to W
 - Drop f , add e to get a cheaper spanning tree



Correctness of Prim's algorithm

Minimum Separator Lemma

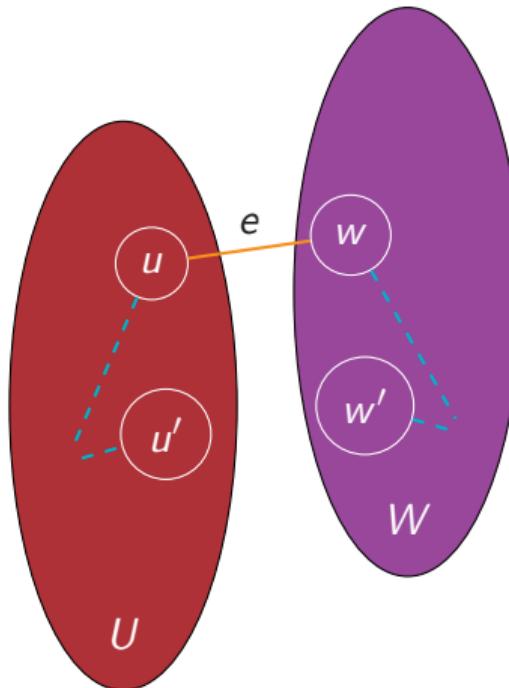
- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
 - Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
 - Every MCST must include e
-
- Assume for now, all edge weights distinct
 - What if two edges have the same weight?



Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
 - Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
 - Every MCST must include e
-
- Assume for now, all edge weights distinct
 - What if two edges have the same weight?
 - Assign each edge a unique index from 0 to $m - 1$

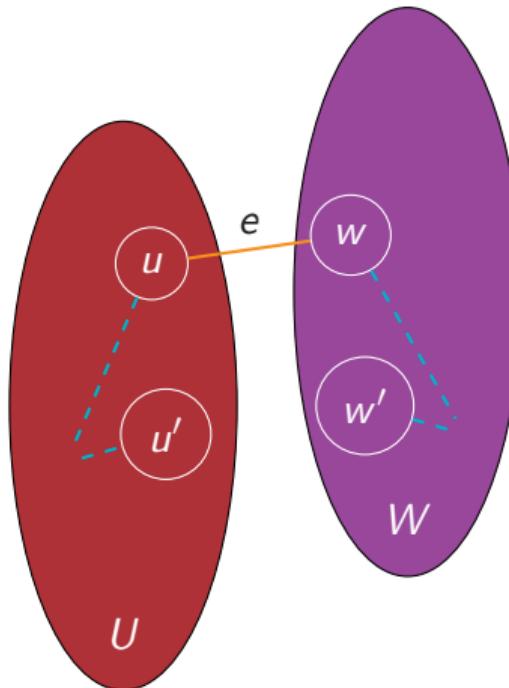


Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
- Every MCST must include e

- Assume for now, all edge weights distinct
- What if two edges have the same weight?
- Assign each edge a unique index from 0 to $m - 1$
- Define $(e, i) < (f, j)$ if $W(e) < W(f)$ or $W(e) = W(f)$ and $i < j$



Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
- Every MCST must include e

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
 - Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
 - Every MCST must include e
-
- In Prim's algorithm, TV and $W = V \setminus TV$ partition V

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
- Every MCST must include e

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V
- Algorithm picks smallest edge connecting TV and W , which must belong to every MCST

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
- Every MCST must include e

- In fact, for any $v \in V$, $\{v\}$ and $V \setminus \{v\}$ form a partition

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V
- Algorithm picks smallest edge connecting TV and W , which must belong to every MCST

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
- Every MCST must include e

- In fact, for any $v \in V$, $\{v\}$ and $V \setminus \{v\}$ form a partition
- The smallest weight edge leaving any vertex must belong to every MCST

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V
- Algorithm picks smallest edge connecting TV and W , which must belong to every MCST

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
- Every MCST must include e

- In Prim's algorithm, T and $W = V \setminus T$ partition V
- Algorithm picks smallest edge connecting T and W , which must belong to every MCST

- In fact, for any $v \in V$, $\{v\}$ and $V \setminus \{v\}$ form a partition
- The smallest weight edge leaving any vertex must belong to every MCST
- We started with overall minimum cost edge

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
- Every MCST must include e

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V
- Algorithm picks smallest edge connecting TV and W , which must belong to every MCST

- In fact, for any $v \in V$, $\{v\}$ and $V \setminus \{v\}$ form a partition
- The smallest weight edge leaving any vertex must belong to every MCST
- We started with overall minimum cost edge
- Instead, can start at any vertex v , with $TV = \{v\}$ and $TE = \emptyset$

Correctness of Prim's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
- Every MCST must include e

- In Prim's algorithm, TV and $W = V \setminus TV$ partition V
- Algorithm picks smallest edge connecting TV and W , which must belong to every MCST

- In fact, for any $v \in V$, $\{v\}$ and $V \setminus \{v\}$ form a partition
- The smallest weight edge leaving any vertex must belong to every MCST
- We started with overall minimum cost edge
- Instead, can start at any vertex v , with $TV = \{v\}$ and $TE = \emptyset$
- First iteration will pick minimum cost edge from v

Implementation

- Keep track of

- `visited[v]` – is `v` in the spanning tree?
- `distance[v]` – shortest distance from `v` to the tree
- `TreeEdges` – edges in the current spanning tree

```
def primlist(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
    (visited,distance,TreeEdges) = ({}, {}, [])  
    for v in WList.keys():  
        (visited[v],distance[v]) = (False,infinity)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        distance[v] = d  
    for i in WList.keys():  
        (mindist,nextv) = (infinity,None)  
        for u in WList.keys():  
            for (v,d) in WList[u]:  
                if visited[u] and (not visited[v]) and d < mindist:  
                    (mindist,nextv,nexte) = (d,v,(u,v))  
    if nextv is None:  
        break  
    visited[nextv] = True  
    TreeEdges.append(nexte)  
    for (v,d) in WList[nextv]:  
        if not visited[v]:  
            distance[v] = min(distance[v],d)  
return(TreeEdges)
```

Implementation

- Keep track of
 - `visited[v]` – is `v` in the spanning tree?
 - `distance[v]` – shortest distance from `v` to the tree
 - `TreeEdges` – edges in the current spanning tree
- Initialize `visited[v]` to `False`, `distance[v]` to `infinity`

```
def primlist(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
(visited,distance,TreeEdges) = ({}, {}, [])  
for v in WList.keys():  
    (visited[v],distance[v]) = (False,infinity)  
visited[0] = True  
for (v,d) in WList[0]:  
    distance[v] = d  
for i in WList.keys():  
    (mindist,nextv) = (infinity,None)  
    for u in WList.keys():  
        for (v,d) in WList[u]:  
            if visited[u] and (not visited[v]) and d < mindist:  
                (mindist,nextv,nexte) = (d,v,(u,v))  
    if nextv is None:  
        break  
    visited[nextv] = True  
    TreeEdges.append(nexte)  
    for (v,d) in WList[nextv]:  
        if not visited[v]:  
            distance[v] = min(distance[v],d)  
return(TreeEdges)
```

Implementation

- Keep track of
 - `visited[v]` – is `v` in the spanning tree?
 - `distance[v]` – shortest distance from `v` to the tree
 - `TreeEdges` – edges in the current spanning tree
- Initialize `visited[v]` to `False`, `distance[v]` to `infinity`
- First add vertex `0` to tree

```
def primlist(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
(visited,distance,TreeEdges) = ({}, {}, [])  
for v in WList.keys():  
    (visited[v],distance[v]) = (False,infinity)  
visited[0] = True  
for (v,d) in WList[0]:  
    distance[v] = d  
for i in WList.keys():  
    (mindist,nextv) = (infinity,None)  
    for u in WList.keys():  
        for (v,d) in WList[u]:  
            if visited[u] and (not visited[v]) and d < mindist:  
                (mindist,nextv,nexte) = (d,v,(u,v))  
    if nextv is None:  
        break  
    visited[nextv] = True  
    TreeEdges.append(nexte)  
    for (v,d) in WList[nextv]:  
        if not visited[v]:  
            distance[v] = min(distance[v],d)  
return(TreeEdges)
```

Implementation

- Keep track of
 - `visited[v]` – is `v` in the spanning tree?
 - `distance[v]` – shortest distance from `v` to the tree
 - `TreeEdges` – edges in the current spanning tree
- Initialize `visited[v]` to `False`, `distance[v]` to `infinity`
- First add vertex `0` to tree
- Find edge `(u,v)` leaving the tree where `distance[v]` is minimum, add it to the tree, update `distance[w]` of neighbours

```
def primlist(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
(visited,distance,TreeEdges) = ({}, {}, [])  
for v in WList.keys():  
    (visited[v],distance[v]) = (False,infinity)  
visited[0] = True  
for (v,d) in WList[0]:  
    distance[v] = d  
for i in WList.keys():  
    (mindist,nextv) = (infinity,None)  
    for u in WList.keys():  
        for (v,d) in WList[u]:  
            if visited[u] and (not visited[v]) and d < mindist:  
                (mindist,nextv,nexte) = (d,v,(u,v))  
    if nextv is None:  
        break  
    visited[nextv] = True  
    TreeEdges.append(nexte)  
    for (v,d) in WList[nextv]:  
        if not visited[v]:  
            distance[v] = min(distance[v],d)  
return(TreeEdges)
```

Complexity

- Initialization takes ($O(n)$)

```
def primlist(WList):
    infinity = 1 + max([d for u in WList.keys()
                        for (v,d) in WList[u]])
    (visited,distance,TreeEdges) = ({}, {}, [])
    for v in WList.keys():
        (visited[v],distance[v]) = (False,infinity)
    visited[0] = True
    for (v,d) in WList[0]:
        distance[v] = d
    for i in WList.keys():
        (mindist,nextv) = (infinity,None)
        for u in WList.keys():
            for (v,d) in WList[u]:
                if visited[u] and (not visited[v]) and d < mindist:
                    (mindist,nextv,nexte) = (d,v,(u,v))
        if nextv is None:
            break
        visited[nextv] = True
        TreeEdges.append(nexte)
        for (v,d) in WList[nextv]:
            if not visited[v]:
                distance[v] = min(distance[v],d)
    return(TreeEdges)
```

Complexity

- Initialization takes $O(n)$
- Loop to add nodes to the tree runs $O(n)$ times

```
def primlist(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
    (visited,distance,TreeEdges) = ({}, {}, [])  
    for v in WList.keys():  
        (visited[v],distance[v]) = (False,infinity)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        distance[v] = d  
    for i in WList.keys():  
        (mindist,nextv) = (infinity,None)  
        for u in WList.keys():  
            for (v,d) in WList[u]:  
                if visited[u] and (not visited[v]) and d < mindist:  
                    (mindist,nextv,nexte) = (d,v,(u,v))  
        if nextv is None:  
            break  
        visited[nextv] = True  
        TreeEdges.append(nexte)  
        for (v,d) in WList[nextv]:  
            if not visited[v]:  
                distance[v] = min(distance[v],d)  
    return(TreeEdges)
```

Complexity

- Initialization takes $O(n)$
- Loop to add nodes to the tree runs $O(n)$ times
- Each iteration takes $O(m)$ time to find a node to add

```
def primlist(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
    (visited,distance,TreeEdges) = ({}, {}, [])  
    for v in WList.keys():  
        (visited[v],distance[v]) = (False,infinity)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        distance[v] = d  
    for i in WList.keys():  
        (mindist,nextv) = (infinity,None)  
        for u in WList.keys():  
            for (v,d) in WList[u]:  
                if visited[u] and (not visited[v]) and d < mindist:  
                    (mindist,nextv,nexte) = (d,v,(u,v))  
        if nextv is None:  
            break  
        visited[nextv] = True  
        TreeEdges.append(nexte)  
        for (v,d) in WList[nextv]:  
            if not visited[v]:  
                distance[v] = min(distance[v],d)  
    return(TreeEdges)
```

Complexity

- Initialization takes $O(n)$
- Loop to add nodes to the tree runs $O(n)$ times
- Each iteration takes $O(m)$ time to find a node to add
- Overall time is $O(mn)$, which could be $O(n^3)$!

```
def primlist(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
    (visited,distance,TreeEdges) = ({}, {}, [])  
    for v in WList.keys():  
        (visited[v],distance[v]) = (False,infinity)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        distance[v] = d  
    for i in WList.keys():  
        (mindist,nextv) = (infinity,None)  
        for u in WList.keys():  
            for (v,d) in WList[u]:  
                if visited[u] and (not visited[v]) and d < mindist:  
                    (mindist,nextv,nexte) = (d,v,(u,v))  
    if nextv is None:  
        break  
    visited[nextv] = True  
    TreeEdges.append(nexte)  
    for (v,d) in WList[nextv]:  
        if not visited[v]:  
            distance[v] = min(distance[v],d)  
return(TreeEdges)
```

Complexity

- Initialization takes $O(n)$
- Loop to add nodes to the tree runs $O(n)$ times
- Each iteration takes $O(m)$ time to find a node to add
- Overall time is $O(mn)$, which could be $O(n^3)$!
- Can we do better?

```
def primlist(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
    (visited,distance,TreeEdges) = ({}, {}, [])  
    for v in WList.keys():  
        (visited[v],distance[v]) = (False,infinity)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        distance[v] = d  
    for i in WList.keys():  
        (mindist,nextv) = (infinity,None)  
        for u in WList.keys():  
            for (v,d) in WList[u]:  
                if visited[u] and (not visited[v]) and d < mindist:  
                    (mindist,nextv,nexte) = (d,v,(u,v))  
    if nextv is None:  
        break  
    visited[nextv] = True  
    TreeEdges.append(nexte)  
    for (v,d) in WList[nextv]:  
        if not visited[v]:  
            distance[v] = min(distance[v],d)  
return(TreeEdges)
```

Improved implementation

- For each v , keep track of its nearest neighbour in the tree
 - $\text{visited}[v]$ – is v in the spanning tree?
 - $\text{distance}[v]$ – shortest distance from v to the tree
 - $\text{nbr}[v]$ – nearest neighbour of v in tree

```
def primlist2(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
    (visited,distance,nbr) = ({}, {}, {})  
    for v in WList.keys():  
        (visited[v],distance[v],nbr[v]) = (False,infinity,-1)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        (distance[v],nbr[v]) = (d,0)  
    for i in range(1,len(WList.keys())):  
        nextd = min([distance[v] for v in WList.keys()  
                     if not visited[v]])  
        nextvlist = [v for v in WList.keys()  
                     if (not visited[v]) and  
                         distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for (v,d) in WList[nextv]:  
            if not visited[v]:  
                (distance[v],nbr[v]) = (min(distance[v],d),nextv)  
    return(nbr)
```

Improved implementation

- For each v , keep track of its nearest neighbour in the tree
 - $\text{visited}[v]$ – is v in the spanning tree?
 - $\text{distance}[v]$ – shortest distance from v to the tree
 - $\text{nbr}[v]$ – nearest neighbour of v in tree
- Scan all non-tree vertices to find nextv with minimum distance

```
def primlist2(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
    (visited,distance,nbr) = ({}, {}, {})  
    for v in WList.keys():  
        (visited[v],distance[v],nbr[v]) = (False,infinity,-1)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        (distance[v],nbr[v]) = (d,0)  
    for i in range(1,len(WList.keys())):  
        nextd = min([distance[v] for v in WList.keys()  
                     if not visited[v]])  
        nextvlist = [v for v in WList.keys()  
                     if (not visited[v]) and  
                         distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for (v,d) in WList[nextv]:  
            if not visited[v]:  
                (distance[v],nbr[v]) = (min(distance[v],d),nextv)  
    return(nbr)
```

Improved implementation

- For each v , keep track of its nearest neighbour in the tree
 - $\text{visited}[v]$ – is v in the spanning tree?
 - $\text{distance}[v]$ – shortest distance from v to the tree
 - $\text{nbr}[v]$ – nearest neighbour of v in tree
- Scan all non-tree vertices to find nextv with minimum distance
- Then $(\text{nbr}[\text{nextv}], \text{nextv})$ is the tree edge to add

```
def primlist2(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
    (visited,distance,nbr) = ({}, {}, {})  
    for v in WList.keys():  
        (visited[v],distance[v],nbr[v]) = (False,infinity,-1)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        (distance[v],nbr[v]) = (d,0)  
    for i in range(1,len(WList.keys())):  
        nextd = min([distance[v] for v in WList.keys()  
                     if not visited[v]])  
        nextvlist = [v for v in WList.keys()  
                     if (not visited[v]) and  
                         distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for (v,d) in WList[nextv]:  
            if not visited[v]:  
                (distance[v],nbr[v]) = (min(distance[v],d),nextv)  
    return(nbr)
```

Improved implementation

- For each v , keep track of its nearest neighbour in the tree
 - $\text{visited}[v]$ – is v in the spanning tree?
 - $\text{distance}[v]$ – shortest distance from v to the tree
 - $\text{nbr}[v]$ – nearest neighbour of v in tree
- Scan all non-tree vertices to find nextv with minimum distance
- Then $(\text{nbr}[\text{nextv}], \text{nextv})$ is the tree edge to add
- Update $\text{distance}[v]$ and $\text{nbr}[v]$ for all neighbours of nextv

```
def primlist2(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                      for (v,d) in WList[u]])  
    (visited,distance,nbr) = ({},{},{})  
    for v in WList.keys():  
        (visited[v],distance[v],nbr[v]) = (False,infinity,-1)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        (distance[v],nbr[v]) = (d,0)  
    for i in range(1,len(WList.keys())):  
        nextd = min([distance[v] for v in WList.keys()  
                     if not visited[v]])  
        nextvlist = [v for v in WList.keys()  
                     if (not visited[v]) and  
                         distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for (v,d) in WList[nextv]:  
            if not visited[v]:  
                (distance[v],nbr[v]) = (min(distance[v],d),nextv)  
    return(nbr)
```

Improved implementation — complexity

- Now the scan to find the next vertex to add is $O(n)$

```
def primlist2(WList):
    infinity = 1 + max([d for u in WList.keys()
                        for (v,d) in WList[u]])
    (visited,distance,nbr) = ({}, {}, {})
    for v in WList.keys():
        (visited[v],distance[v],nbr[v]) = (False,infinity,-1)
    visited[0] = True
    for (v,d) in WList[0]:
        (distance[v],nbr[v]) = (d,0)
    for i in range(1,len(WList.keys())):
        nextd = min([distance[v] for v in WList.keys()
                     if not visited[v]])
        nextvlist = [v for v in WList.keys()
                     if (not visited[v]) and
                        distance[v] == nextd]
        if nextvlist == []:
            break
        nextv = min(nextvlist)
        visited[nextv] = True
        for (v,d) in WList[nextv]:
            if not visited[v]:
                (distance[v],nbr[v]) = (min(distance[v],d),nextv)
return(nbr)
```

Improved implementation — complexity

- Now the scan to find the next vertex to add is $O(n)$
- Very similar to Dijkstra's algorithm, except for the update rule for distance

```
def primlist2(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
    (visited,distance,nbr) = ({},{},{})  
    for v in WList.keys():  
        (visited[v],distance[v],nbr[v]) = (False,infinity,-1)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        (distance[v],nbr[v]) = (d,0)  
    for i in range(1,len(WList.keys())):  
        nextd = min([distance[v] for v in WList.keys()  
                     if not visited[v]])  
        nextvlist = [v for v in WList.keys()  
                     if (not visited[v]) and  
                         distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for (v,d) in WList[nextv]:  
            if not visited[v]:  
                (distance[v],nbr[v]) = (min(distance[v],d),nextv)  
    return(nbr)
```

Improved implementation — complexity

- Now the scan to find the next vertex to add is $O(n)$
- Very similar to Dijkstra's algorithm, except for the update rule for distance
- Like Dijkstra's algorithm, this is still $O(n^2)$ even for adjacency lists

```
def primlist2(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
    (visited,distance,nbr) = ({},{},{})  
    for v in WList.keys():  
        (visited[v],distance[v],nbr[v]) = (False,infinity,-1)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        (distance[v],nbr[v]) = (d,0)  
    for i in range(1,len(WList.keys())):  
        nextd = min([distance[v] for v in WList.keys()  
                     if not visited[v]])  
        nextvlist = [v for v in WList.keys()  
                     if (not visited[v]) and  
                         distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for (v,d) in WList[nextv]:  
            if not visited[v]:  
                (distance[v],nbr[v]) = (min(distance[v],d),nextv)  
    return(nbr)
```

Improved implementation — complexity

- Now the scan to find the next vertex to add is $O(n)$
- Very similar to Dijkstra's algorithm, except for the update rule for distance
- Like Dijkstra's algorithm, this is still $O(n^2)$ even for adjacency lists
- With a more clever data structure to extract the minimum, we can do better

```
def primlist2(WList):  
    infinity = 1 + max([d for u in WList.keys()  
                        for (v,d) in WList[u]])  
    (visited,distance,nbr) = ({}, {}, {})  
    for v in WList.keys():  
        (visited[v],distance[v],nbr[v]) = (False,infinity,-1)  
    visited[0] = True  
    for (v,d) in WList[0]:  
        (distance[v],nbr[v]) = (d,0)  
    for i in range(1,len(WList.keys())):  
        nextd = min([distance[v] for v in WList.keys()  
                     if not visited[v]])  
        nextvlist = [v for v in WList.keys()  
                     if (not visited[v]) and  
                         distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for (v,d) in WList[nextv]:  
            if not visited[v]:  
                (distance[v],nbr[v]) = (min(distance[v],d),nextv)  
    return(nbr)
```

Summary

- Prim's algorithm grows an MCST starting with any vertex
- At each step, connect one more vertex to the tree using minimum cost edge from inside the tree to outside the tree
- Correctness follows from Minimum Separator Lemma
- Implementation similar to Dijkstra's algorithms
 - Update rule for distance is different
- Complexity is $O(n^2)$
 - Even with adjacency lists
 - Bottleneck is identifying unvisited vertex with minimum distance
 - Need a better data structure to identify and remove minimum (or maximum) from a collection

Minimum Cost Spanning Trees: Kruskal's Algorithm

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 5

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V

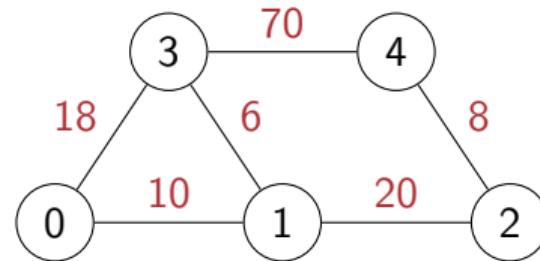
Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- **Strategy 2**
 - Start with n components, each a single vertex
 - Process edges in ascending order of cost
 - Include edge if it does not create a cycle

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- Strategy 2
 - Start with n components, each a single vertex
 - Process edges in ascending order of cost
 - Include edge if it does not create a cycle

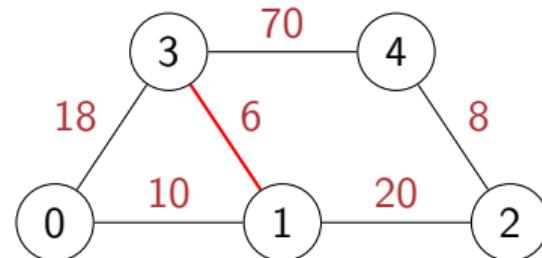
Example



Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- Strategy 2
 - Start with n components, each a single vertex
 - Process edges in ascending order of cost
 - Include edge if it does not create a cycle

Example

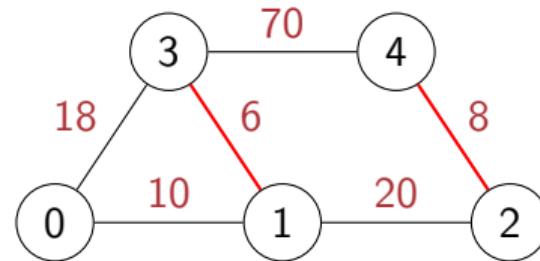


- Start with smallest edge, $(1, 3)$

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- **Strategy 2**
 - Start with n components, each a single vertex
 - Process edges in ascending order of cost
 - Include edge if it does not create a cycle

Example

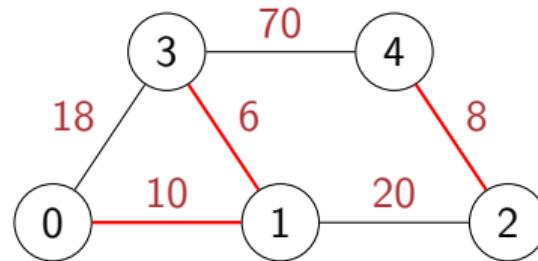


- Start with smallest edge, $(1, 3)$
- Add next smallest edge, $(2, 4)$

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- **Strategy 2**
 - Start with n components, each a single vertex
 - Process edges in ascending order of cost
 - Include edge if it does not create a cycle

Example

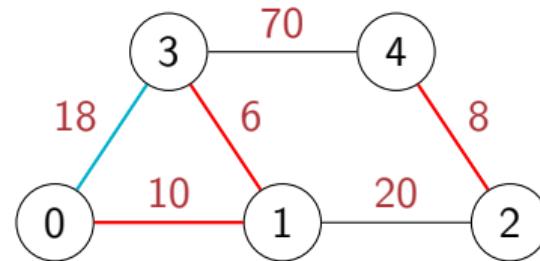


- Start with smallest edge, $(1, 3)$
- Add next smallest edge, $(2, 4)$
- Add next smallest edge, $(0, 1)$

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- **Strategy 2**
 - Start with n components, each a single vertex
 - Process edges in ascending order of cost
 - Include edge if it does not create a cycle

Example

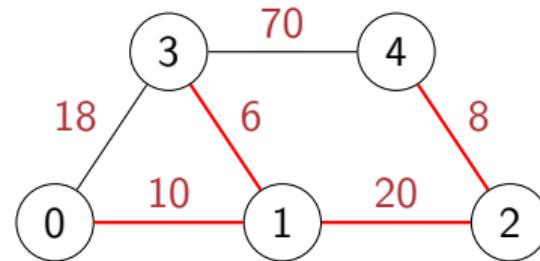


- Start with smallest edge, $(1, 3)$
- Add next smallest edge, $(2, 4)$
- Add next smallest edge, $(0, 1)$
- Can't add $(0, 3)$, forms a cycle

Minimum cost spanning tree (MCST)

- Weighted undirected graph,
 $G = (V, E), W : E \rightarrow \mathbb{R}$
 - G assumed to be connected
- Find a minimum cost **spanning tree**
 - Tree connecting all vertices in V
- **Strategy 2**
 - Start with n components, each a single vertex
 - Process edges in ascending order of cost
 - Include edge if it does not create a cycle

Example



- Start with smallest edge, $(1, 3)$
- Add next smallest edge, $(2, 4)$
- Add next smallest edge, $(0, 1)$
- Can't add $(0, 3)$, forms a cycle
- Add next smallest edge, $(1, 2)$

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$

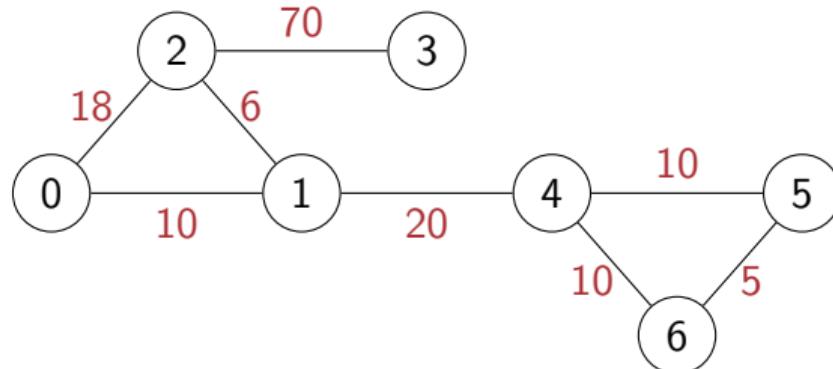
Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$
- Scan E from e_0 to e_{m-1}
 - If adding e_i to TE creates a loop, skip it
 - Otherwise, add e_i to TE

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$
- Scan E from e_0 to e_{m-1}
 - If adding e_i to TE creates a loop, skip it
 - Otherwise, add e_i to TE

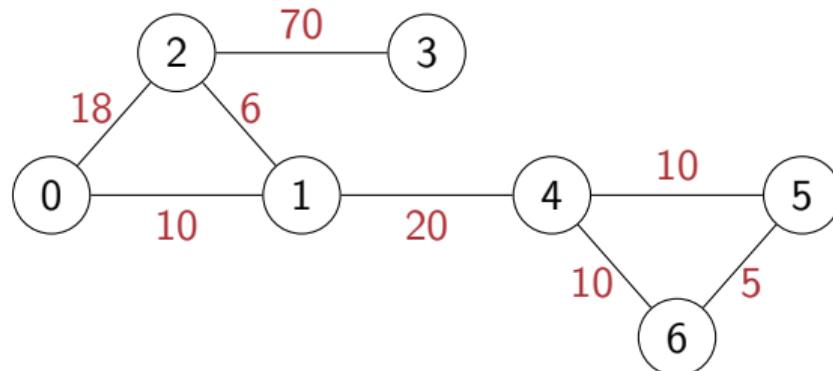
Example



Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$
- Scan E from e_0 to e_{m-1}
 - If adding e_i to TE creates a loop, skip it
 - Otherwise, add e_i to TE

Example



Sort E as

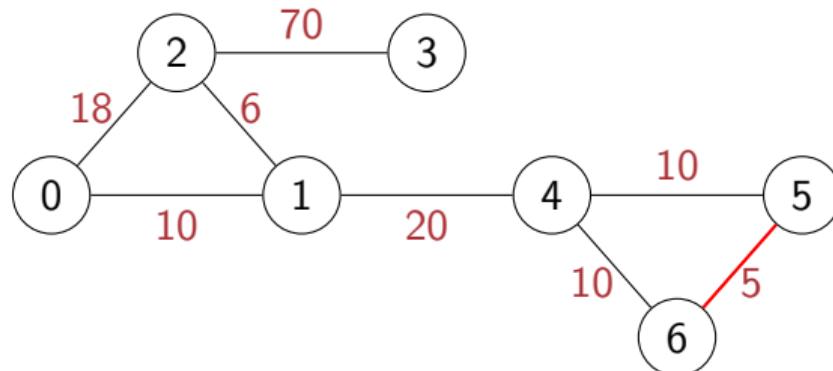
$\{(5, 6), (1, 2), (0, 1), (4, 5), (4, 6), (0, 2), (1, 4), (2, 3)\}$

Set $TE = \emptyset$

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$
- Scan E from e_0 to e_{m-1}
 - If adding e_i to TE creates a loop, skip it
 - Otherwise, add e_i to TE

Example



Sort E as

$\{(5, 6), (1, 2), (0, 1), (4, 5), (4, 6), (0, 2), (1, 4), (2, 3)\}$

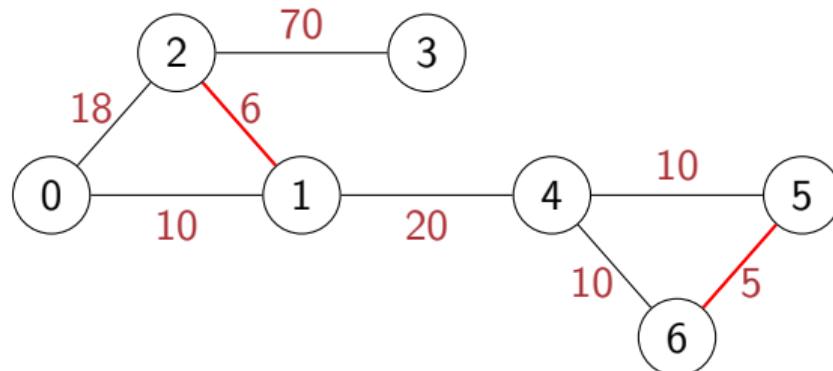
Add $(5, 6)$

Set $TE = \{(5, 6)\}$

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$
- Scan E from e_0 to e_{m-1}
 - If adding e_i to TE creates a loop, skip it
 - Otherwise, add e_i to TE

Example



Sort E as

$\{(5, 6), (1, 2), (0, 1), (4, 5), (4, 6), (0, 2), (1, 4), (2, 3)\}$

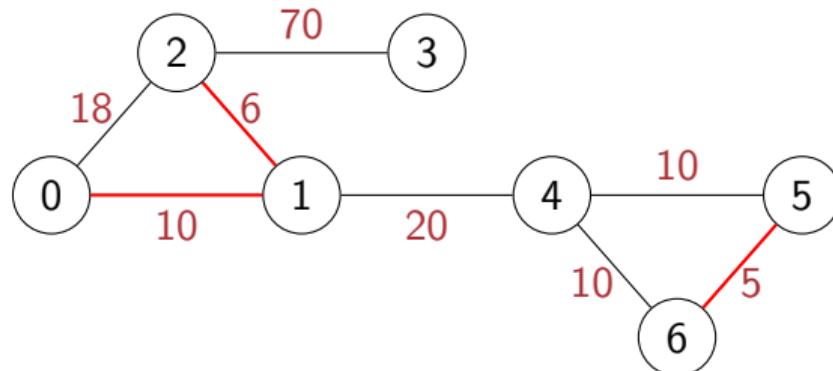
Add $(1, 2)$

Set $TE = \{(5, 6), (1, 2)\}$

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$
- Scan E from e_0 to e_{m-1}
 - If adding e_i to TE creates a loop, skip it
 - Otherwise, add e_i to TE

Example



Sort E as

$\{(5, 6), (1, 2), (0, 1), (4, 5), (4, 6), (0, 2), (1, 4), (2, 3)\}$

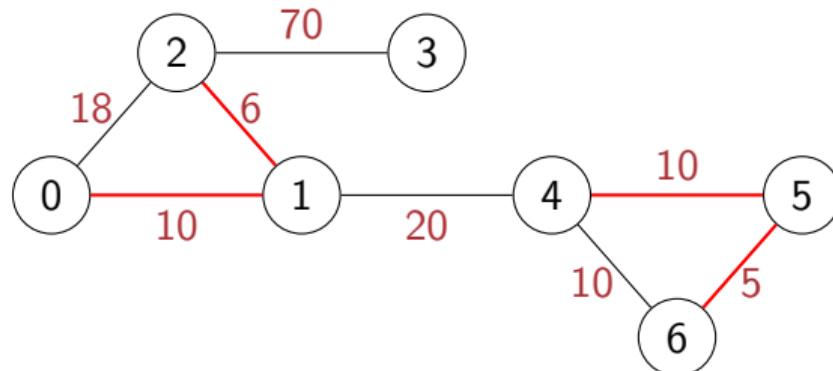
Add $(0, 1)$

Set $TE = \{(5, 6), (1, 2), (0, 1)\}$

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$
- Scan E from e_0 to e_{m-1}
 - If adding e_i to TE creates a loop, skip it
 - Otherwise, add e_i to TE

Example



Sort E as

$\{(5, 6), (1, 2), (0, 1), (4, 5), (4, 6), (0, 2), (1, 4), (2, 3)\}$

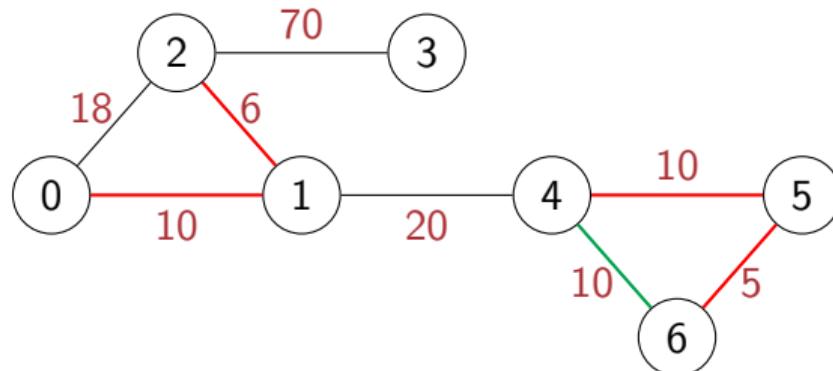
Add $(4, 5)$

Set $TE = \{(5, 6), (1, 2), (0, 1), (4, 5)\}$

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$
- Scan E from e_0 to e_{m-1}
 - If adding e_i to TE creates a loop, skip it
 - Otherwise, add e_i to TE

Example



Sort E as

$\{(5, 6), (1, 2), (0, 1), (4, 5), (4, 6), (0, 2), (1, 4), (2, 3)\}$

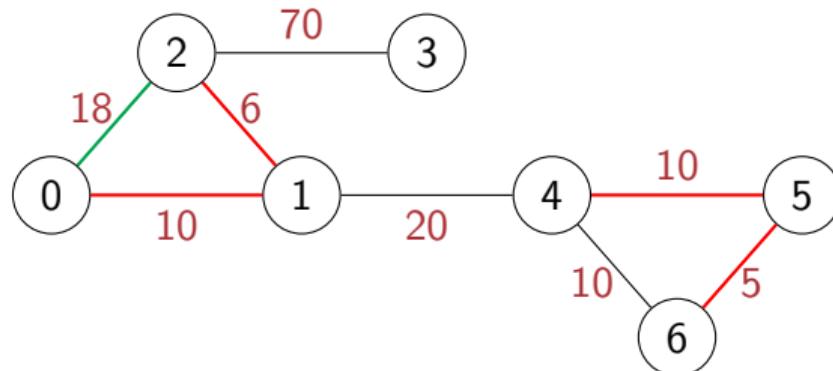
Skip (4, 6)

Set $TE = \{(5, 6), (1, 2), (0, 1), (4, 5)\}$

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$
- Scan E from e_0 to e_{m-1}
 - If adding e_i to TE creates a loop, skip it
 - Otherwise, add e_i to TE

Example



Sort E as

$$\{(5, 6), (1, 2), (0, 1), (4, 5), (4, 6), (0, 2), (1, 4), (2, 3)\}$$

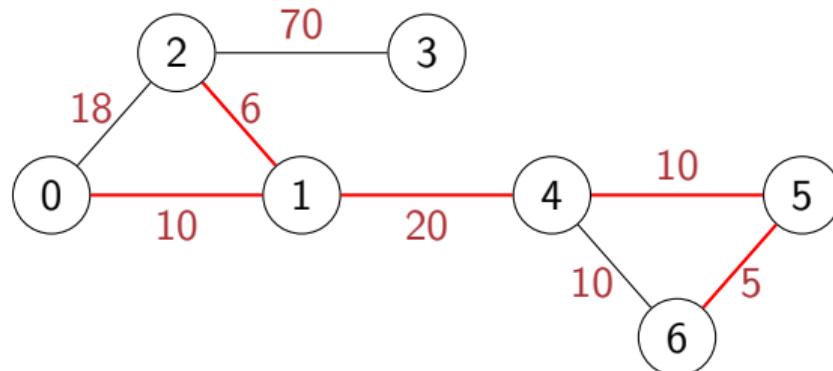
Skip (0, 2)

Set $TE = \{(5, 6), (1, 2), (0, 1), (4, 5)\}$

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$
- Scan E from e_0 to e_{m-1}
 - If adding e_i to TE creates a loop, skip it
 - Otherwise, add e_i to TE

Example



Sort E as

$$\{(5, 6), (1, 2), (0, 1), (4, 5), (4, 6), (0, 2), (1, 4), (2, 3)\}$$

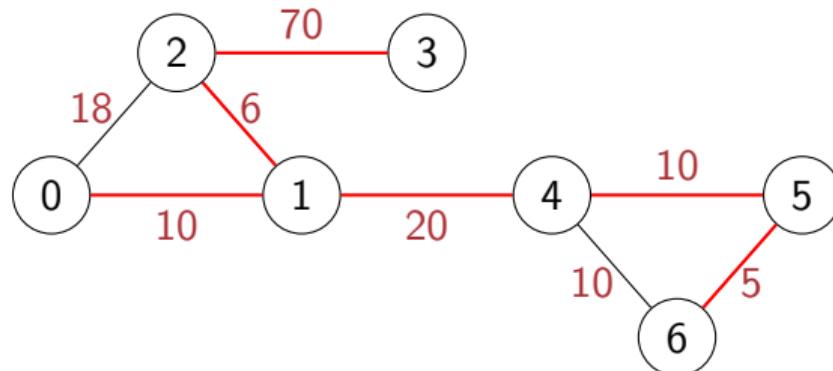
Add $(1, 4)$

Set $TE = \{(5, 6), (1, 2), (0, 1), (4, 5), (1, 4)\}$

Kruskal's algorithm

- $G = (V, E)$, $W : E \rightarrow \mathbb{R}$
- Let $E = \{e_0, e_1, \dots, e_{m-1}\}$ be edges sorted in ascending order by weight
- Let $TE \subseteq E$ be the set of tree edges already added to MCST
- Initially, $TE = \emptyset$
- Scan E from e_0 to e_{m-1}
 - If adding e_i to TE creates a loop, skip it
 - Otherwise, add e_i to TE

Example



Sort E as

$\{(5, 6), (1, 2), (0, 1), (4, 5), (4, 6), (0, 2), (1, 4), (2, 3)\}$

Add $(2, 3)$

Set $TE = \{(5, 6), (1, 2), (0, 1), (4, 5), (1, 4), (2, 3)\}$

Correctness of Kruskal's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
- Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
- Every MCST must include e

Correctness of Kruskal's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
 - Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
 - Every MCST must include e
-
- Edges in TE partition vertices into connected components
 - Initially each vertex is a separate component

Correctness of Kruskal's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
 - Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
 - Every MCST must include e
-
- Edges in TE partition vertices into connected components
 - Initially each vertex is a separate component
 - Adding $e = (u, w)$ merges components of u and w
 - If u and w are in the same component, e forms a cycle and is discarded

Correctness of Kruskal's algorithm

Minimum Separator Lemma

- Let V be partitioned into two non-empty sets U and $W = V \setminus U$
 - Let $e = (u, w)$ be the minimum cost edge with $u \in U, w \in W$
 - Every MCST must include e
-
- Edges in TE partition vertices into connected components
 - Initially each vertex is a separate component
 - Adding $e = (u, w)$ merges components of u and w
 - If u and w are in the same component, e forms a cycle and is discarded
 - Let U be component of u , W be $V \setminus U$
 - U, W form a partition of V with $u \in U$ and $w \in W$
 - Since we are scanning edges in ascending order of cost, e is minimum cost edge connecting U and W , so it must be part of any MCST

Implementing Kruskal's algorithm

- Collect edges in a list as (d, u, v)
 - Weight as first component for easy sorting

```
def kruskal(WList):
    (edges, component, TE) = ([], {}, [])
    for u in WList.keys():
        # Weight as first component to sort easily
        edges.extend([(d,u,v) for (v,d) in WList[u]])
        component[u] = u
    edges.sort()

    for (d,u,v) in edges:
        if component[u] != component[v]:
            TE.append((u,v))
            c = component[u]
            for w in WList.keys():
                if component[w] == c:
                    component[w] = component[v]
    return(TE)
```

Implementing Kruskal's algorithm

- Collect edges in a list as (d, u, v)
 - Weight as first component for easy sorting
- Main challenge is to keep track of connected components
 - Dictionary to record component of each vertex
 - Initially each vertex is an isolated component
 - When we add an edge (u, v) , merge the components of u and v

```
def kruskal(WList):  
    (edges, component, TE) = ([], {}, [])  
    for u in WList.keys():  
        # Weight as first component to sort easily  
        edges.extend([(d,u,v) for (v,d) in WList[u]])  
        component[u] = u  
    edges.sort()  
  
    for (d,u,v) in edges:  
        if component[u] != component[v]:  
            TE.append((u,v))  
            c = component[u]  
            for w in WList.keys():  
                if component[w] == c:  
                    component[w] = component[v]  
    return(TE)
```

Implementing Kruskal's algorithm

Analysis

- Sorting the edges is $O(m \log m)$

- Since m is at most n^2 ,
equivalently $O(m \log n)$

```
def kruskal(WList):
    (edges, component, TE) = ([], {}, [])
    for u in WList.keys():
        # Weight as first component to sort easily
        edges.extend([(d,u,v) for (v,d) in WList[u]])
        component[u] = u
    edges.sort()

    for (d,u,v) in edges:
        if component[u] != component[v]:
            TE.append((u,v))
            c = component[u]
            for w in WList.keys():
                if component[w] == c:
                    component[w] = component[v]
    return(TE)
```

Implementing Kruskal's algorithm

Analysis

- Sorting the edges is $O(m \log m)$
 - Since m is at most n^2 , equivalently $O(m \log n)$
- Outer loop runs m times
 - Each time we add a tree edge, we have to merge components — $O(n)$ scan
 - $n - 1$ tree edges, so this is done $O(n)$ times

```
def kruskal(WList):  
    (edges, component, TE) = ([], {}, [])  
    for u in WList.keys():  
        # Weight as first component to sort easily  
        edges.extend([(d,u,v) for (v,d) in WList[u]])  
        component[u] = u  
    edges.sort()  
  
    for (d,u,v) in edges:  
        if component[u] != component[v]:  
            TE.append((u,v))  
            c = component[u]  
            for w in WList.keys():  
                if component[w] == c:  
                    component[w] = component[v]  
  
    return(TE)
```

Implementing Kruskal's algorithm

Analysis

- Sorting the edges is $O(m \log m)$
 - Since m is at most n^2 , equivalently $O(m \log n)$
- Outer loop runs m times
 - Each time we add a tree edge, we have to merge components — $O(n)$ scan
 - $n - 1$ tree edges, so this is done $O(n)$ times
- Overall, $O(n^2)$

```
def kruskal(WList):  
    (edges, component, TE) = ([], {}, [])  
    for u in WList.keys():  
        # Weight as first component to sort easily  
        edges.extend([(d,u,v) for (v,d) in WList[u]])  
        component[u] = u  
    edges.sort()  
  
    for (d,u,v) in edges:  
        if component[u] != component[v]:  
            TE.append((u,v))  
            c = component[u]  
            for w in WList.keys():  
                if component[w] == c:  
                    component[w] = component[v]  
    return(TE)
```

Implementing Kruskal's algorithm

- Complexity is $O(n^2)$
- Bottleneck is naive strategy to label and merge components

```
def kruskal(WList):  
    (edges, component, TE) = ([], {}, [])  
    for u in WList.keys():  
        # Weight as first component to sort easily  
        edges.extend([(d,u,v) for (v,d) in WList[u]])  
        component[u] = u  
    edges.sort()  
  
    for (d,u,v) in edges:  
        if component[u] != component[v]:  
            TE.append((u,v))  
            c = component[u]  
            for w in WList.keys():  
                if component[w] == c:  
                    component[w] = component[v]  
    return(TE)
```

Implementing Kruskal's algorithm

- Complexity is $O(n^2)$
- Bottleneck is naive strategy to label and merge components
- Components **partition** vertices
 - Collection of disjoint sets

```
def kruskal(WList):  
    (edges, component, TE) = ([], {}, [])  
    for u in WList.keys():  
        # Weight as first component to sort easily  
        edges.extend([(d,u,v) for (v,d) in WList[u]])  
        component[u] = u  
    edges.sort()  
  
    for (d,u,v) in edges:  
        if component[u] != component[v]:  
            TE.append((u,v))  
            c = component[u]  
            for w in WList.keys():  
                if component[w] == c:  
                    component[w] = component[v]  
    return(TE)
```

Implementing Kruskal's algorithm

- Complexity is $O(n^2)$
- Bottleneck is naive strategy to label and merge components
- Components **partition** vertices
 - Collection of disjoint sets
- Data structure to maintain collection of disjoint sets
 - find(v)** — return set containing **v**
 - union(u,v)** — merge sets of **u, v**

```
def kruskal(WList):  
    edges, component, TE = ([], {}, [])  
    for u in WList.keys():  
        # Weight as first component to sort easily  
        edges.extend([(d,u,v) for (v,d) in WList[u]])  
        component[u] = u  
    edges.sort()  
  
    for (d,u,v) in edges:  
        if component[u] != component[v]:  
            TE.append((u,v))  
            c = component[u]  
            for w in WList.keys():  
                if component[w] == c:  
                    component[w] = component[v]  
    return(TE)
```

Implementing Kruskal's algorithm

- Complexity is $O(n^2)$
- Bottleneck is naive strategy to label and merge components
- Components **partition** vertices
 - Collection of disjoint sets
- Data structure to maintain collection of disjoint sets
 - find(v)** — return set containing **v**
 - union(u,v)** — merge sets of **u, v**
- Efficient **union-find** brings complexity down to $O(m \log n)$

```
def kruskal(WList):  
    edges, component, TE = ([], {}, [])  
    for u in WList.keys():  
        # Weight as first component to sort easily  
        edges.extend([(d,u,v) for (v,d) in WList[u]])  
        component[u] = u  
    edges.sort()  
  
    for (d,u,v) in edges:  
        if component[u] != component[v]:  
            TE.append((u,v))  
            c = component[u]  
            for w in WList.keys():  
                if component[w] == c:  
                    component[w] = component[v]  
    return(TE)
```

Summary

- Kruskal's algorithm builds an MCST bottom up
 - Start with n components, each an isolated vertex
 - Scan edges in ascending order of cost
 - Whenever an edge merges disjoint components, add it to the MCST

Summary

- Kruskal's algorithm builds an MCST bottom up
 - Start with n components, each an isolated vertex
 - Scan edges in ascending order of cost
 - Whenever an edge merges disjoint components, add it to the MCST
- Correctness follows from Minimum Separator Lemma

Summary

- Kruskal's algorithm builds an MCST bottom up
 - Start with n components, each an isolated vertex
 - Scan edges in ascending order of cost
 - Whenever an edge merges disjoint components, add it to the MCST
- Correctness follows from Minimum Separator Lemma
- Complexity is $O(n^2)$ due to naive handling of components
 - Will see how to improve to $O(m \log n)$

Summary

- Kruskal's algorithm builds an MCST bottom up
 - Start with n components, each an isolated vertex
 - Scan edges in ascending order of cost
 - Whenever an edge merges disjoint components, add it to the MCST
- Correctness follows from Minimum Separator Lemma
- Complexity is $O(n^2)$ due to naive handling of components
 - Will see how to improve to $O(m \log n)$
- If edge weights repeat, MCST is not unique

Summary

- Kruskal's algorithm builds an MCST bottom up
 - Start with n components, each an isolated vertex
 - Scan edges in ascending order of cost
 - Whenever an edge merges disjoint components, add it to the MCST
- Correctness follows from Minimum Separator Lemma
- Complexity is $O(n^2)$ due to naive handling of components
 - Will see how to improve to $O(m \log n)$
- If edge weights repeat, MCST is not unique
- “Choose minimum cost edge” will allow choices
 - Consider a triangle on 3 vertices with all edges equal

Summary

- Kruskal's algorithm builds an MCST bottom up
 - Start with n components, each an isolated vertex
 - Scan edges in ascending order of cost
 - Whenever an edge merges disjoint components, add it to the MCST
- Correctness follows from Minimum Separator Lemma
- Complexity is $O(n^2)$ due to naive handling of components
 - Will see how to improve to $O(m \log n)$
- If edge weights repeat, MCST is not unique
- “Choose minimum cost edge” will allow choices
 - Consider a triangle on 3 vertices with all edges equal
- Different choices lead to different spanning trees

Summary

- Kruskal's algorithm builds an MCST bottom up
 - Start with n components, each an isolated vertex
 - Scan edges in ascending order of cost
 - Whenever an edge merges disjoint components, add it to the MCST
- Correctness follows from Minimum Separator Lemma
- Complexity is $O(n^2)$ due to naive handling of components
 - Will see how to improve to $O(m \log n)$
- If edge weights repeat, MCST is not unique
- “Choose minimum cost edge” will allow choices
 - Consider a triangle on 3 vertices with all edges equal
- Different choices lead to different spanning trees
- In general, there may be a very large number of minimum cost spanning trees

Union-Find data structure

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 6

Kruskal's algorithm for minimum cost spanning tree (MCST)

- Process edges in ascending order of cost
- If edge (u, v) does not create a cycle,
add it
 - (u, v) can be added if u and v are in
different components
 - Adding edge (u, v) merges these
components
- How can we keep track of components
and merge them efficiently?

Kruskal's algorithm for minimum cost spanning tree (MCST)

- Process edges in ascending order of cost
 - If edge (u, v) does not create a cycle, add it
 - (u, v) can be added if u and v are in different components
 - Adding edge (u, v) merges these components
 - How can we keep track of components and merge them efficiently?
- Components **partition** vertices
 - Collection of disjoint sets
 - Need data structure to maintain collection of disjoint sets
 - `find(v)` — return set containing v
 - `union(u,v)` — merge sets of u, v

Union-Find data structure

- A set S partitioned into components $\{C_1, C_2, \dots, C_k\}$
 - Each $s \in S$ belongs to exactly one C_j

Union-Find data structure

- A set S partitioned into components $\{C_1, C_2, \dots, C_k\}$
 - Each $s \in S$ belongs to exactly one C_j
- Support the following operations
 - `MakeUnionFind(S)` — set up initial singleton components $\{s\}$, for each $s \in S$
 - `Find(s)` — return the component containing s
 - `Union(s, s')` — merges components containing s, s'

Naive implementation

- Assume $S = \{0, 1, \dots, n - 1\}$

Naive implementation

- Assume $S = \{0, 1, \dots, n - 1\}$
- Set up a array/dictionary Component

Naive implementation

- Assume $S = \{0, 1, \dots, n - 1\}$
- Set up a array/dictionary `Component`
- `MakeUnionFind(S)` —
 - Set `Component[i] = i` for each `i`

Naive implementation

- Assume $S = \{0, 1, \dots, n - 1\}$
- Set up a array/dictionary `Component`
- `MakeUnionFind(S)` —
 - Set `Component[i] = i` for each `i`
- `Find(i)`
 - Return `Component[i]`

Naive implementation

- Assume $S = \{0, 1, \dots, n - 1\}$
- Set up a array/dictionary Component
- MakeUnionFind(S) —
 - Set Component[i] = i for each i
- Find(i)
 - Return Component[i]
- Union(i, j)

```
c_old = Component[i]
c_new = Component[j]
for k in range(n):
    if Component[k] == c_old:
        Component[k] = c_new
```

Naive implementation

- Assume $S = \{0, 1, \dots, n - 1\}$
- Set up a array/dictionary `Component`
- `MakeUnionFind(S)` —
 - Set `Component[i] = i` for each i
- `Find(i)`
 - Return `Component[i]`
- `Union(i, j)`

```
c_old = Component[i]
c_new = Component[j]
for k in range(n):
    if Component[k] == c_old:
        Component[k] = c_new
```

Complexity

- `MakeUnionFind(S)` — $O(n)$
- `Find(i)` — $O(1)$
- `Union(i, j)` — $O(n)$
- Sequence of m `Union()` operations takes time $O(mn)$

Improved implementation

- Another array/dictionary **Members**

Improved implementation

- Another array/dictionary `Members`
- For each component `c`, `Members[c]` is a list of its members
- `Size[c] = length(Members[c])` is the number of members

Improved implementation

- Another array/dictionary `Members`
 - For each component `c`, `Members[c]` is a list of its members
 - `Size[c] = length(Members[c])` is the number of members
-
- `MakeUnionFind(S)`
 - Set `Component[i] = i` for all `i`
 - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`

Improved implementation

- Another array/dictionary `Members`
 - For each component c , `Members[c]` is a list of its members
 - `Size[c] = length(Members[c])` is the number of members
- `MakeUnionFind(S)`
 - Set `Component[i] = i` for all i
 - Set `Members[i] = [i], Size[i] = 1` for all i
 - `Find(i)`
 - Return `Component[i]`

Improved implementation

- Another array/dictionary `Members`
- For each component `c`, `Members[c]` is a list of its members
- `Size[c] = len(Members[c])` is the number of members

■ `MakeUnionFind(S)`

- Set `Component[i] = i` for all `i`
- Set `Members[i] = [i]`, `Size[i] = 1` for all `i`

■ `Find(i)`

- Return `Component[i]`

■ `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

Why does this help?

- **MakeUnionFind(S)**

- Set `Component[i] = i` for all `i`
 - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`

- **Find(i)**

- Return `Component[i]`

- **Union(i,j)**

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

Why does this help?

- `MakeUnionFind(S)`

- Set `Component[i] = i` for all `i`
 - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`

- `Find(i)`

- Return `Component[i]`

- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- `Members[c_old]` allows us to merge `Component[i]` into `Component[j]` in time $O(\text{Size}[c_{old}])$ rather than $O(n)$

Why does this help?

- `MakeUnionFind(S)`

- Set `Component[i] = i` for all `i`
 - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`

- `Find(i)`

- Return `Component[i]`

- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- `Members[c_old]` allows us to merge `Component[i]` into `Component[j]` in time $O(\text{Size}[c_{old}])$ rather than $O(n)$

- How can we make use of `Size[c]`

- Always merge smaller component into larger one
 - If `Size[c] < Size[c']` relabel `c` as `c'`, else relabel `c'` as `c`

Why does this help?

- `MakeUnionFind(S)`

- Set `Component[i] = i` for all `i`
 - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`

- `Find(i)`

- Return `Component[i]`

- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- `Members[c_old]` allows us to merge `Component[i]` into `Component[j]` in time $O(\text{Size}[c_{\text{old}}])$ rather than $O(n)$

- How can we make use of `Size[c]`

- Always merge smaller component into larger one
 - If `Size[c] < Size[c']` relabel `c` as `c'`, else relabel `c'` as `c`

- Individual merge operations can still take time $O(n)$

- Both `Size[c]`, `Size[c']` could be about $n/2$
 - More careful accounting

Why does this help?

- `MakeUnionFind(S)`
 - Set `Component[i] = i` for all `i`
 - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
 - `Find(i)`
 - Return `Component[i]`
 - `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```
- Always merge smaller component into larger one

Why does this help?

- `MakeUnionFind(S)`
 - Set `Component[i] = i` for all `i`
 - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Always merge smaller component into larger one`
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled

■ `Find(i)`

- Return `Component[i]`

■ `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

Why does this help?

- `MakeUnionFind(S)`

- Set `Component[i] = i` for all `i`
 - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`

- `Find(i)`

- Return `Component[i]`

- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- Always merge smaller component into larger one

- For each `i`, size of `Component[i]` at least doubles each time it is relabelled

- After m `Union()` operations, at most $2m$ elements have been “touched”

- Size of `Component[i]` is at most $2m$

Why does this help?

- `MakeUnionFind(S)`

- Set `Component[i] = i` for all `i`
 - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`

- `Find(i)`

- Return `Component[i]`

- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- Always merge smaller component into larger one

- For each `i`, size of `Component[i]` at least doubles each time it is relabelled

- After m `Union()` operations, at most $2m$ elements have been “touched”

- Size of `Component[i]` is at most $2m$

- Size of `Component[i]` grows as $1, 2, 4, \dots$, so `i` changes component at most $\log m$ times

Why does this help?

- Always merge smaller component into larger one
- For each i , size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been “touched”
 - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \dots$, so i changes component at most $\log m$ times

Why does this help?

- Always merge smaller component into larger one
- For each i , size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been “touched”
 - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \dots$, so i changes component at most $\log m$ times
- Over m updates
 - At most $2m$ elements are relabelled
 - Each one at most $O(\log m)$ times

Why does this help?

- Always merge smaller component into larger one
- For each i , size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been “touched”
 - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \dots$, so i changes component at most $\log m$ times
- Over m updates
 - At most $2m$ elements are relabelled
 - Each one at most $O(\log m)$ times
- Overall, m `Union()` operations take time $O(m \log m)$

Why does this help?

- Always merge smaller component into larger one
- For each i , size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been “touched”
 - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \dots$, so i changes component at most $\log m$ times
- Over m updates
 - At most $2m$ elements are relabelled
 - Each one at most $O(\log m)$ times
- Overall, m `Union()` operations take time $O(m \log m)$
- Works out to time $O(\log m)$ per `Union()` operation
 - Amortised complexity of `Union()` is $O(\log m)$

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \dots, e_{m-1}\}$ in ascending order

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \dots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex j is in component j

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \dots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex j is in component j
- Adding and edge $e_k = (u, v)$ to the tree
 - Check that `Find(u) != Find(v)`
 - Merge components:
`Union(Component[u], Component[v])`

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \dots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex j is in component j
- Adding and edge $e_k = (u, v)$ to the tree
 - Check that `Find(u) != Find(v)`
 - Merge components:
`Union(Component[u], Component[v])`
- Tree has $n - 1$ edges, so $O(n)$ `Union()` operations
 - $O(n \log n)$ amortised cost, overall

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \dots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex j is in component j
- Adding and edge $e_k = (u, v)$ to the tree
 - Check that `Find(u) != Find(v)`
 - Merge components:
`Union(Component[u], Component[v])`
- Tree has $n - 1$ edges, so $O(n)$ `Union()` operations
 - $O(n \log n)$ amortised cost, overall
- Sorting E takes $O(m \log m)$
 - Equivalently $O(m \log n)$, since $m \leq n^2$

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \dots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex j is in component j
- Adding and edge $e_k = (u, v)$ to the tree
 - Check that `Find(u) != Find(v)`
 - Merge components:
`Union(Component[u], Component[v])`
- Tree has $n - 1$ edges, so $O(n)$ `Union()` operations
 - $O(n \log n)$ amortised cost, overall
- Sorting E takes $O(m \log m)$
 - Equivalently $O(m \log n)$, since $m \leq n^2$
 - Overall time, $O((m + n) \log n)$

Summary

- Implement Union-Find using arrays/dictionaries Component, Member, Size
 - `MakeUnionFind(S)` is $O(n)$
 - `Find(i)` is $O(1)$
 - Across m operations, amortised complexity of each `Union()` operation is $\log m$

Summary

- Implement Union-Find using arrays/dictionaries `Component`, `Member`, `Size`
 - `MakeUnionFind(S)` is $O(n)$
 - `Find(i)` is $O(1)$
 - Across m operations, amortised complexity of each `Union()` operation is $\log m$
- Can also maintain `Members[k]` as a tree rather than as a list
 - `Union()` becomes $O(1)$
 - With clever updates to the tree, `Find()` has amortised complexity very close to $O(1)$

Priority Queues

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 6

Dealing with priorities

Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities

Dealing with priorities

Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it

Dealing with priorities

Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it
- New jobs may join the list at any time

Dealing with priorities

Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it
- New jobs may join the list at any time
- How should the scheduler maintain the list of pending jobs and their priorities?

Dealing with priorities

Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it
- New jobs may join the list at any time
- How should the scheduler maintain the list of pending jobs and their priorities?

Priority queue

- Need to maintain a collection of items with priorities to optimise the following operations
- `delete_max()`
 - Identify and remove item with highest priority
 - Need not be unique
- `insert()`
 - Add a new item to the collection

Implementing priority queues with one dimensional structures

- `delete_max()`

- Identify and remove item with highest priority
- Need not be unique

- `insert()`

- Add a new item to the list

Implementing priority queues with one dimensional structures

- Unsorted list

- `insert()` is $O(1)$
- `delete_max()` is $O(n)$

- `delete_max()`

- Identify and remove item with highest priority
- Need not be unique

- `insert()`

- Add a new item to the list

Implementing priority queues with one dimensional structures

- Unsorted list
 - `insert()` is $O(1)$
 - `delete_max()` is $O(n)$
 - Sorted list
 - `delete_max()` is $O(1)$
 - `insert()` is $O(n)$
-
- `delete_max()`
 - Identify and remove item with highest priority
 - Need not be unique
 - `insert()`
 - Add a new item to the list

Implementing priority queues with one dimensional structures

- Unsorted list
 - `insert()` is $O(1)$
 - `delete_max()` is $O(n)$
- Sorted list
 - `delete_max()` is $O(1)$
 - `insert()` is $O(n)$
- Processing n items requires $O(n^2)$
- `delete_max()`
 - Identify and remove item with highest priority
 - Need not be unique
- `insert()`
 - Add a new item to the list

Moving to two dimensions

First attempt

- Assume N processes enter/leave the queue

$$N = 25$$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

Moving to two dimensions

First attempt

- Assume N processes enter/leave the queue
- Maintain a $\sqrt{N} \times \sqrt{N}$ array

$$N = 25$$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

Moving to two dimensions

First attempt

- Assume N processes enter/leave the queue
- Maintain a $\sqrt{N} \times \sqrt{N}$ array
- Each row is in sorted order

$$N = 25$$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

insert()

- Keep track of the size of each row

$$N = 25$$

3	19	23	35	58	5
12	17	25	43	67	5
10	13	20			3
11	16	28	49		4
6	14				2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
 - Use size of row to determine

$$N = 25$$

3	19	23	35	58	5
12	17	25	43	67	5
10	13	20			3
11	16	28	49		4
6	14				2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
 - Use size of row to determine
- Insert 15

$$N = 25$$

3	19	23	35	58	5
12	17	25	43	67	5
10	13	20			3
11	16	28	49		4
6	14				2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
 - Use size of row to determine
- Insert 15

$$N = 25$$

15	3	19	23	35	58	5
	12	17	25	43	67	5
	10	13	20			3
	11	16	28	49		4
	6	14				2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
 - Use size of row to determine
- Insert 15

$N = 25$

3	19	23	35	58	5
12	17	25	43	67	5
10	13	20			3
11	16	28	49		4
6	14				2

15

insert()

- Keep track of the size of each row
- Insert into the first row that has space
 - Use size of row to determine
- Insert 15

$N = 25$

3	19	23	35	58	5
12	17	25	43	67	5
10	13	20			3
11	16	28	49		4
6	14				2

15

insert()

- Keep track of the size of each row
- Insert into the first row that has space
 - Use size of row to determine
- Insert 15

$$N = 25$$

3	19	23	35	58	5
12	17	25	43	67	5
10	13	15	20		3
11	16	28	49		4
6	14				2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
 - Use size of row to determine
- Insert 15

$$N = 25$$

3	19	23	35	58	5
12	17	25	43	67	5
10	13	15	20		4
11	16	28	49		4
6	14				2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
 - Use size of row to determine
- Insert 15
- Takes time $O(\sqrt{N})$
 - Scan size column to locate row to insert, $O(\sqrt{N})$
 - Insert into the first row with free space, $O(\sqrt{N})$

$$N = 25$$

3	19	23	35	58	5
12	17	25	43	67	5
10	13	15	20		4
11	16	28	49		4
6	14				2

delete_max()

- Maximum in each row is the last element

$$N = 25$$

3	19	23	35	58
12	17	25	43	67
10	13	15	20	
11	16	28	49	
6	14			

5
5
4
4
2

delete_max()

- Maximum in each row is the last element
- Position is available through size column

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	15	20	
11	16	28	49	
6	14			

5
5
4
4
2

delete_max()

- Maximum in each row is the last element
- Position is available through size column
- Identify the maximum amongst these

$$N = 25$$

3	19	23	35	58
12	17	25	43	67
10	13	15	20	
11	16	28	49	
6	14			

5
5
4
4
2

delete_max()

- Maximum in each row is the last element
- Position is available through size column
- Identify the maximum amongst these
- Delete it

$$N = 25$$

3	19	23	35	58
12	17	25	43	
10	13	15	20	
11	16	28	49	
6	14			

5
4
4
4
2

delete_max()

- Maximum in each row is the last element
- Position is available through size column
- Identify the maximum amongst these
- Delete it
- Again $O(\sqrt{N})$
 - Find the maximum among last entries, $O(\sqrt{N})$
 - Delete it, $O(1)$

$N = 25$

3	19	23	35	58
12	17	25	43	
10	13	15	20	
11	16	28	49	
6	14			

5
4
4
4
2

Summary

- 2D $\sqrt{N} \times \sqrt{N}$ array with sorted rows

- `insert()` is $O(\sqrt{N})$
- `delete_max()` is $O(\sqrt{N})$
- Processing N items is $O(N\sqrt{N})$

$$N = 25$$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

Summary

- 2D $\sqrt{N} \times \sqrt{N}$ array with sorted rows
 - `insert()` is $O(\sqrt{N})$
 - `delete_max()` is $O(\sqrt{N})$
 - Processing N items is $O(N\sqrt{N})$
- Can we do better?

$$N = 25$$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

Summary

- 2D $\sqrt{N} \times \sqrt{N}$ array with sorted rows
 - `insert()` is $O(\sqrt{N})$
 - `delete_max()` is $O(\sqrt{N})$
 - Processing N items is $O(N\sqrt{N})$
- Can we do better?
- Maintain a special binary tree — **heap**
 - Height $O(\log N)$
 - `insert()` is $O(\log N)$
 - `delete_max()` is $O(\log N)$
 - Processing N items is $O(N \log N)$

$$N = 25$$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

Summary

- 2D $\sqrt{N} \times \sqrt{N}$ array with sorted rows
 - `insert()` is $O(\sqrt{N})$
 - `delete_max()` is $O(\sqrt{N})$
 - Processing N items is $O(N\sqrt{N})$
- Can we do better?
- Maintain a special binary tree — **heap**
 - Height $O(\log N)$
 - `insert()` is $O(\log N)$
 - `delete_max()` is $O(\log N)$
 - Processing N items is $O(N \log N)$
- Flexible — need not fix N in advance

$$N = 25$$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

Heaps

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 6

Priority queue

- Need to maintain a collection of items with priorities to optimise the following operations
- `delete_max()`
 - Identify and remove item with highest priority
 - Need not be unique
- `insert()`
 - Add a new item to the list

Priority queue

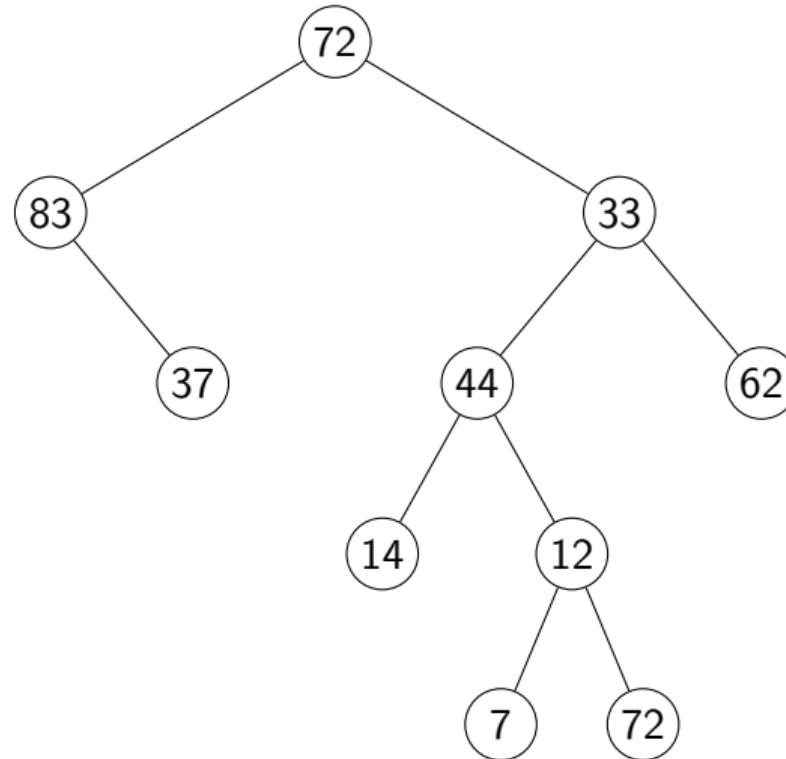
- Need to maintain a collection of items with priorities to optimise the following operations
 - `delete_max()`
 - Identify and remove item with highest priority
 - Need not be unique
 - `insert()`
 - Add a new item to the list
- Maintaining as a list incurs cost $O(N^2)$ across N inserts and deletions

Priority queue

- Need to maintain a collection of items with priorities to optimise the following operations
- `delete_max()`
 - Identify and remove item with highest priority
 - Need not be unique
- `insert()`
 - Add a new item to the list
- Maintaining as a list incurs cost $O(N^2)$ across N inserts and deletions
- Using a $\sqrt{N} \times \sqrt{N}$ array reduces the cost to $O(\sqrt{N})$ per operations
 - $O(N\sqrt{N})$ across N inserts and deletions

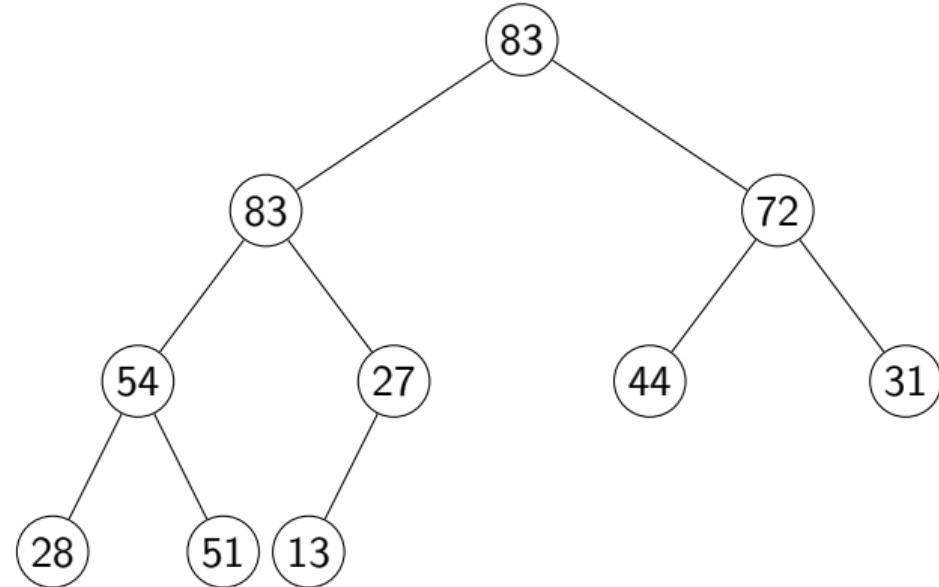
Binary trees

- Values are stored as nodes in a rooted tree
- Each node has up to two children
 - Left child and right child
 - Order is important
- Other than the root, each node has a unique parent
- Leaf node — no children
- Size — number of nodes
- Height — number of levels



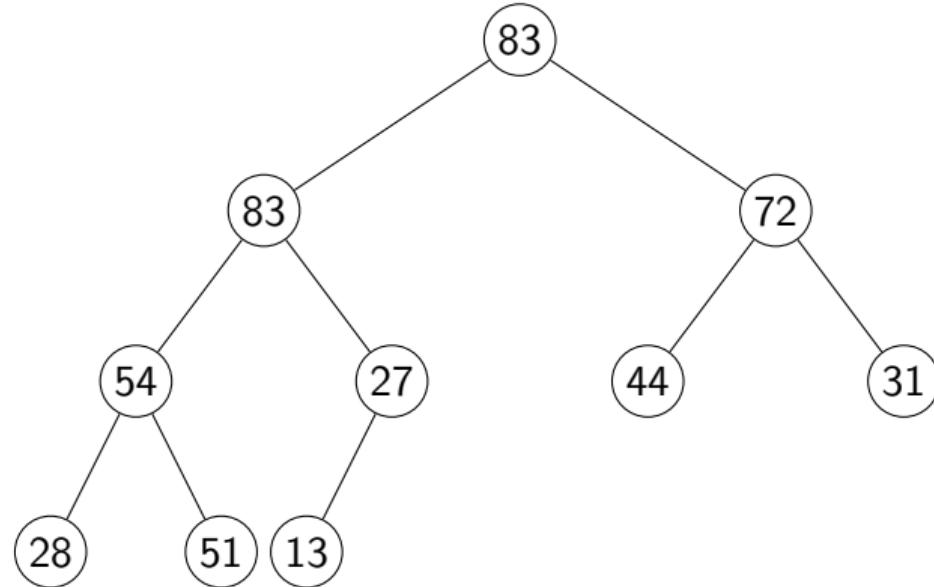
Heap

- Binary tree filled level by level, left to right
- The value at each node is at least as big the values of its children
 - max-heap



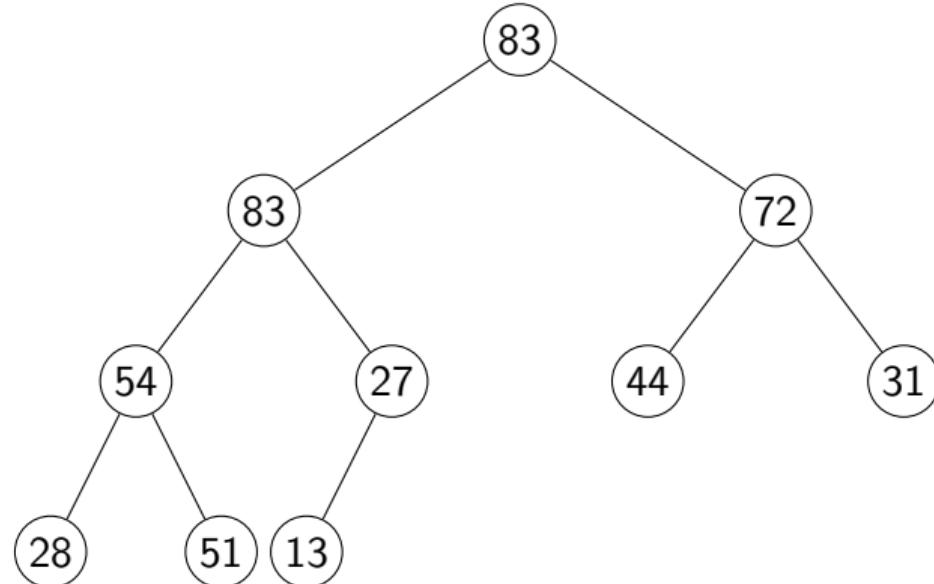
Heap

- Binary tree filled level by level, left to right
- The value at each node is at least as big the values of its children
 - max-heap
- Binary tree on the right is an example of a heap



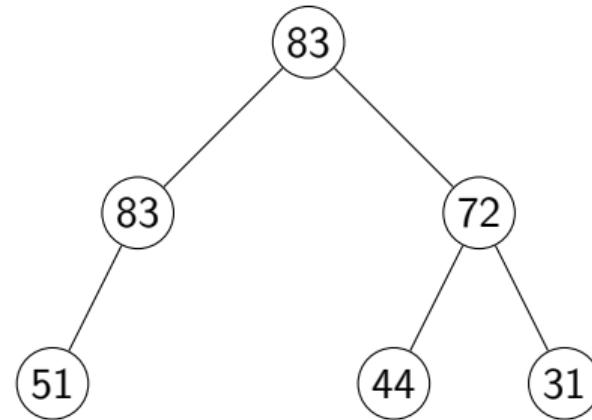
Heap

- Binary tree filled level by level, left to right
- The value at each node is at least as big the values of its children
 - max-heap
- Binary tree on the right is an example of a heap
- Root always has the largest value
 - By induction, because of the max-heap property



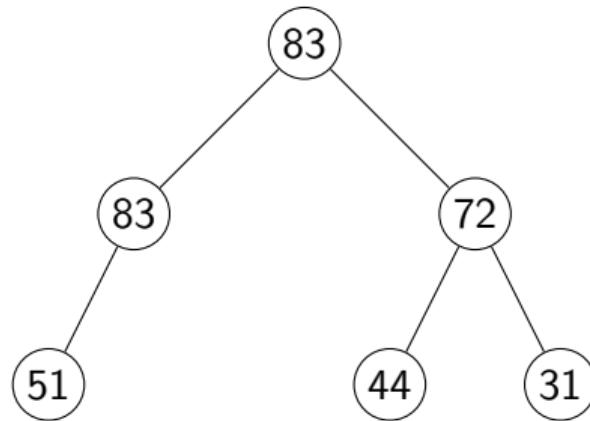
Non-examples

No “holes” allowed

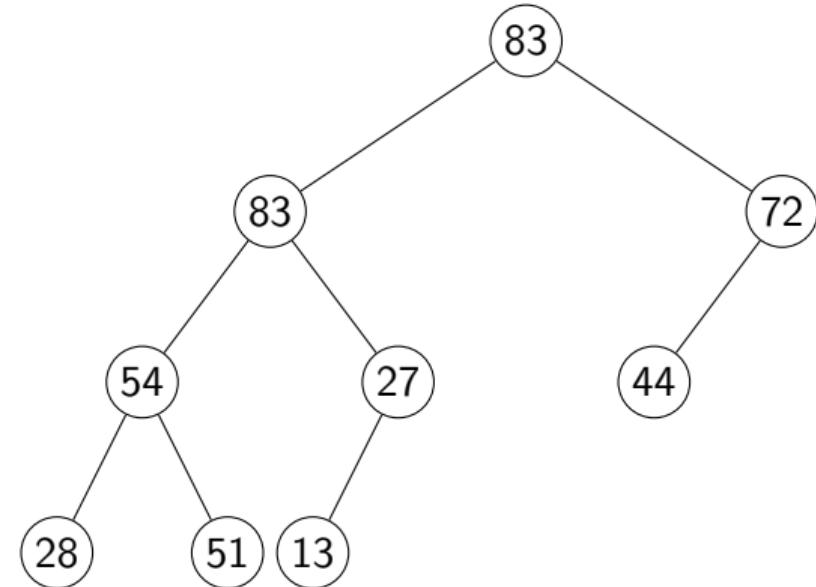


Non-examples

No “holes” allowed

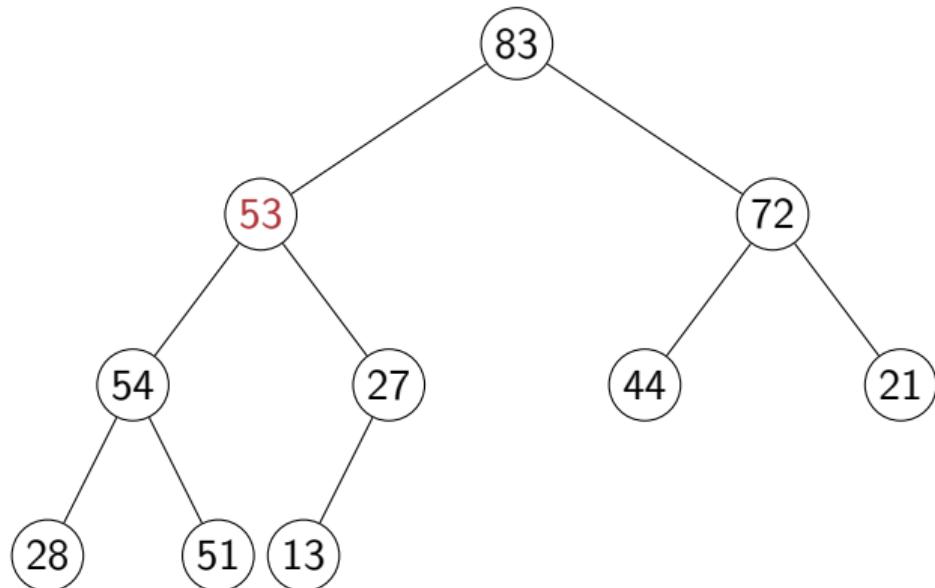


Cannot leave a level incomplete



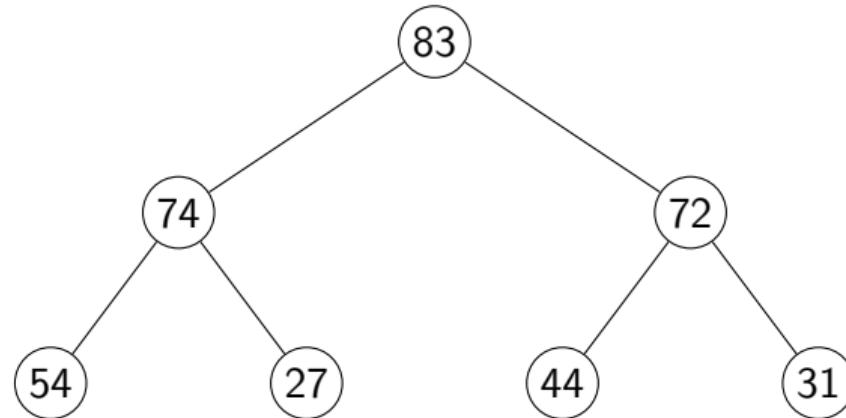
Non-examples

Heap property is violated



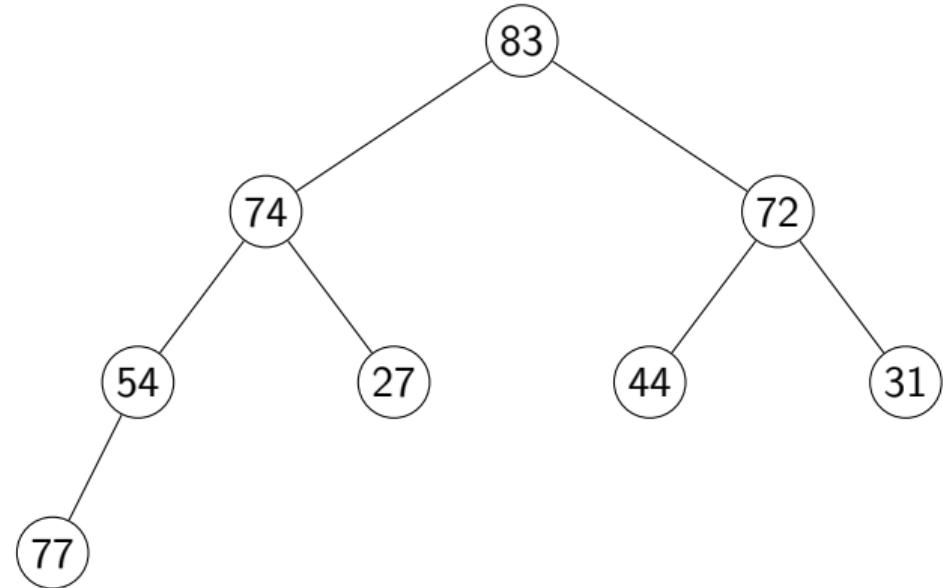
insert()

■ insert(77)



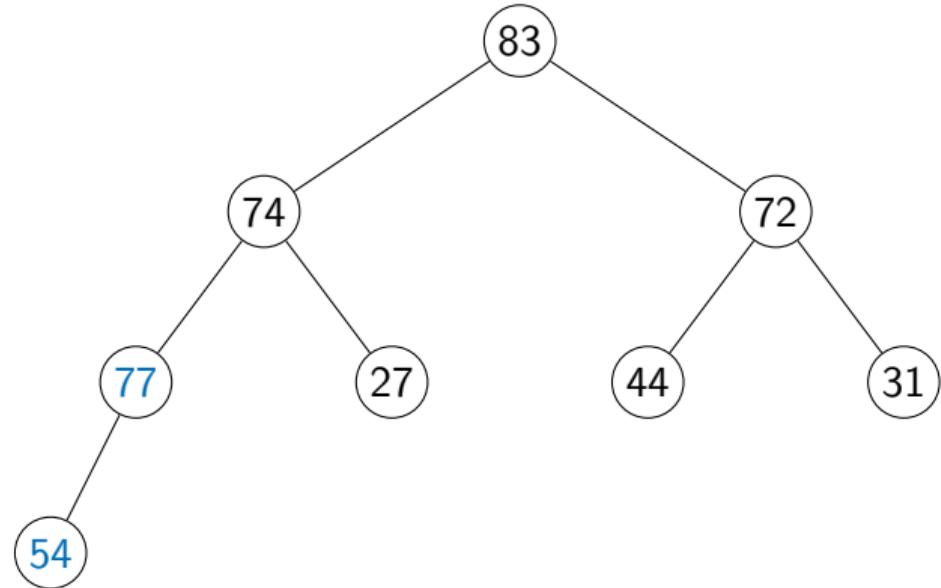
insert()

- **insert(77)**
- Add a new node at dictated by heap structure



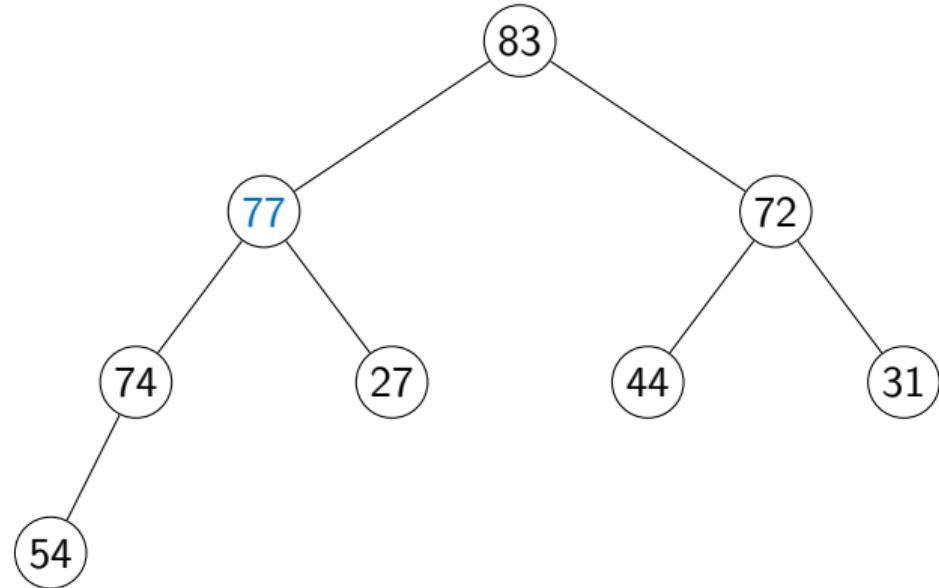
insert()

- **insert(77)**
- Add a new node at dictated by heap structure
- Restore the heap property along path to the root



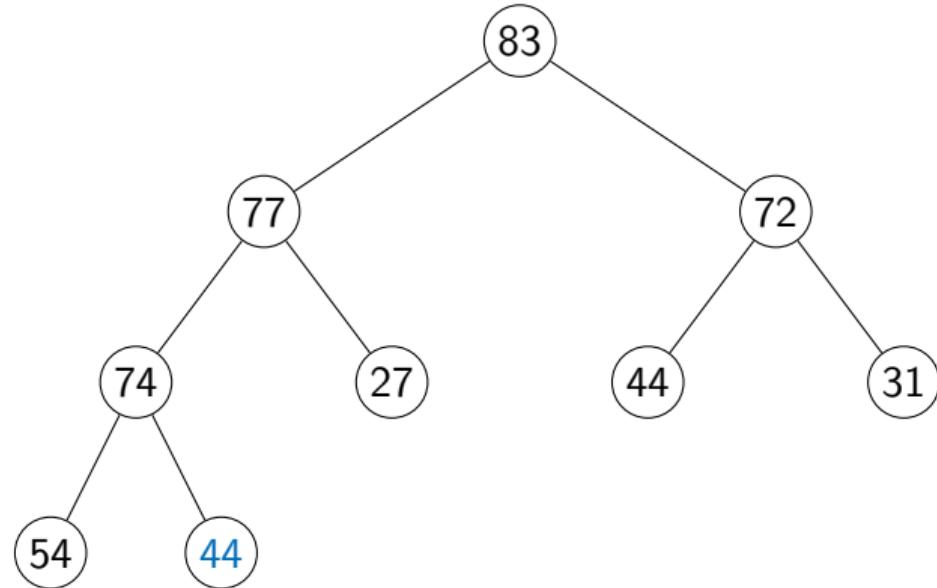
insert()

- **insert(77)**
- Add a new node at dictated by heap structure
- Restore the heap property along path to the root



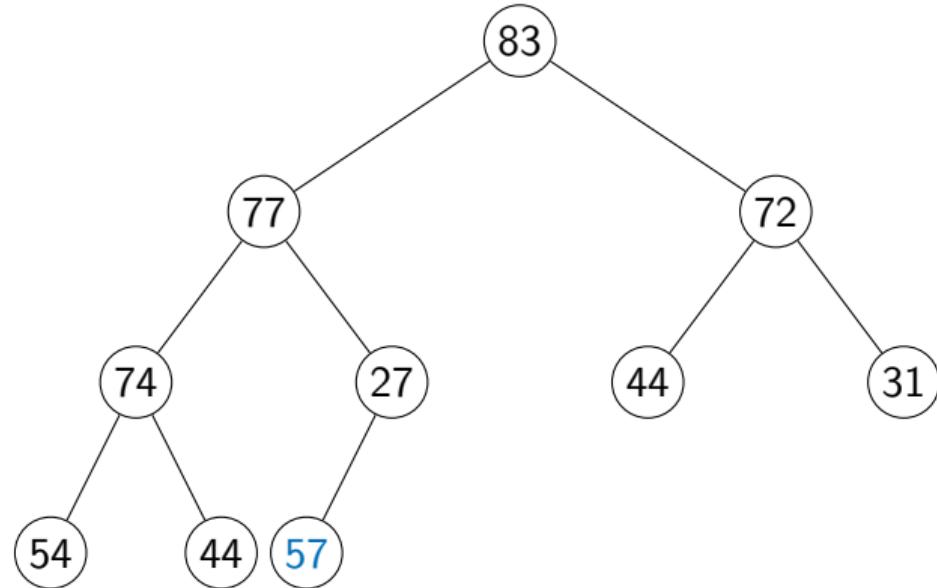
insert()

- `insert(77)`
- Add a new node at dictated by heap structure
- Restore the heap property along path to the root
- `insert(44)`



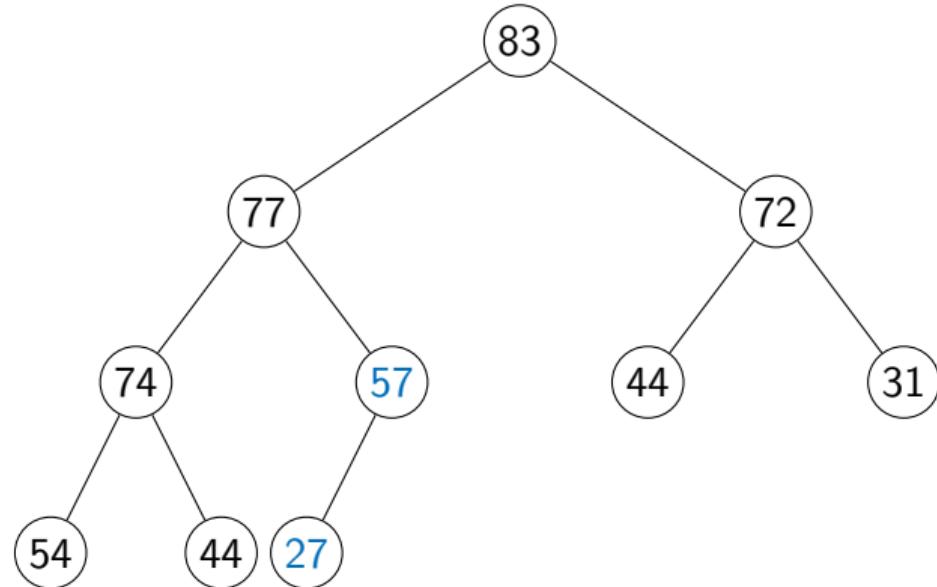
insert()

- `insert(77)`
- Add a new node at dictated by heap structure
- Restore the heap property along path to the root
- `insert(44)`
- `insert(57)`



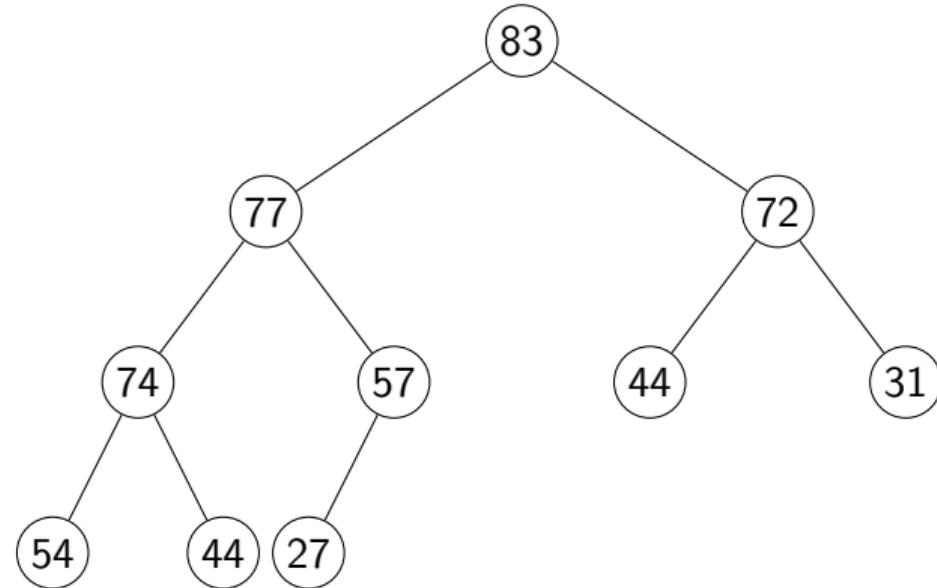
insert()

- `insert(77)`
- Add a new node at dictated by heap structure
- Restore the heap property along path to the root
- `insert(44)`
- `insert(57)`



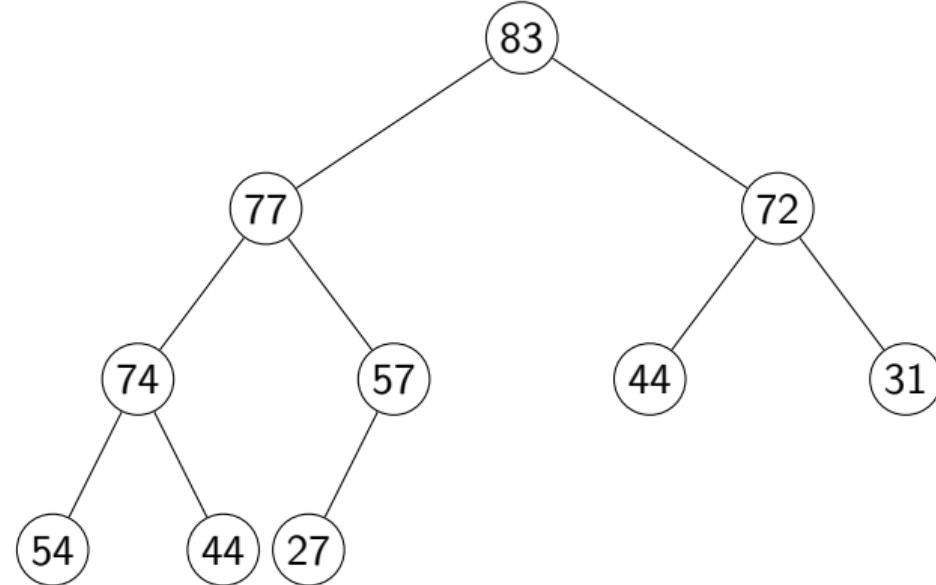
Complexity of insert()

- Need to walk up from the leaf to the root
 - Height of the tree



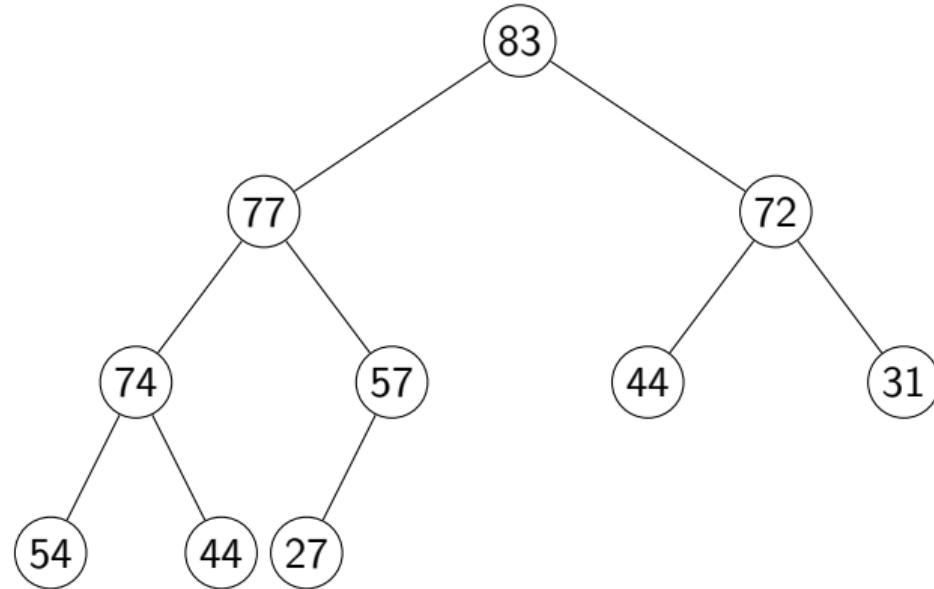
Complexity of insert()

- Need to walk up from the leaf to the root
 - Height of the tree
- Number of nodes at level 0 is $2^0 = 1$



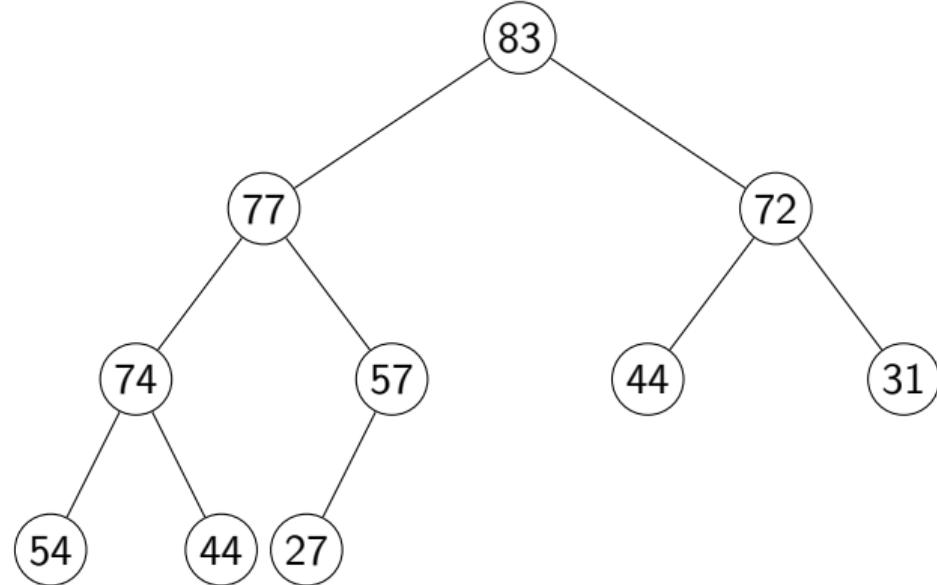
Complexity of insert()

- Need to walk up from the leaf to the root
 - Height of the tree
- Number of nodes at level 0 is $2^0 = 1$
- Number of nodes at level j is 2^j



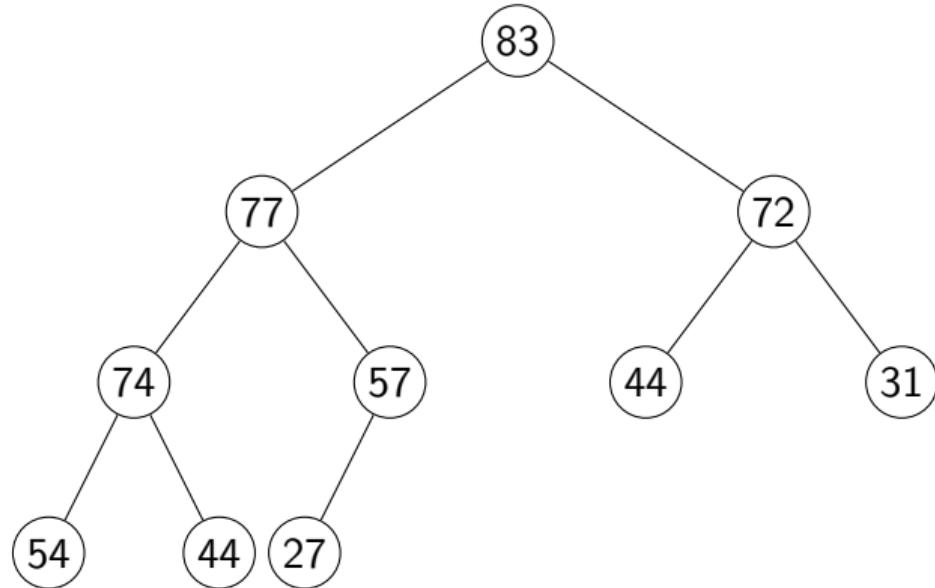
Complexity of insert()

- Need to walk up from the leaf to the root
 - Height of the tree
- Number of nodes at level 0 is $2^0 = 1$
- Number of nodes at level j is 2^j
- If we fill k levels,
$$2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$$
nodes



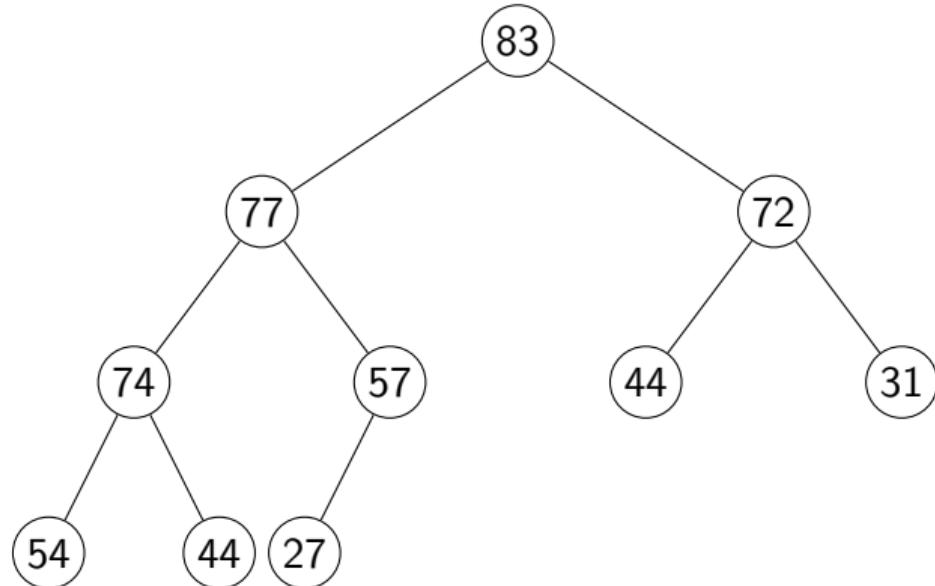
Complexity of insert()

- Need to walk up from the leaf to the root
 - Height of the tree
- Number of nodes at level 0 is $2^0 = 1$
- Number of nodes at level j is 2^j
- If we fill k levels,
$$2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$$
nodes
- If we have N nodes, at most $1 + \log N$ levels



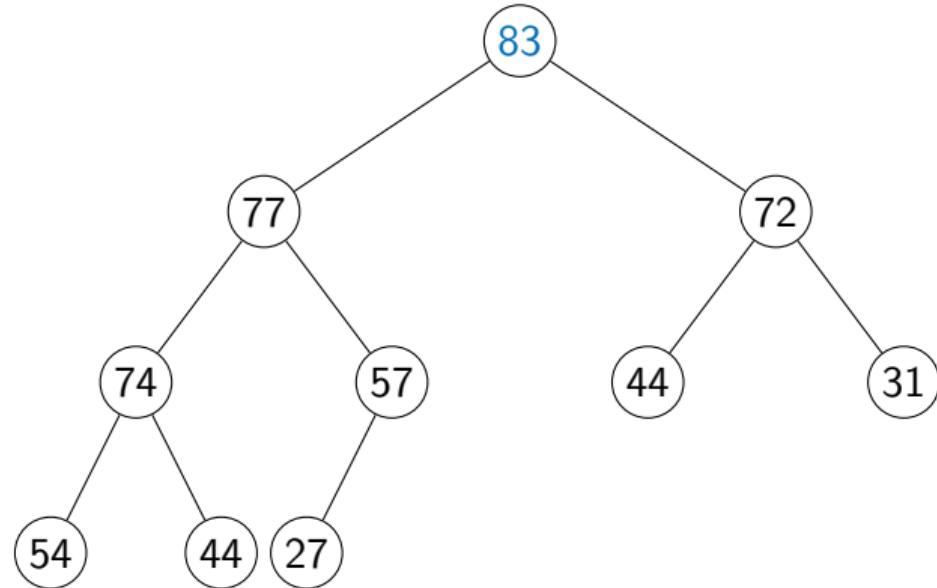
Complexity of insert()

- Need to walk up from the leaf to the root
 - Height of the tree
- Number of nodes at level 0 is $2^0 = 1$
- Number of nodes at level j is 2^j
- If we fill k levels,
$$2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$$
nodes
- If we have N nodes, at most $1 + \log N$ levels
- `insert()` is $O(\log N)$



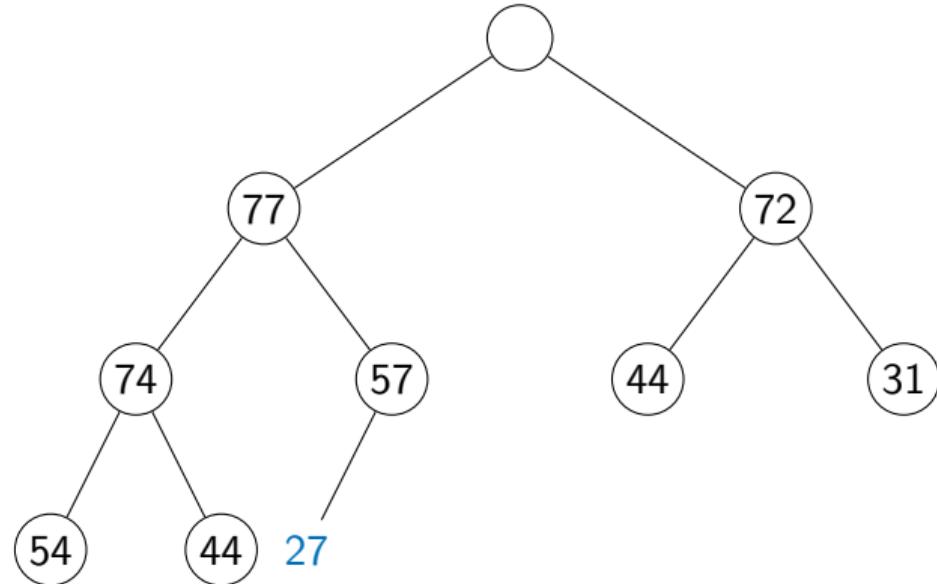
delete_max()

- Maximum value is always at the root



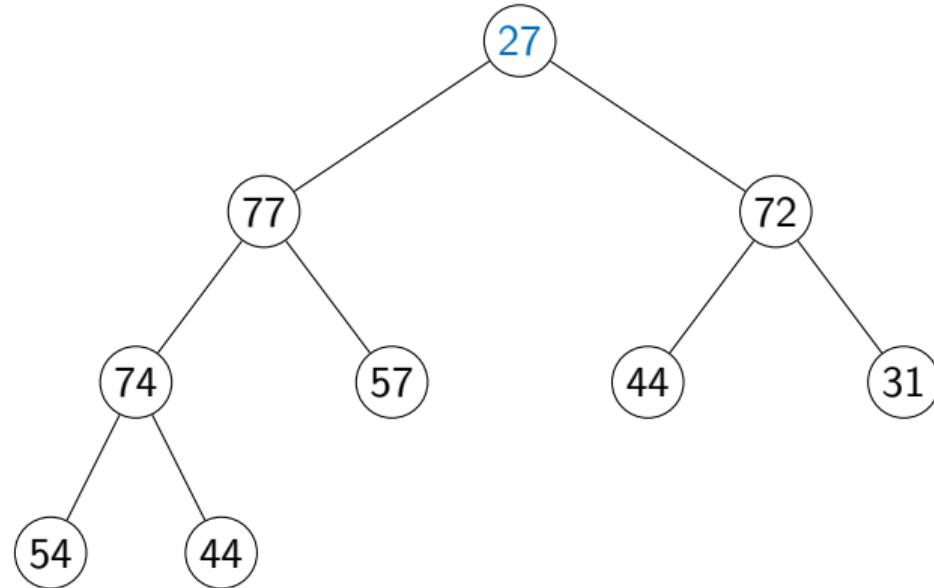
delete_max()

- Maximum value is always at the root
- After we delete one value, tree shrinks
 - Node to delete is rightmost at lowest level



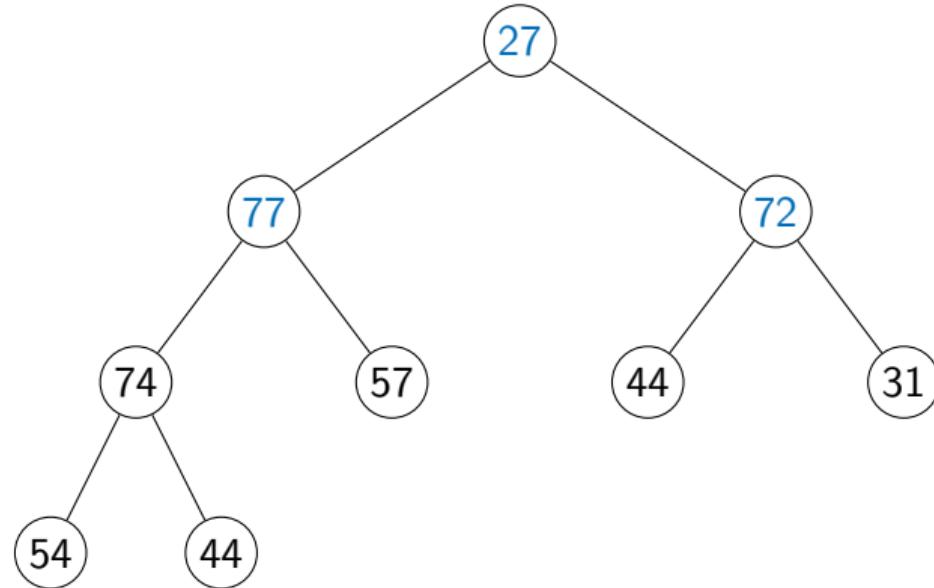
delete_max()

- Maximum value is always at the root
- After we delete one value, tree shrinks
 - Node to delete is rightmost at lowest level
- Move “homeless” value to the root



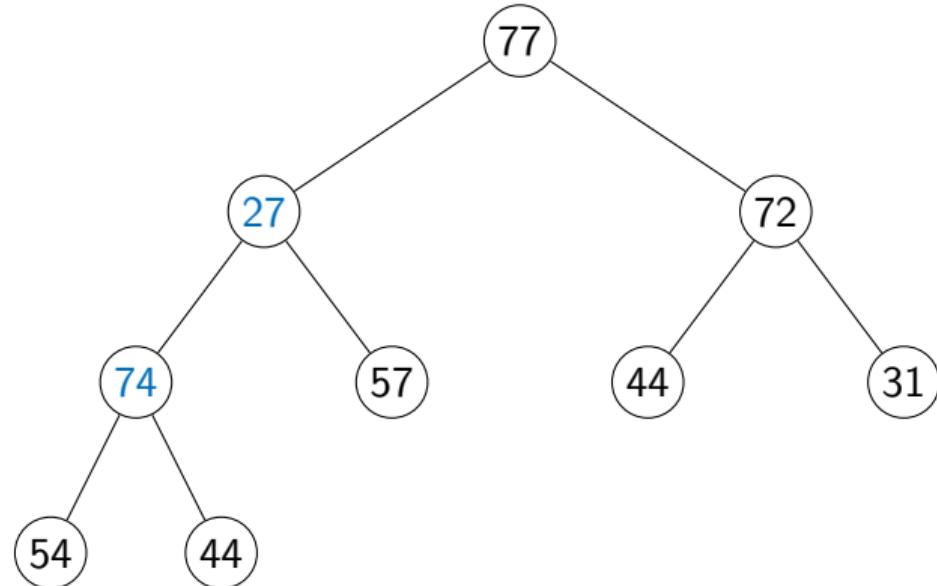
delete_max()

- Maximum value is always at the root
- After we delete one value, tree shrinks
 - Node to delete is rightmost at lowest level
- Move “homeless” value to the root
- Restore the heap property downwards



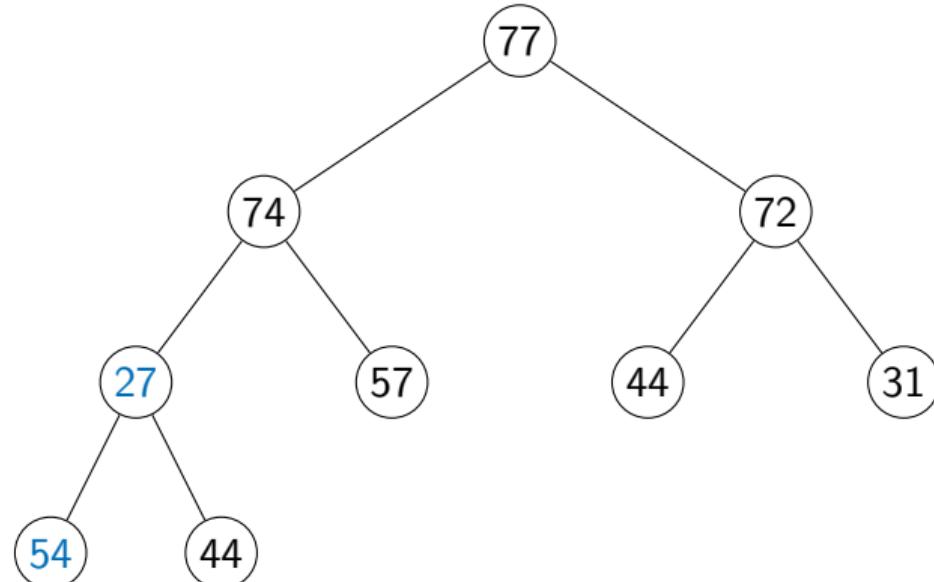
delete_max()

- Maximum value is always at the root
- After we delete one value, tree shrinks
 - Node to delete is rightmost at lowest level
- Move “homeless” value to the root
- Restore the heap property downwards
- Only need to follow a single path down
 - Again $O(\log N)$



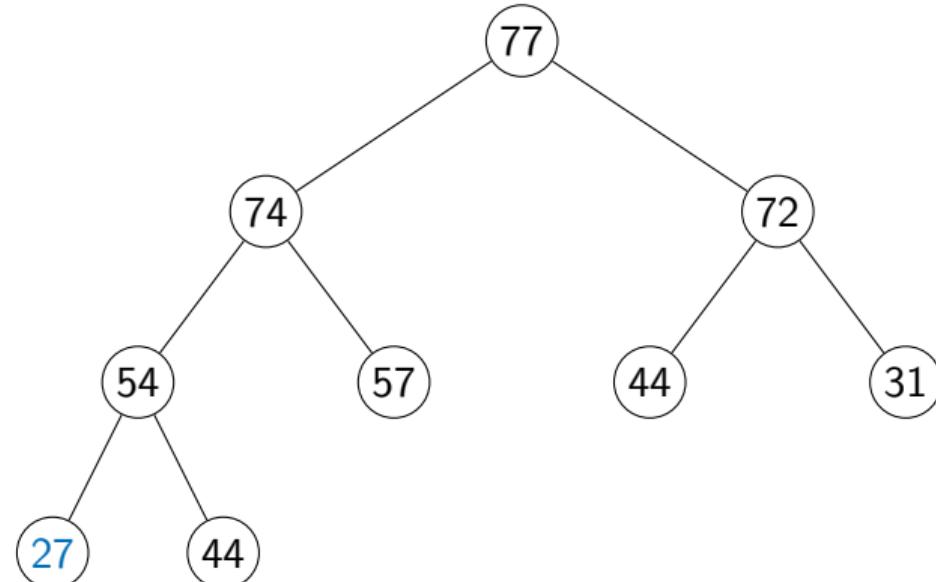
delete_max()

- Maximum value is always at the root
- After we delete one value, tree shrinks
 - Node to delete is rightmost at lowest level
- Move “homeless” value to the root
- Restore the heap property downwards
- Only need to follow a single path down
 - Again $O(\log N)$



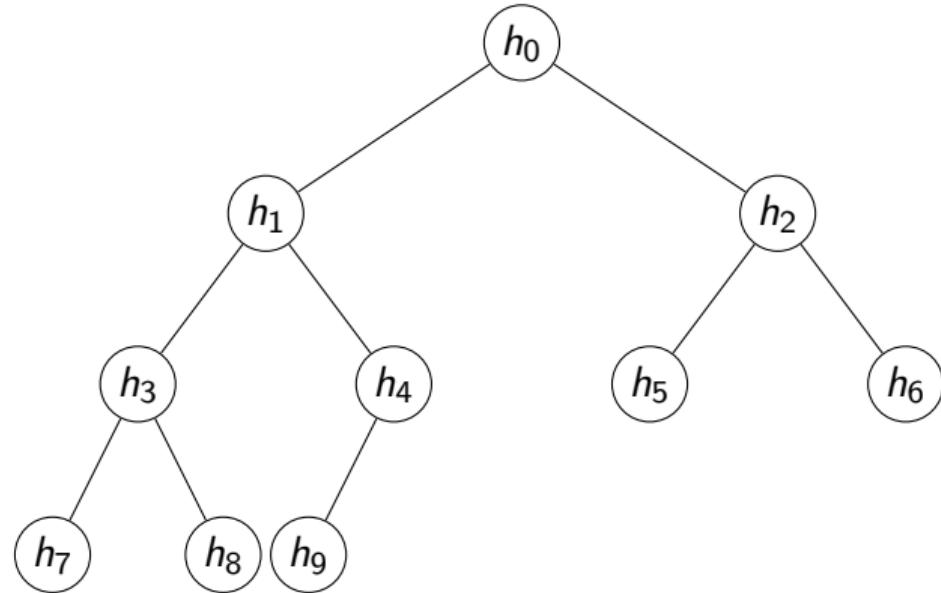
delete_max()

- Maximum value is always at the root
- After we delete one value, tree shrinks
 - Node to delete is rightmost at lowest level
- Move “homeless” value to the root
- Restore the heap property downwards
- Only need to follow a single path down
 - Again $O(\log N)$



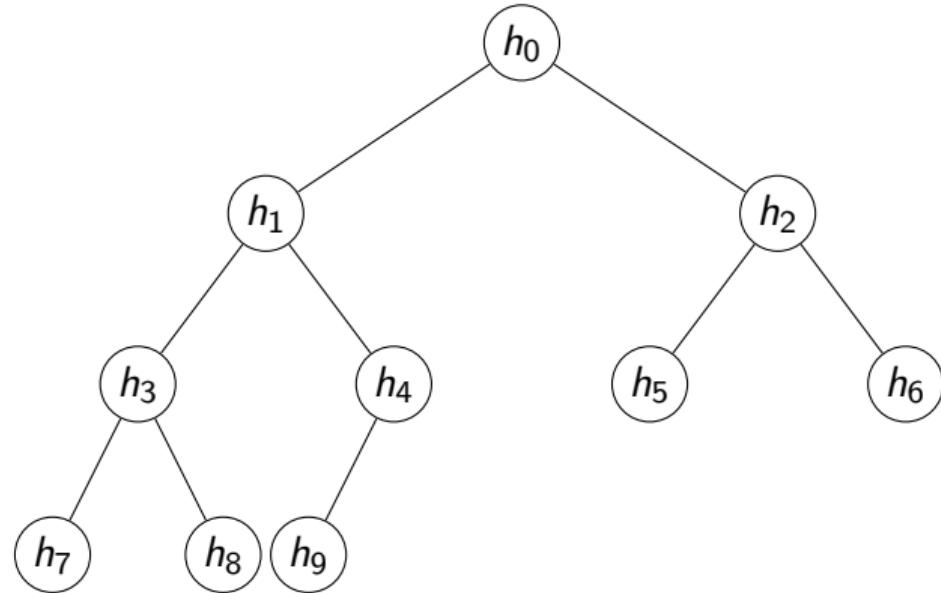
Implementation

- Number the nodes top to bottom left right
- Store as a list
 $H = [h_0, h_1, h_2, \dots, h_9]$
- Children of $H[i]$ are at $H[2*i+1], H[2*i+2]$
- Parent of $H[i]$ is at $H[(i-1)//2]$,
for $i > 0$



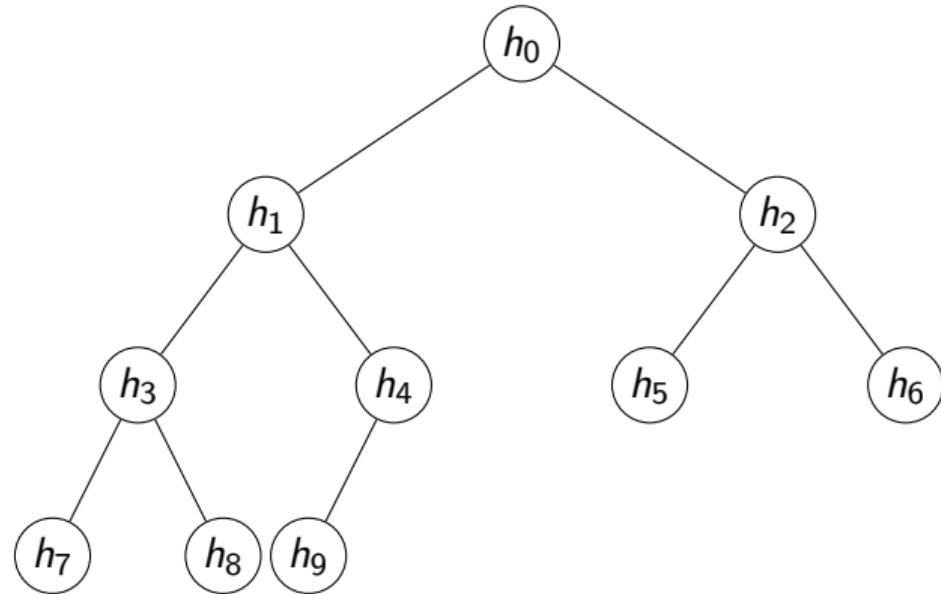
Building a heap — heapify()

- Convert a list `[v0, v1, ..., vN]` into a heap



Building a heap — heapify()

- Convert a list `[v0, v1, ..., vN]` into a heap
- Simple strategy
 - Start with an empty heap
 - Repeatedly apply `insert(vj)`
 - Total time is $O(N \log N)$



Better heapify()

- List $L = [v_0, v_1, \dots, v_N]$

Better heapify()

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
Slice `L[mid:]` has only leaf nodes
 - Already satisfy heap condition

Better heapify()

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
Slice `L[mid:]` has only leaf nodes
 - Already satisfy heap condition
- Fix heap property downwards for second last level

Better heapify()

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
Slice `L[mid:]` has only leaf nodes
 - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level

Better heapify()

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
Slice `L[mid:]` has only leaf nodes
 - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level
- ...
- Fix heap property at level 1
- Fix heap property at the root

Better heapify()

- List `L = [v0,v1,...,vN]`
 - `mid = len(L)//2`,
Slice `L[mid:]` has only leaf nodes
 - Already satisfy heap condition
 - Fix heap property downwards for second last level
 - Fix heap property downwards for third last level
 - ...
 - Fix heap property at level 1
 - Fix heap property at the root
- Each time we go up one level, one extra step per node to fix heap property

Better heapify()

- List `L = [v0,v1,...,vN]`
 - `mid = len(L)//2`,
Slice `L[mid:]` has only leaf nodes
 - Already satisfy heap condition
 - Fix heap property downwards for second last level
 - Fix heap property downwards for third last level
 - ...
 - Fix heap property at level 1
 - Fix heap property at the root
- Each time we go up one level, one extra step per node to fix heap property
 - However, number of nodes to fix halves

Better heapify()

- List `L = [v0,v1,...,vN]`
 - `mid = len(L)//2`,
Slice `L[mid:]` has only leaf nodes
 - Already satisfy heap condition
 - Fix heap property downwards for second last level
 - Fix heap property downwards for third last level
 - ...
 - Fix heap property at level 1
 - Fix heap property at the root
- Each time we go up one level, one extra step per node to fix heap property
 - However, number of nodes to fix halves
 - Second last level, $n/4 \times 1$ steps

Better heapify()

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
Slice `L[mid:]` has only leaf nodes
 - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level
- ...
- Fix heap property at level 1
- Fix heap property at the root
- Each time we go up one level, one extra step per node to fix heap property
- However, number of nodes to fix halves
- Second last level, $n/4 \times 1$ steps
- Third last level, $n/8 \times 2$ steps

Better heapify()

- List $L = [v_0, v_1, \dots, v_N]$
- $mid = \text{len}(L) // 2$,
Slice $L[mid:]$ has only leaf nodes
 - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level
- ...
- Fix heap property at level 1
- Fix heap property at the root
- Each time we go up one level, one extra step per node to fix heap property
- However, number of nodes to fix halves
 - Second last level, $n/4 \times 1$ steps
 - Third last level, $n/8 \times 2$ steps
 - Fourth last level, $n/16 \times 3$ steps

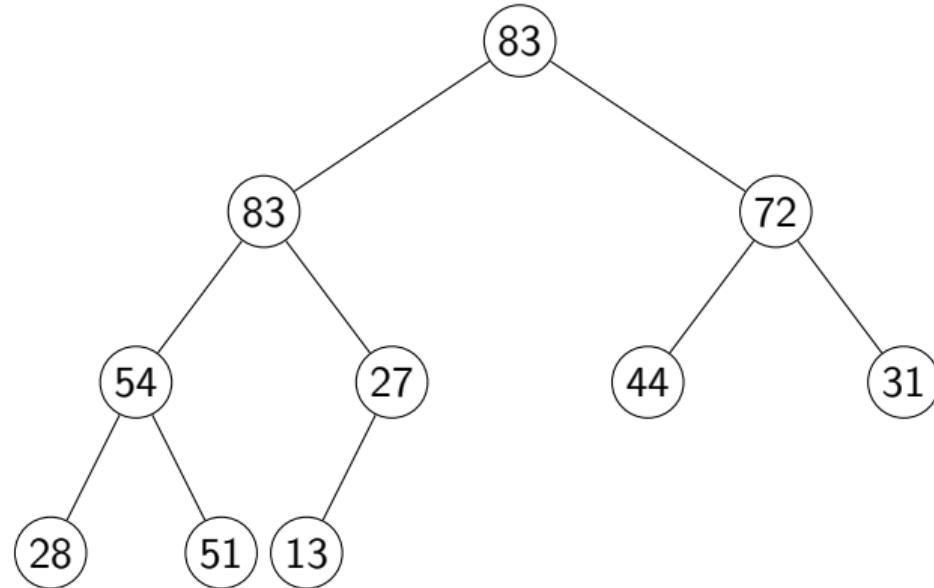
Better heapify()

- List $L = [v_0, v_1, \dots, v_N]$
- $mid = \text{len}(L) // 2$,
Slice $L[mid:]$ has only leaf nodes
 - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level
- ...
- Fix heap property at level 1
- Fix heap property at the root
- Each time we go up one level, one extra step per node to fix heap property
- However, number of nodes to fix halves
- Second last level, $n/4 \times 1$ steps
- Third last level, $n/8 \times 2$ steps
- Fourth last level, $n/16 \times 3$ steps
- ...
- Cost turns out to be $O(n)$

Summary

- Heaps are a tree implementation of priority queues

- `insert()` is $O(\log N)$
- `delete_max()` is $O(\log N)$
- `heapify()` builds a heap in $O(N)$

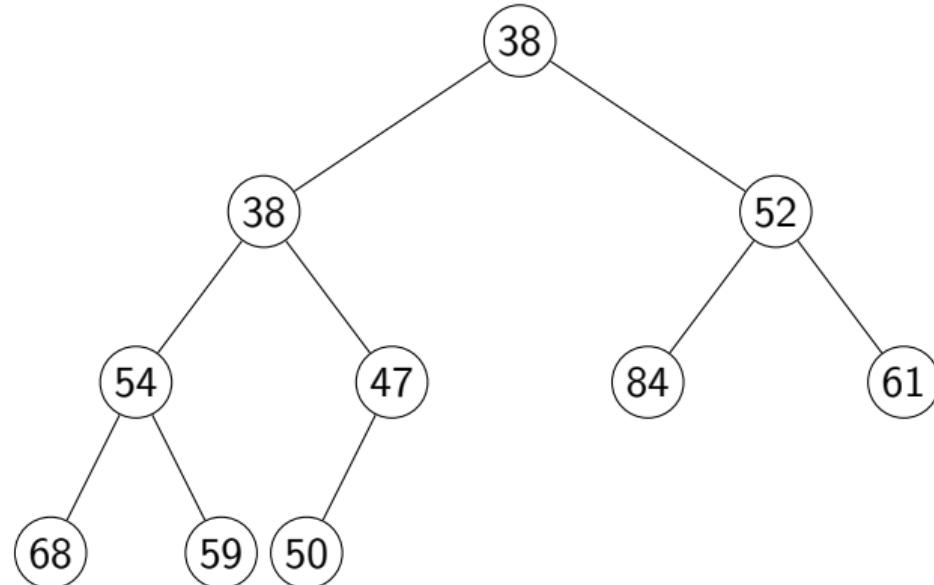


Summary

- Heaps are a tree implementation of priority queues

- `insert()` is $O(\log N)$
 - `delete_max()` is $O(\log N)$
 - `heapify()` builds a heap in $O(N)$

- Can invert the heap condition
 - Each node is smaller than its children
 - **min-heap**
 - `delete_min()` rather than `delete_max()`



Using Heaps in Algorithms

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 6

Priority queues and heaps

- Priority queues support the following operations
 - `insert()`
 - `delete_max()` or `delete_min()`
- Heaps are a tree based implementation of priority queues
 - `insert()`, `delete_max()` / `delete_min()` are both $O(\log n)$
 - `heapify()` builds a heap from a list/array in time $O(n)$
- Heap can be represented as a list/array
 - Simple index arithmetic to find parent and children of a node
- What more do we need to use a heap in an algorithm?

Dijkstra's algorithm

- Maintain two dictionaries with vertices as keys
 - `visited`, initially `False` for all `v`
 - `distance`, initially `infinity` for all `v`
- Set `distance[s]` to 0
- Repeat, until all reachable vertices are visited
 - Find unvisited vertex `nextv` with minimum distance
 - Set `visited[nextv]` to True
 - Recompute `distance[v]` for every neighbour `v` of `nextv`

```
def dijkstra(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    (visited,distance) = ({},{} )  
    for v in range(rows):  
        (visited[v],distance[v]) = (False,infinity)  
    distance[s] = 0  
    for u in range(rows):  
        nextd = min([distance[v] for v in range(rows)  
                    if not visited[v]])  
        nextvlist = [v for v in range(rows)  
                    if (not visited[v]) and  
                        distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for v in range(cols):  
            if WMat[nextv,v,0] == 1 and (not visited[v]):  
                distance[v] = min(distance[v],distance[nextv]  
                                +WMat[nextv,v,1])  
  
    return(distance)
```

Dijkstra's algorithm

Bottleneck

- Find unvisited vertex j with minimum distance
 - Naive implementation requires an $O(n)$ scan

```
def dijkstra(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    (visited,distance) = ({},{} )  
    for v in range(rows):  
        (visited[v],distance[v]) = (False,infinity)  
    distance[s] = 0  
    for u in range(rows):  
        nextd = min([distance[v] for v in range(rows)  
                    if not visited[v]])  
        nextvlist = [v for v in range(rows)  
                    if (not visited[v]) and  
                        distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for v in range(cols):  
            if WMat[nextv,v,0] == 1 and (not visited[v]):  
                distance[v] = min(distance[v],distance[nextv]  
                                +WMat[nextv,v,1])  
    return(distance)
```

Dijkstra's algorithm

Bottleneck

- Find unvisited vertex j with minimum distance
 - Naive implementation requires an $O(n)$ scan
- Maintain unvisited vertices as a min-heap
 - `delete_min()` in $O(\log n)$ time

```
def dijkstra(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    (visited,distance) = ({},{} )  
    for v in range(rows):  
        (visited[v],distance[v]) = (False,infinity)  
    distance[s] = 0  
    for u in range(rows):  
        nextd = min([distance[v] for v in range(rows)  
                    if not visited[v]])  
        nextvlist = [v for v in range(rows)  
                    if (not visited[v]) and  
                        distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for v in range(cols):  
            if WMat[nextv,v,0] == 1 and (not visited[v]):  
                distance[v] = min(distance[v],distance[nextv]  
                                +WMat[nextv,v,1])  
    return(distance)
```

Dijkstra's algorithm

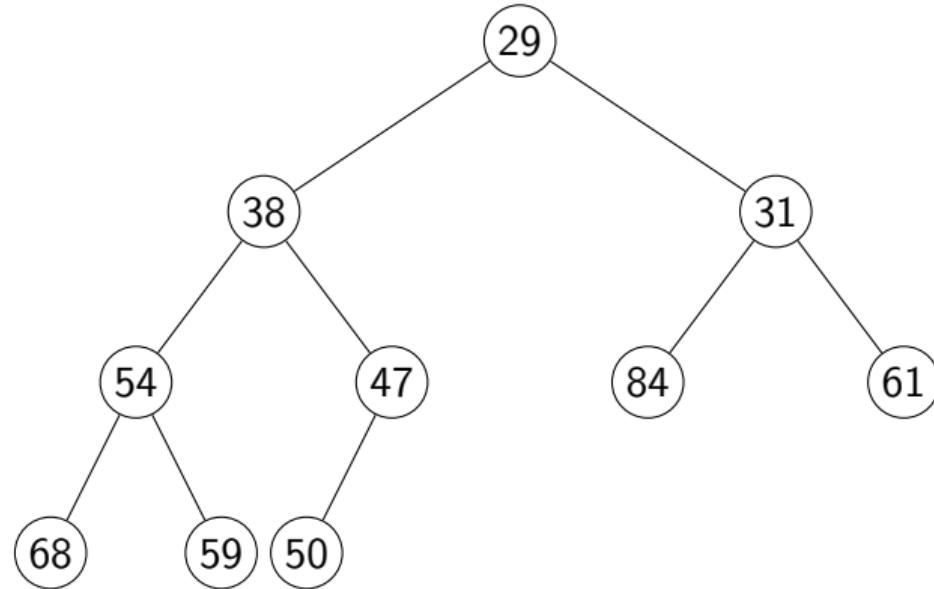
Bottleneck

- Find unvisited vertex j with minimum distance
 - Naive implementation requires an $O(n)$ scan
- Maintain unvisited vertices as a min-heap
 - `delete_min()` in $O(\log n)$ time
- But, also need to update distances of neighbours
 - Unvisited neighbours' distances are inside the min-heap
 - Updating a value is not a basic heap operation

```
def dijkstra(WMat,s):  
    (rows,cols,x) = WMat.shape  
    infinity = np.max(WMat)*rows+1  
    (visited,distance) = ({},{} )  
    for v in range(rows):  
        (visited[v],distance[v]) = (False,infinity)  
    distance[s] = 0  
    for u in range(rows):  
        nextd = min([distance[v] for v in range(rows)  
                    if not visited[v]])  
        nextvlist = [v for v in range(rows)  
                    if (not visited[v]) and  
                        distance[v] == nextd]  
        if nextvlist == []:  
            break  
        nextv = min(nextvlist)  
        visited[nextv] = True  
        for v in range(cols):  
            if WMat[nextv,v,0] == 1 and (not visited[v]):  
                distance[v] = min(distance[v],distance[nextv]  
                                +WMat[nextv,v,1])  
    return(distance)
```

Updating values in a min-heap

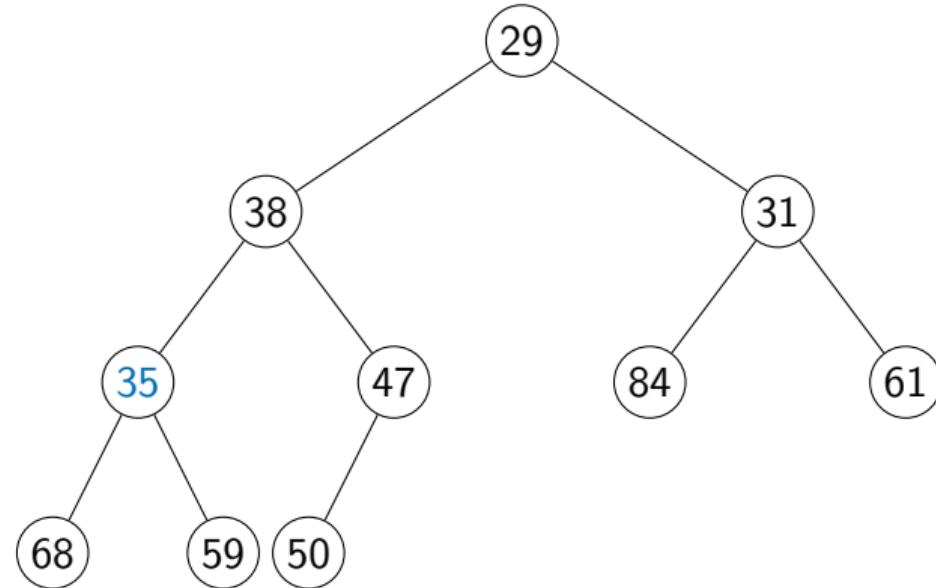
- Change 54 to 35



Updating values in a min-heap

- Change 54 to 35

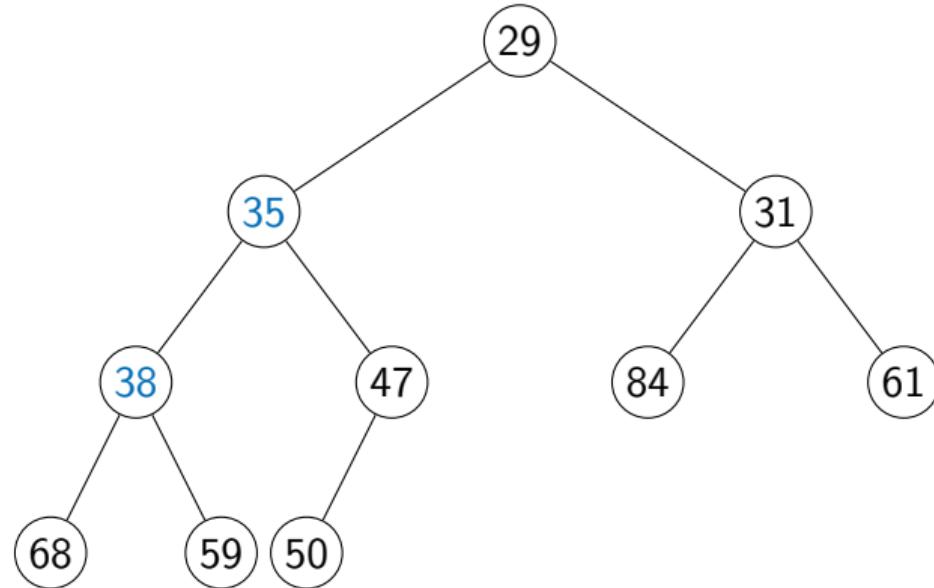
- Reducing a value can create a violation with parent



Updating values in a min-heap

- Change 54 to 35

- Reducing a value can create a violation with parent
- Swap upwards to restore heap, similar to `insert()`

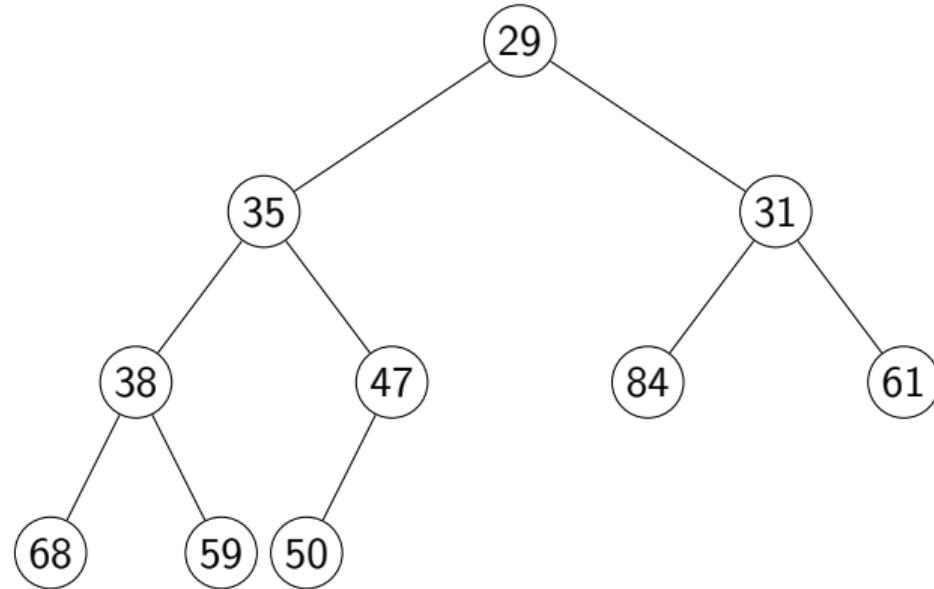


Updating values in a min-heap

- Change 54 to 35

- Reducing a value can create a violation with parent
- Swap upwards to restore heap, similar to `insert()`

- Change 29 to 71



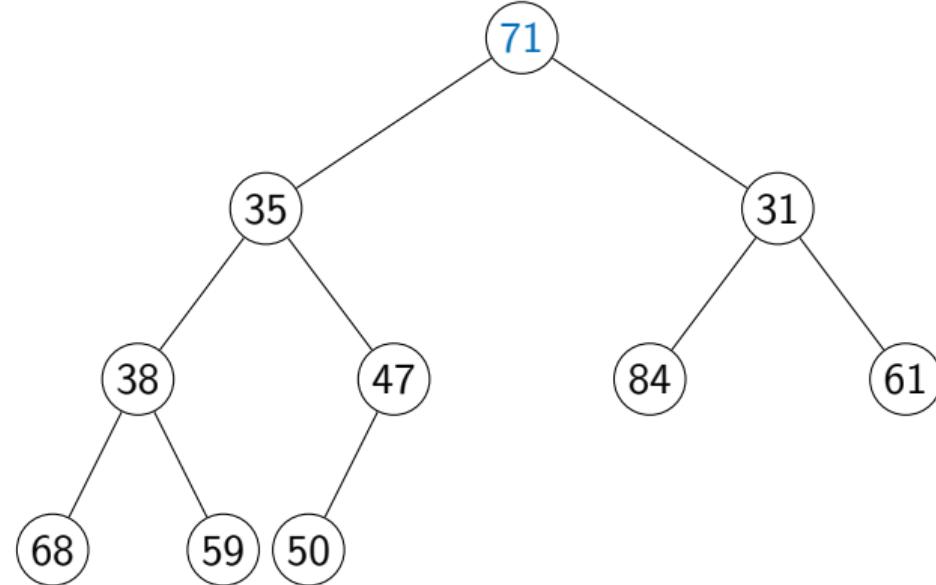
Updating values in a min-heap

- Change 54 to 35

- Reducing a value can create a violation with parent
 - Swap upwards to restore heap, similar to `insert()`

- Change 29 to 71

- Increasing a value can create a violation with child



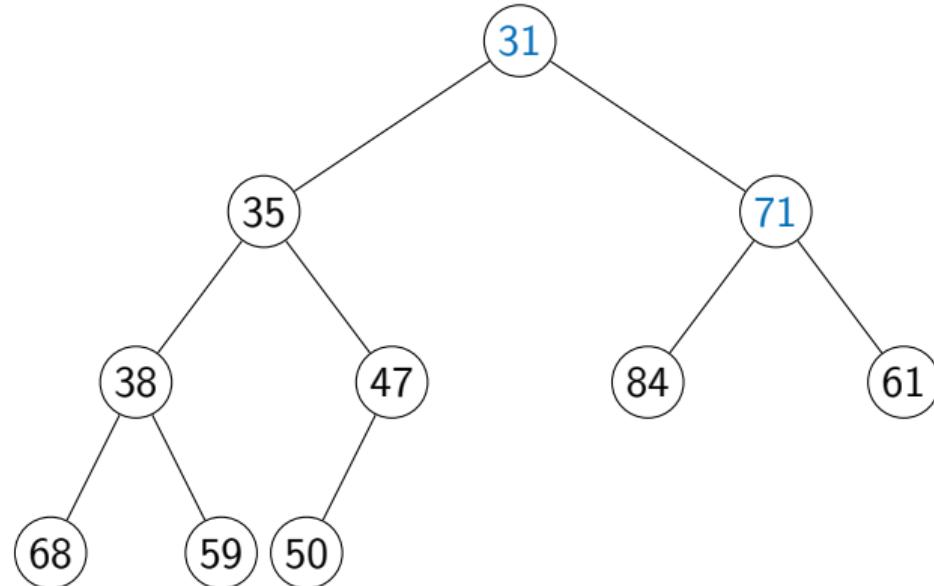
Updating values in a min-heap

- Change 54 to 35

- Reducing a value can create a violation with parent
- Swap upwards to restore heap, similar to `insert()`

- Change 29 to 71

- Increasing a value can create a violation with child
- Swap downwards to restore heap, similar to `delete_min()`



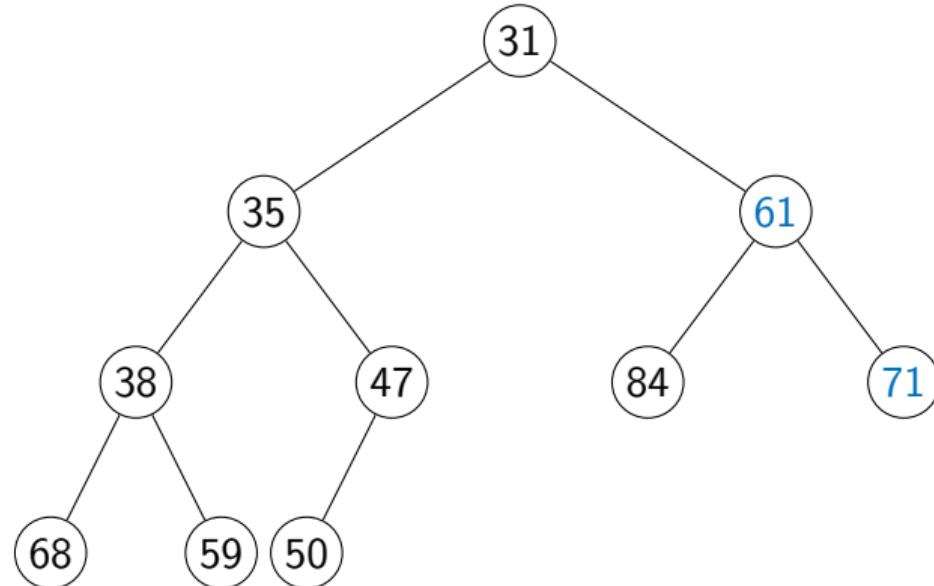
Updating values in a min-heap

- Change 54 to 35

- Reducing a value can create a violation with parent
 - Swap upwards to restore heap, similar to `insert()`

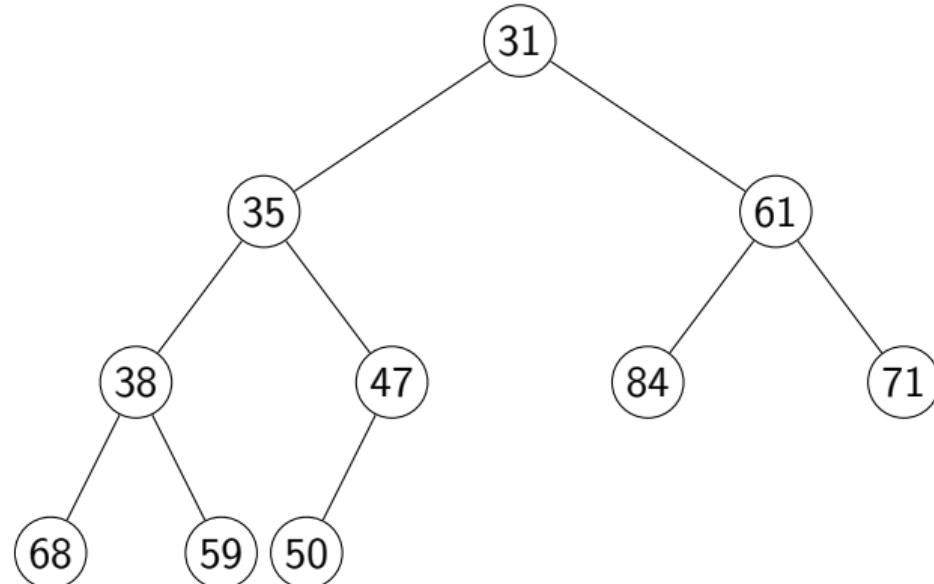
- Change 29 to 71

- Increasing a value can create a violation with child
 - Swap downwards to restore heap, similar to `delete_min()`



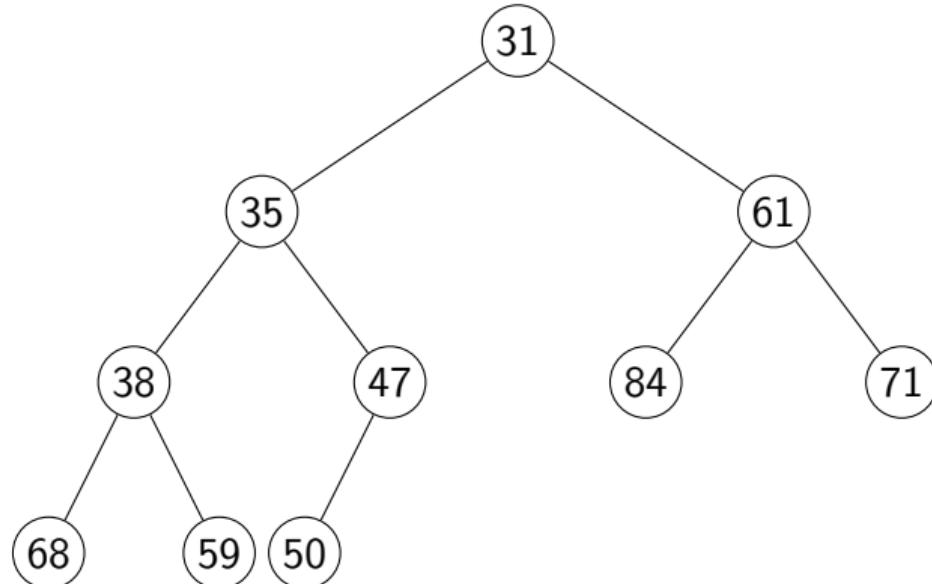
Updating values in a min-heap

- Change 54 to 35
 - Reducing a value can create a violation with parent
 - Swap upwards to restore heap, similar to `insert()`
- Change 29 to 71
 - Increasing a value can create a violation with child
 - Swap downwards to restore heap, similar to `delete_min()`
- Both updates are $O(\log n)$
 - Are we done?



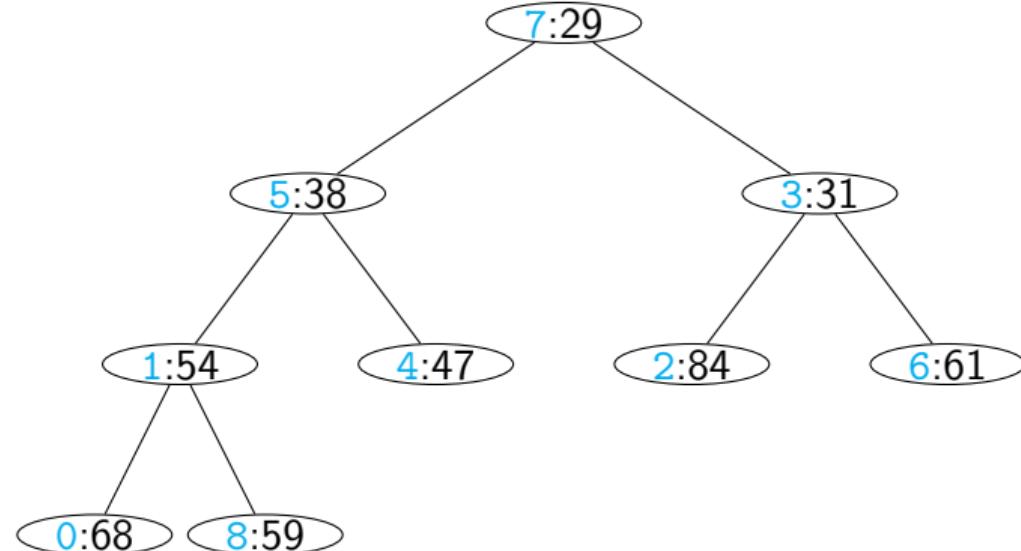
Updating values in a min-heap

- Change 54 to 35
 - Reducing a value can create a violation with parent
 - Swap upwards to restore heap, similar to `insert()`
- Change 29 to 71
 - Increasing a value can create a violation with child
 - Swap downwards to restore heap, similar to `delete_min()`
- Both updates are $O(\log n)$
 - Are we done?
- Locate the node to update?



Updating values in a min-heap

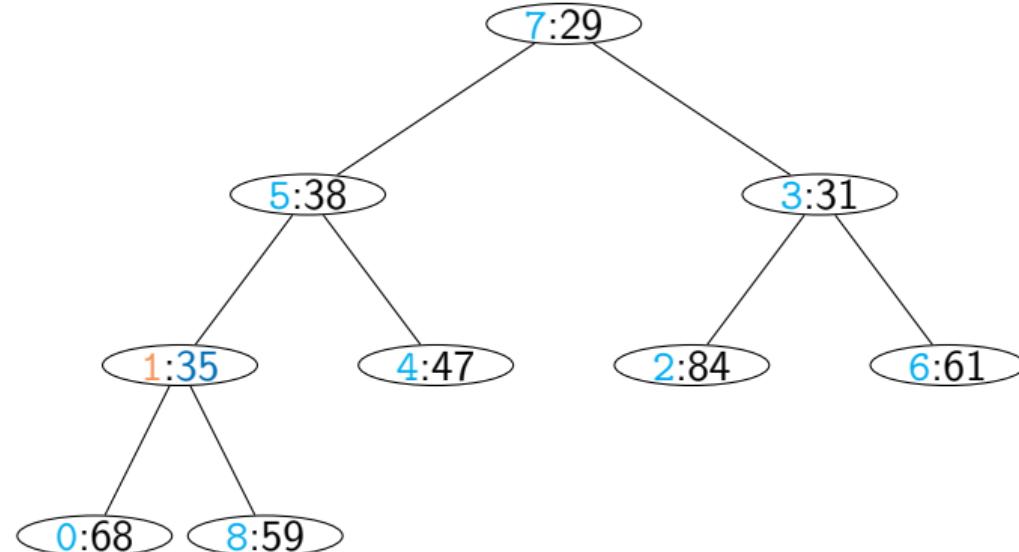
- Maintain two additional dictionaries
 - Vertices are $\{0, 1, \dots, n-1\}$
 - Heap positions are $\{0, 1, \dots, n-1\}$
 - VtoH** maps vertices to heap positions
 - HtoV** maps heap positions to vertices



VtoH	0	1	2	3	4	5	6	7	8
HtoV	0	1	2	3	4	5	6	7	8
0:68	7	3	5	2	4	1	6	0	8
1:54	7	5	3	1	4	2	6	0	8

Updating values in a min-heap

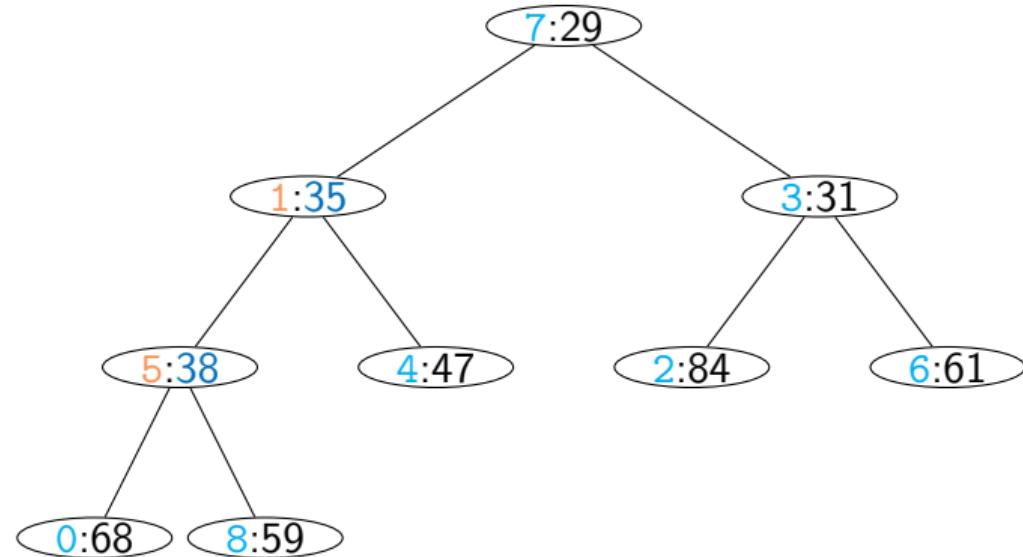
- Maintain two additional dictionaries
 - Vertices are $\{0, 1, \dots, n-1\}$
 - Heap positions are $\{0, 1, \dots, n-1\}$
 - VtoH** maps vertices to heap positions
 - HtoV** maps heap positions to vertices



VtoH	0	1	2	3	4	5	6	7	8
HtoV	0	1	2	3	4	5	6	7	8
0	7	3	5	2	4	1	6	0	8
7	5	3	1	4	2	6	0	8	

Updating values in a min-heap

- Maintain two additional dictionaries
 - Vertices are $\{0, 1, \dots, n-1\}$
 - Heap positions are $\{0, 1, \dots, n-1\}$
 - $V_{to}H$ maps vertices to heap positions
 - $H_{to}V$ maps heap positions to vertices
- Update node 1 to 35
- Update $V_{to}H$ and $H_{to}V$ each time we swap values in the heap

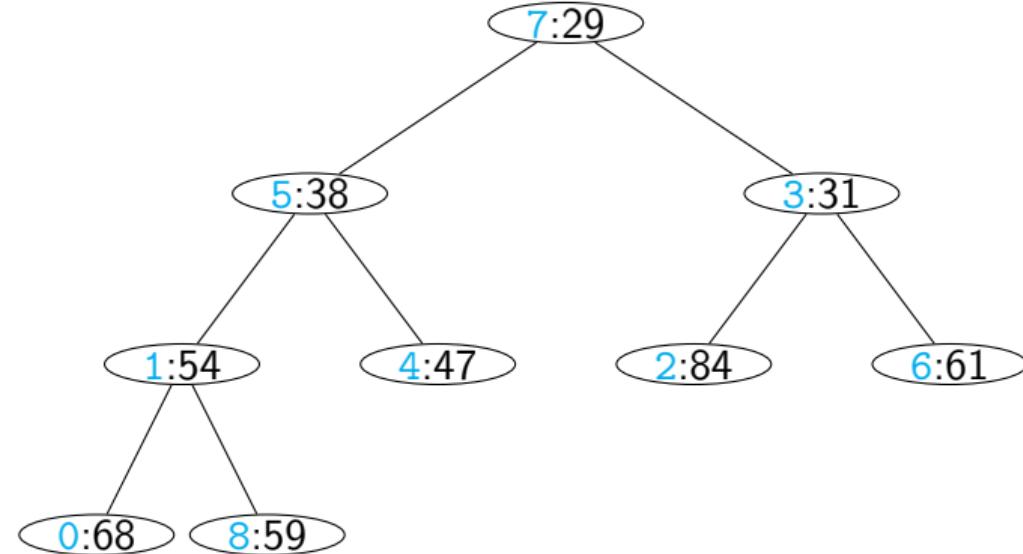


$V_{to}H$	0	1	2	3	4	5	6	7	8
$H_{to}V$	0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8	0
7	1	3	5	4	2	6	0	8	7

Dijkstra's algorithm

- Using min-heaps

- Identifying next vertex to visit is $O(\log n)$
- Updating distance takes $O(\log n)$ per neighbour
- Adjacency list — proportionally to degree



VtoH	0	1	2	3	4	5	6	7	8
HtoV	7	3	5	2	4	1	6	0	8
	0	1	2	3	4	5	6	7	8
	7	5	3	1	4	2	6	0	8

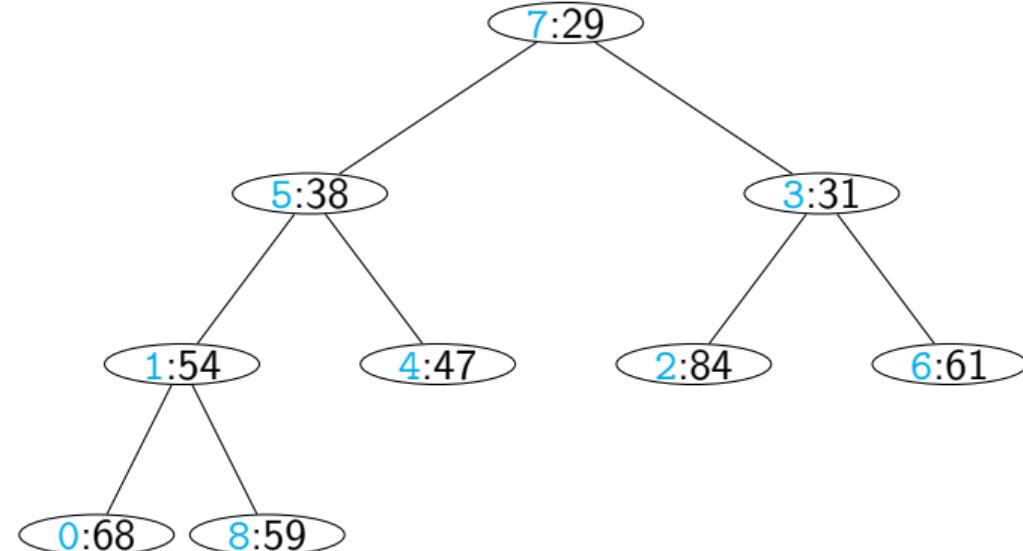
Dijkstra's algorithm

- Using min-heaps

- Identifying next vertex to visit is $O(\log n)$
- Updating distance takes $O(\log n)$ per neighbour
- Adjacency list — proportionally to degree

- Cumulatively

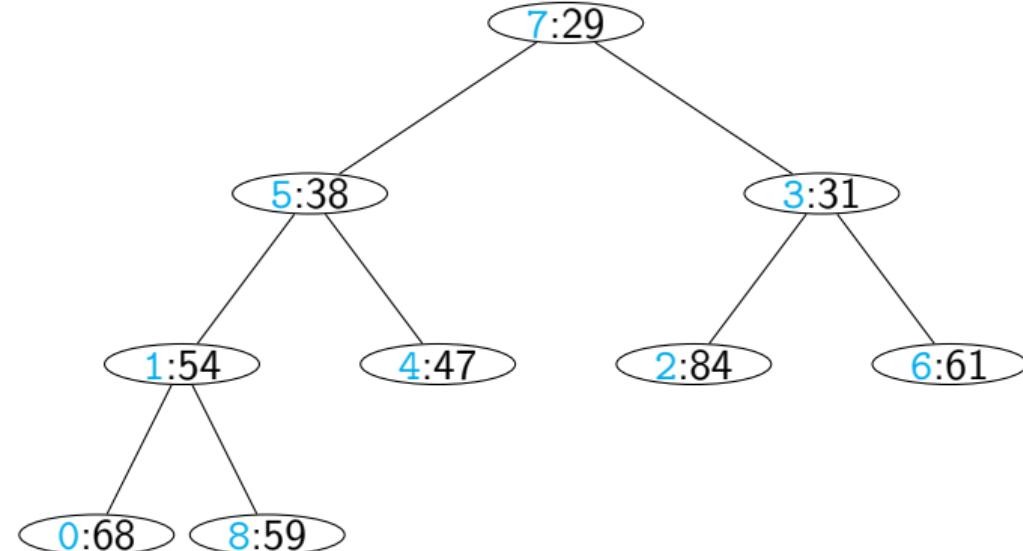
- $O(n \log n)$ to identify vertices to visit across n iterations
- $O(m \log n)$ distance updates overall



VtoH	0	1	2	3	4	5	6	7	8
HtoV	7	3	5	2	4	1	6	0	8
VtoH	0	1	2	3	4	5	6	7	8
HtoV	7	5	3	1	4	2	6	0	8

Dijkstra's algorithm

- Using min-heaps
 - Identifying next vertex to visit is $O(\log n)$
 - Updating distance takes $O(\log n)$ per neighbour
 - Adjacency list — proportionally to degree
- Cumulatively
 - $O(n \log n)$ to identify vertices to visit across n iterations
 - $O(m \log n)$ distance updates overall
- Overall $O((m + n) \log n)$



VtoH	0	1	2	3	4	5	6	7	8
HtoV	7	3	5	2	4	1	6	0	8
VtoH	0	1	2	3	4	5	6	7	8
HtoV	7	5	3	1	4	2	6	0	8

Heap sort

- Start with an unordered list

Heap sort

- Start with an unordered list
- Build a heap — $O(n)$

Heap sort

- Start with an unordered list
- Build a heap — $O(n)$
- Call `delete_max()` n times to extract elements in descending order — $O(n \log n)$

Heap sort

- Start with an unordered list
- Build a heap — $O(n)$
- Call `delete_max()` n times to extract elements in descending order — $O(n \log n)$
- After each `delete_max()`, heap shrinks by 1

Heap sort

- Start with an unordered list
- Build a heap — $O(n)$
- Call `delete_max()` n times to extract elements in descending order — $O(n \log n)$
- After each `delete_max()`, heap shrinks by 1
- Store maximum value at the end of current heap

Heap sort

- Start with an unordered list
- Build a heap — $O(n)$
- Call `delete_max()` n times to extract elements in descending order — $O(n \log n)$
- After each `delete_max()`, heap shrinks by 1
- Store maximum value at the end of current heap
- In place $O(n \log n)$ sort

Summary

- Updating a value in a heap takes $O(\log n)$

Summary

- Updating a value in a heap takes $O(\log n)$
- Need to maintain additional pointers to map values to heap positions and vice versa

Summary

- Updating a value in a heap takes $O(\log n)$
- Need to maintain additional pointers to map values to heap positions and vice versa
- With this extended notion of heap, Dijkstra's algorithm complexity improves from $O(n^2)$ to $O((m + n) \log n)$

Summary

- Updating a value in a heap takes $O(\log n)$
- Need to maintain additional pointers to map values to heap positions and vice versa
- With this extended notion of heap, Dijkstra's algorithm complexity improves from $O(n^2)$ to $O((m + n) \log n)$
- In a similar way, improve Prim's algorithm to $O((m + n) \log n)$

Summary

- Updating a value in a heap takes $O(\log n)$
- Need to maintain additional pointers to map values to heap positions and vice versa
- With this extended notion of heap, Dijkstra's algorithm complexity improves from $O(n^2)$ to $O((m + n) \log n)$
- In a similar way, improve Prim's algorithm to $O((m + n) \log n)$
- Heaps can also be used to sort a list in place in $O(n \log n)$

Search Trees

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 6

Dynamic sorted data

- Sorting is useful for efficient searching

Dynamic sorted data

- Sorting is useful for efficient searching
- What if the data is changing dynamically?
 - Items are periodically inserted and deleted

Dynamic sorted data

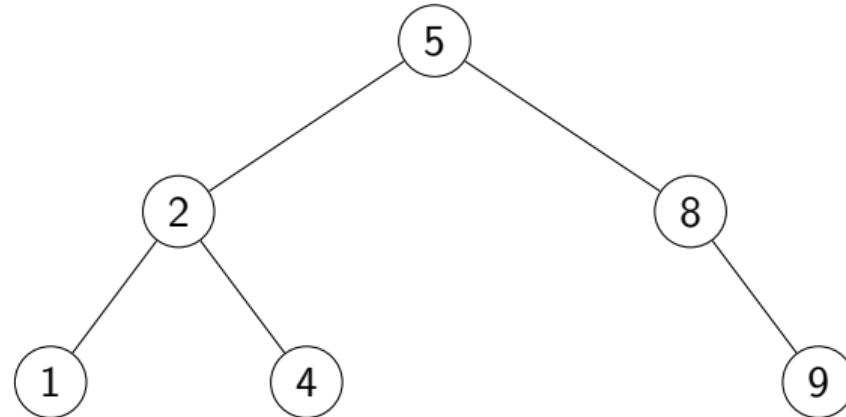
- Sorting is useful for efficient searching
- What if the data is changing dynamically?
 - Items are periodically inserted and deleted
- Insert/delete in a sorted list takes time $O(n)$

Dynamic sorted data

- Sorting is useful for efficient searching
- What if the data is changing dynamically?
 - Items are periodically inserted and deleted
- Insert/delete in a sorted list takes time $O(n)$
- Move to a tree structure, like heaps for priority queues

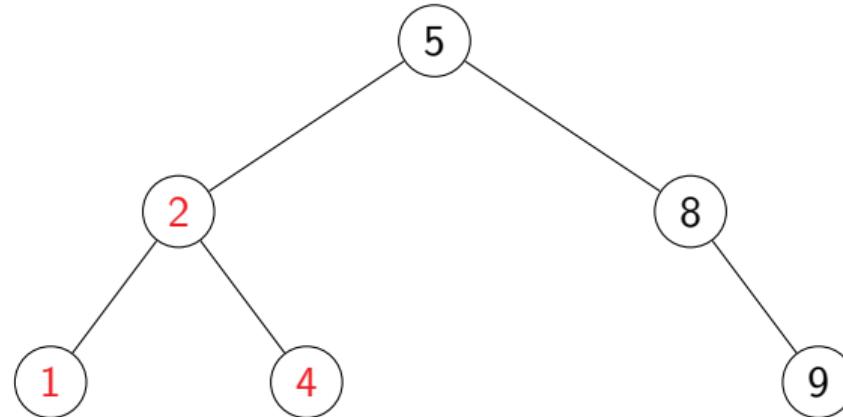
Binary search tree

- For each node with value v



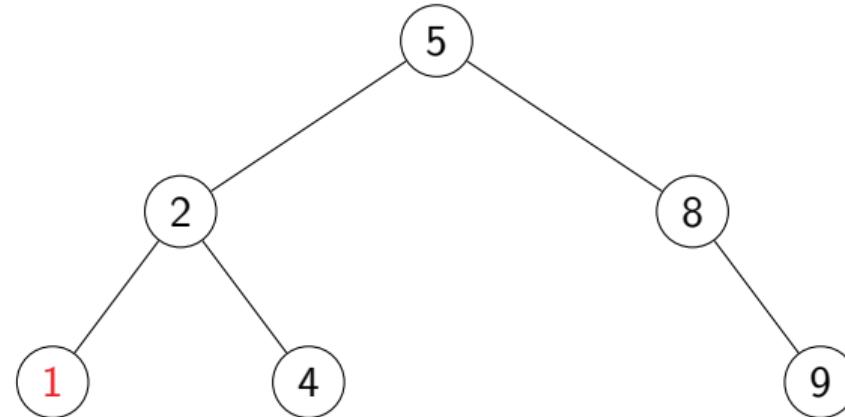
Binary search tree

- For each node with value v
 - All values in the left subtree
are $< v$



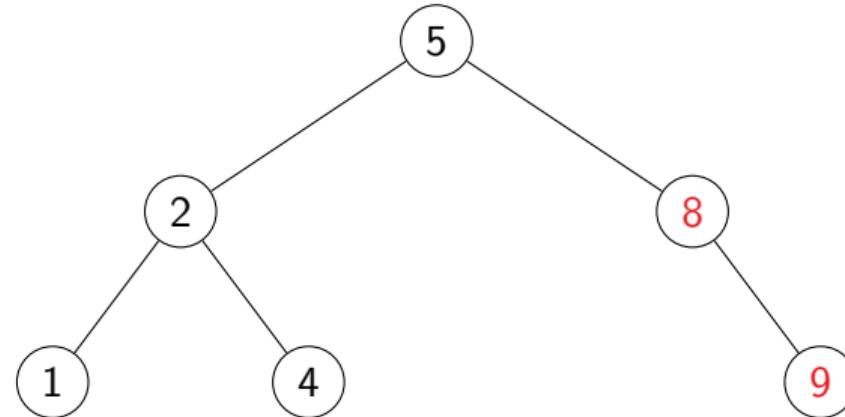
Binary search tree

- For each node with value v
 - All values in the left subtree
are $< v$



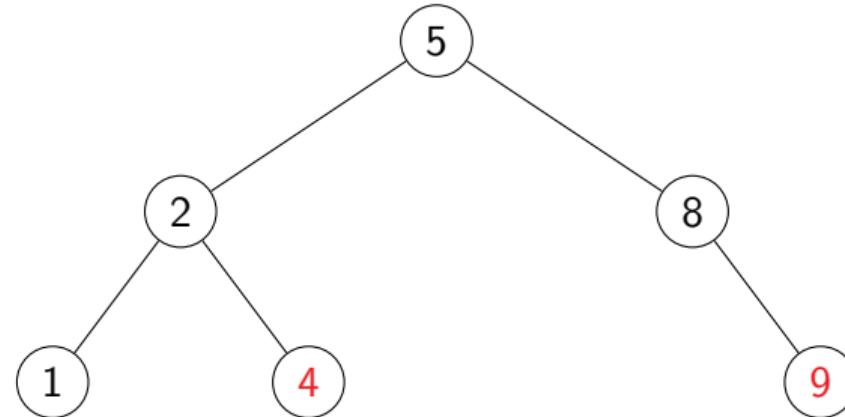
Binary search tree

- For each node with value v
 - All values in the left subtree are $< v$
 - All values in the right subtree are $> v$



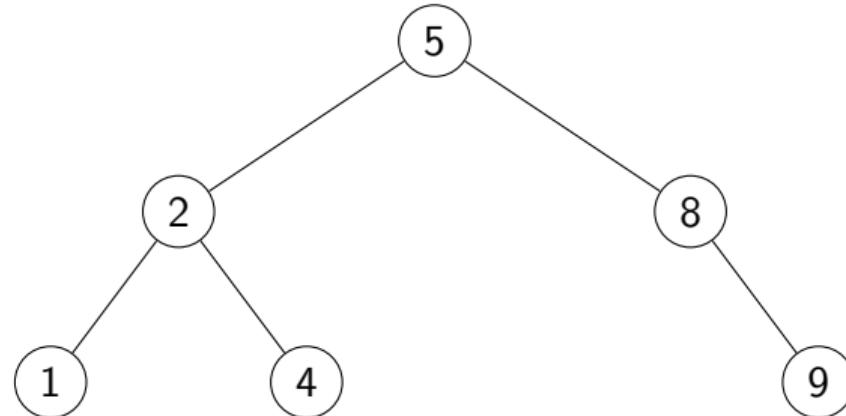
Binary search tree

- For each node with value v
 - All values in the left subtree are $< v$
 - All values in the right subtree are $> v$



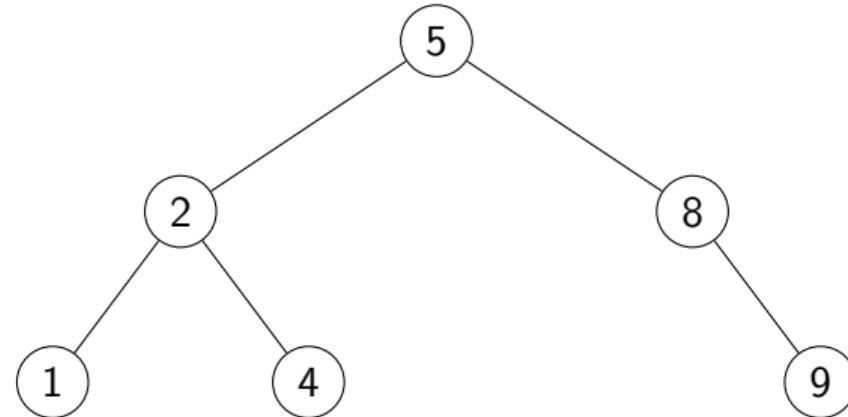
Binary search tree

- For each node with value v
 - All values in the left subtree are $< v$
 - All values in the right subtree are $> v$
- No duplicate values



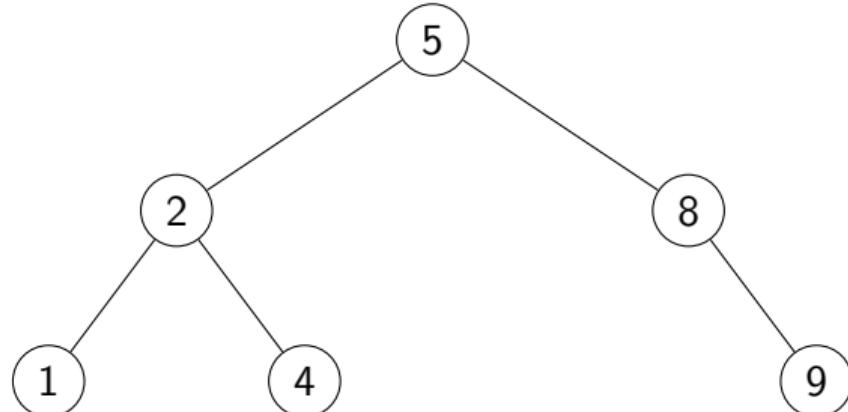
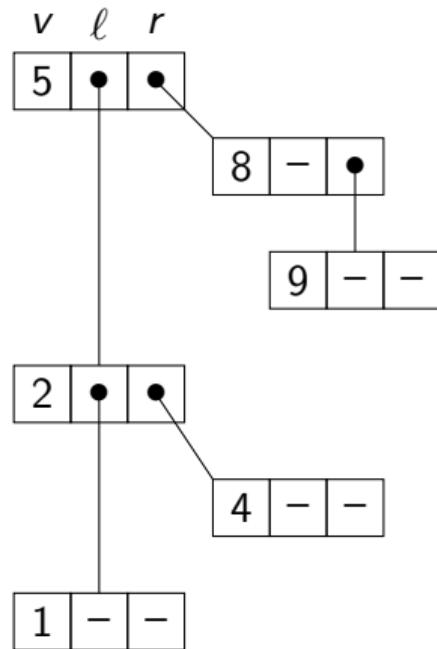
Implementing a binary search tree

- Each node has a value and pointers to its children



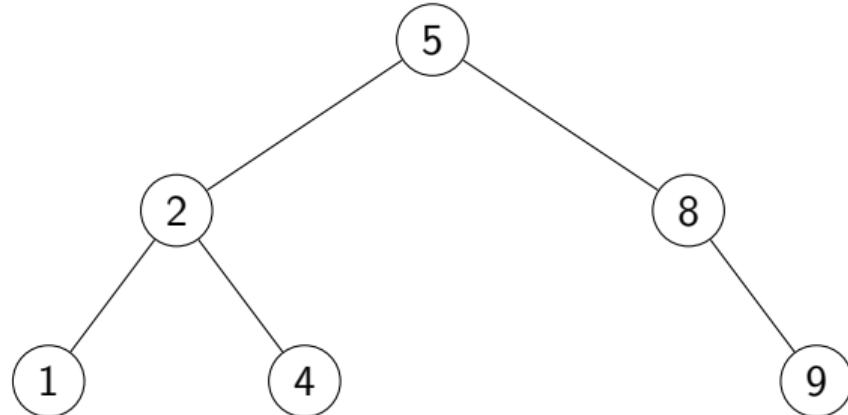
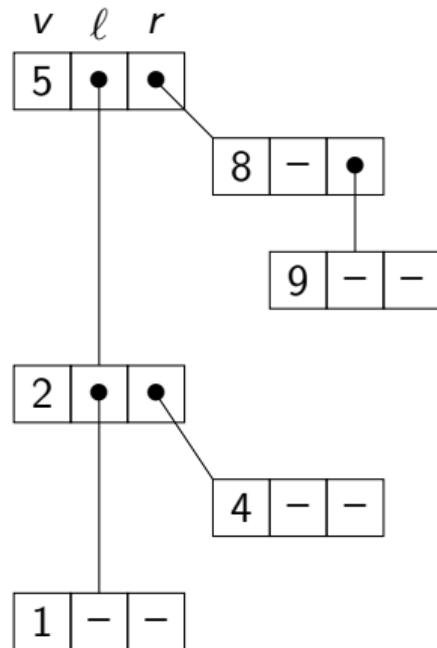
Implementing a binary search tree

- Each node has a value and pointers to its children



Implementing a binary search tree

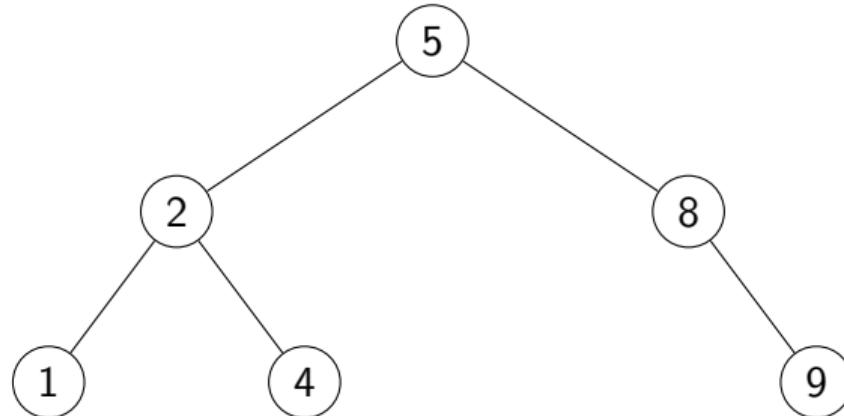
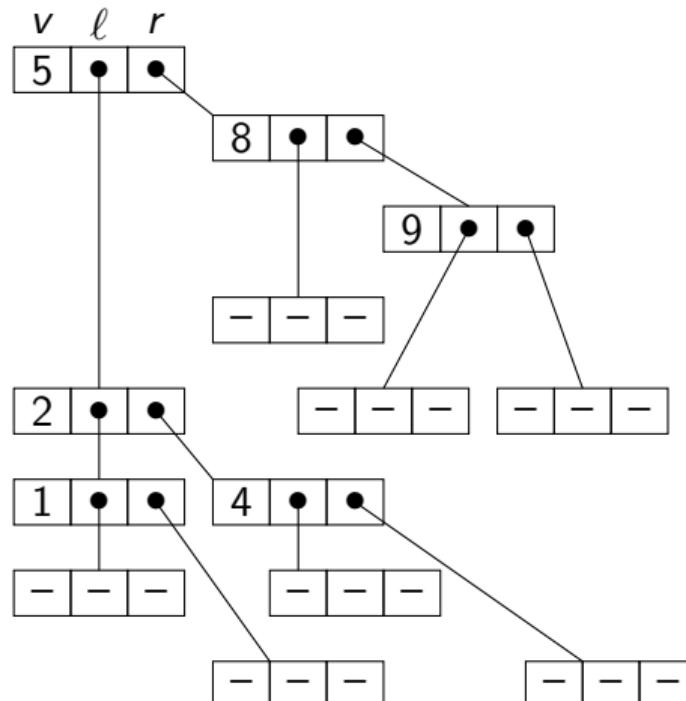
- Each node has a value and pointers to its children



- Add a frontier with empty nodes, all fields –
 - Empty tree is single empty node
 - Leaf node points to empty nodes

Implementing a binary search tree

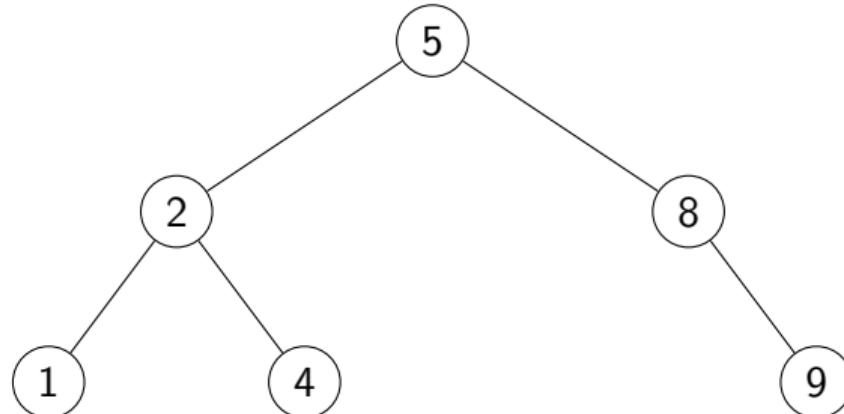
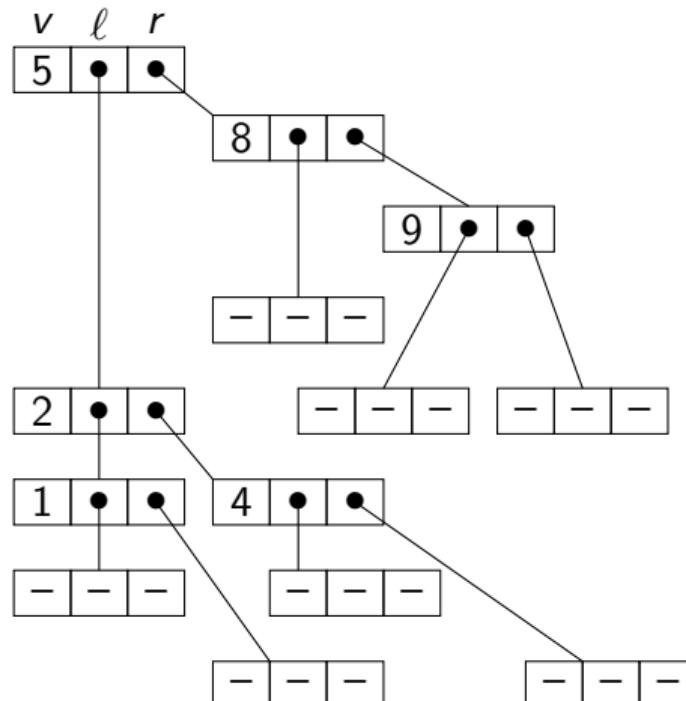
- Each node has a value and pointers to its children



- Add a frontier with empty nodes, all fields –
 - Empty tree is single empty node
 - Leaf node points to empty nodes

Implementing a binary search tree

- Each node has a value and pointers to its children



- Add a frontier with empty nodes, all fields –
 - Empty tree is single empty node
 - Leaf node points to empty nodes
- Easier to implement operations recursively

The class Tree

- Three local fields, `value`, `left`, `right`
- Value `None` for empty value –
- Empty tree has all fields `None`
- Leaf has a nonempty `value` and empty `left` and `right`

```
class Tree:  
  
    # Constructor:  
    def __init__(self, initval=None):  
        self.value = initval  
        if self.value:  
            self.left = Tree()  
            self.right = Tree()  
        else:  
            self.left = None  
            self.right = None  
        return  
  
    # Only empty node has value None  
    def isempty(self):  
        return (self.value == None)  
  
    # Leaf nodes have both children empty  
    def isleaf(self):  
        return (self.value != None and  
                self.left.isempty() and  
                self.right.isempty())
```

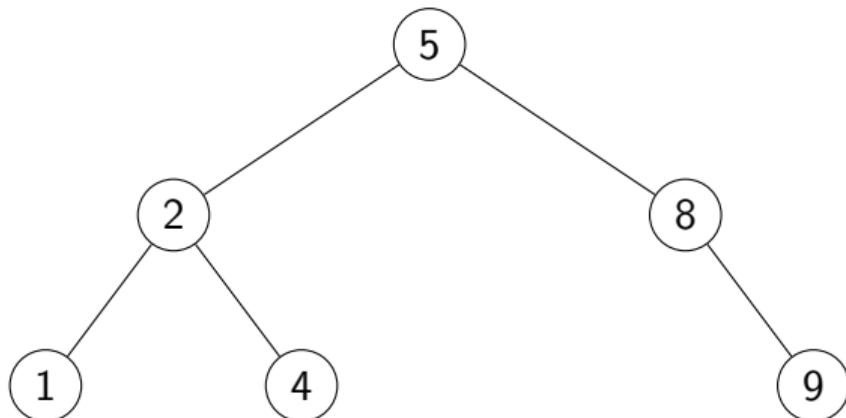
Inorder traversal

- List the left subtree, then the current node, then the right subtree
- Lists values in sorted order
- Use to print the tree

```
class Tree:  
    ...  
    # Inorder traversal  
    def inorder(self):  
        if self.isempty():  
            return []  
        else:  
            return(self.left.inorder())+  
                  [self.value]+  
                  self.right.inorder()  
  
    # Display Tree as a string  
    def __str__(self):  
        return(str(self.inorder()))
```

Inorder traversal

- List the left subtree, then the current node, then the right subtree
- Lists values in sorted order
- Use to print the tree



```
class Tree:  
    ...  
    # Inorder traversal  
    def inorder(self):  
        if self.isempty():  
            return []  
        else:  
            return(self.left.inorder())+  
                  [self.value]+  
                  self.right.inorder()  
  
    # Display Tree as a string  
    def __str__(self):  
        return(str(self.inorder()))
```

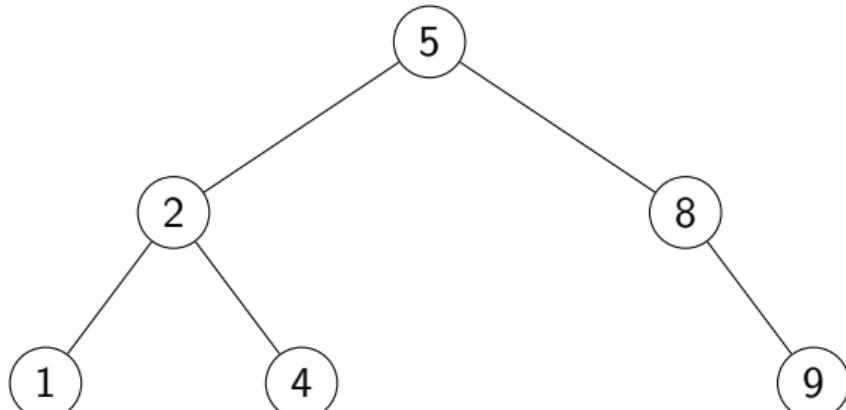
Find a value v

- Check value at current node
- If v smaller than current node, go left
- If v smaller than current node, go right
- Natural generalization of binary search

```
class Tree:  
    ...  
    # Check if value v occurs in tree  
    def find(self,v):  
        if self.isempty():  
            return(False)  
  
        if self.value == v:  
            return(True)  
  
        if v < self.value:  
            return(self.left.find(v))  
  
        if v > self.value:  
            return(self.right.find(v))
```

Find a value v

- Check value at current node
- If v smaller than current node, go left
- If v smaller than current node, go right
- Natural generalization of binary search



```
class Tree:  
    ...  
    # Check if value v occurs in tree  
    def find(self,v):  
        if self.isempty():  
            return(False)  
  
        if self.value == v:  
            return(True)  
  
        if v < self.value:  
            return(self.left.find(v))  
  
        if v > self.value:  
            return(self.right.find(v))
```

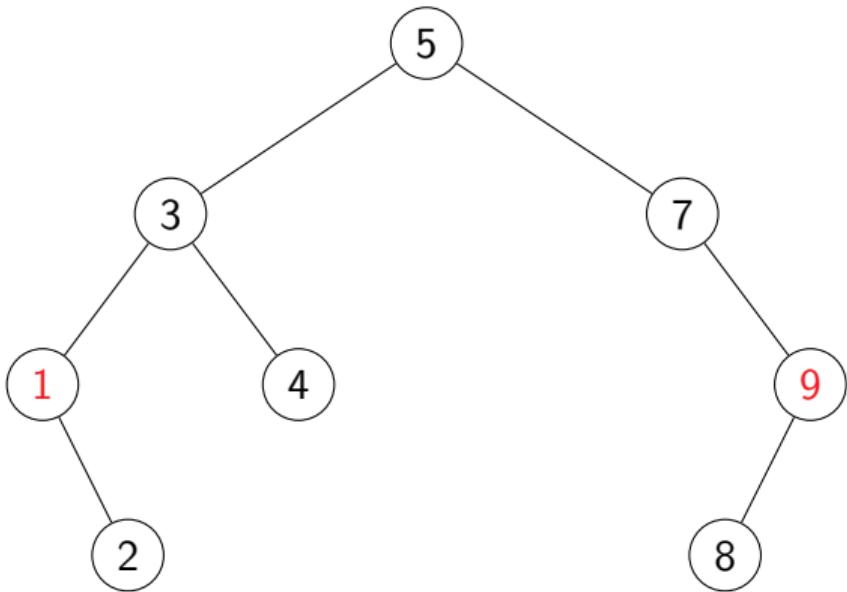
Minimum and maximum

- Minimum is left most node in the tree
- Maximum is right most node in the tree

```
class Tree:  
    ...  
    def minval(self):  
        if self.left.isempty():  
            return(self.value)  
        else:  
            return(self.left.minval())  
  
    def maxval(self):  
        if self.right.isempty():  
            return(self.value)  
        else:  
            return(self.right.maxval())
```

Minimum and maximum

- Minimum is left most node in the tree
- Maximum is right most node in the tree



```
class Tree:  
    ...  
    def minval(self):  
        if self.left.isempty():  
            return(self.value)  
        else:  
            return(self.left.minval())  
  
    def maxval(self):  
        if self.right.isempty():  
            return(self.value)  
        else:  
            return(self.right.maxval())
```

Insert a value v

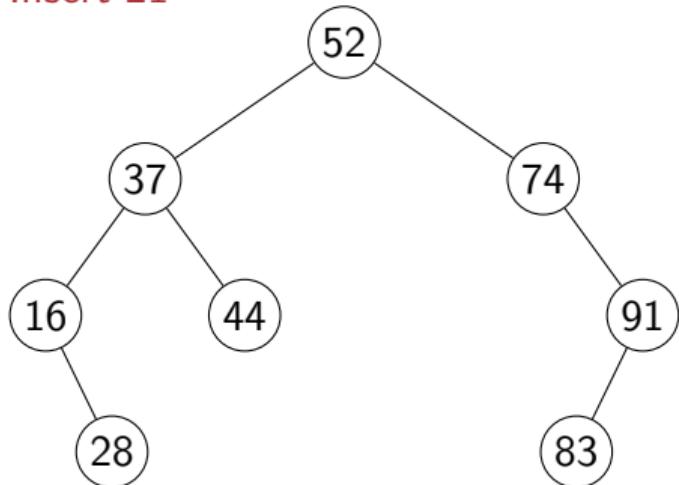
- Try to find v
- Insert at the position where `find` fails

```
class Tree:  
    ...  
    def insert(self,v):  
        if self.isempty():  
            self.value = v  
            self.left = Tree()  
            self.right = Tree()  
  
        if self.value == v:  
            return  
  
        if v < self.value:  
            self.left.insert(v)  
            return  
  
        if v > self.value:  
            self.right.insert(v)  
            return
```

Insert a value v

- Try to find v
- Insert at the position where `find` fails

Insert 21

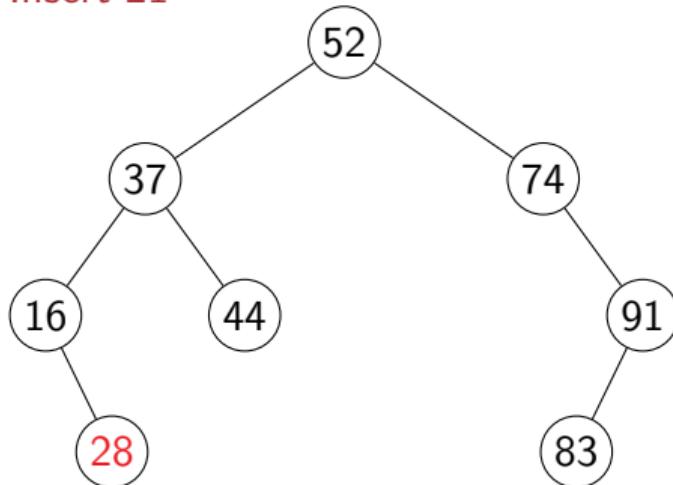


```
class Tree:  
    ...  
    def insert(self,v):  
        if self.isempty():  
            self.value = v  
            self.left = Tree()  
            self.right = Tree()  
  
        if self.value == v:  
            return  
  
        if v < self.value:  
            self.left.insert(v)  
            return  
  
        if v > self.value:  
            self.right.insert(v)  
            return
```

Insert a value v

- Try to find v
- Insert at the position where `find` fails

Insert 21

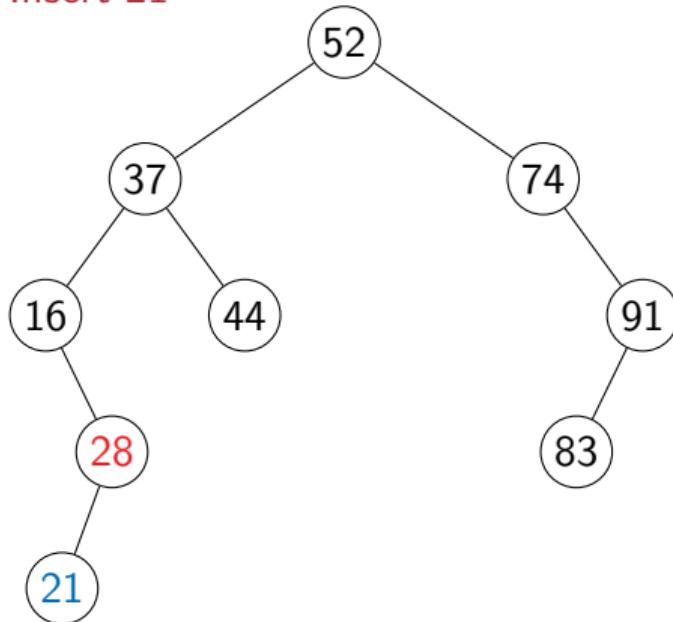


```
class Tree:  
    ...  
    def insert(self,v):  
        if self.isempty():  
            self.value = v  
            self.left = Tree()  
            self.right = Tree()  
  
        if self.value == v:  
            return  
  
        if v < self.value:  
            self.left.insert(v)  
            return  
  
        if v > self.value:  
            self.right.insert(v)  
            return
```

Insert a value v

- Try to find v
- Insert at the position where `find` fails

Insert 21

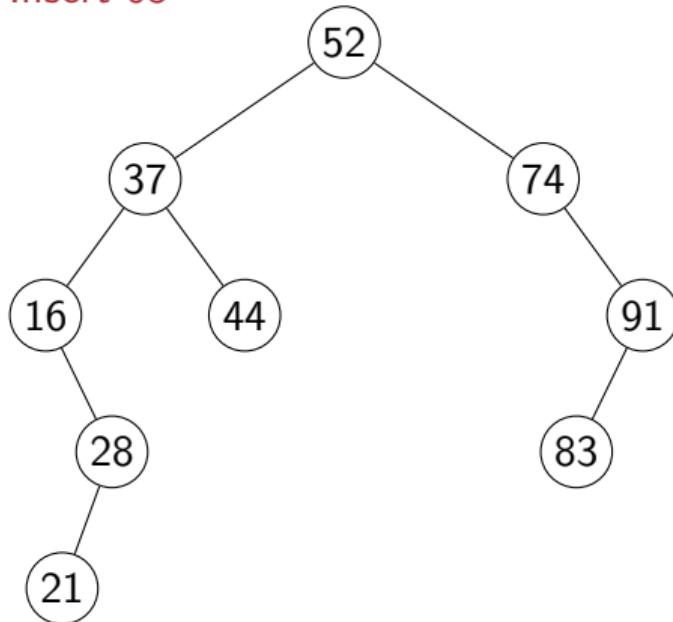


```
class Tree:  
    ...  
    def insert(self,v):  
        if self.isempty():  
            self.value = v  
            self.left = Tree()  
            self.right = Tree()  
  
        if self.value == v:  
            return  
  
        if v < self.value:  
            self.left.insert(v)  
            return  
  
        if v > self.value:  
            self.right.insert(v)  
            return
```

Insert a value v

- Try to find v
- Insert at the position where `find` fails

Insert 65

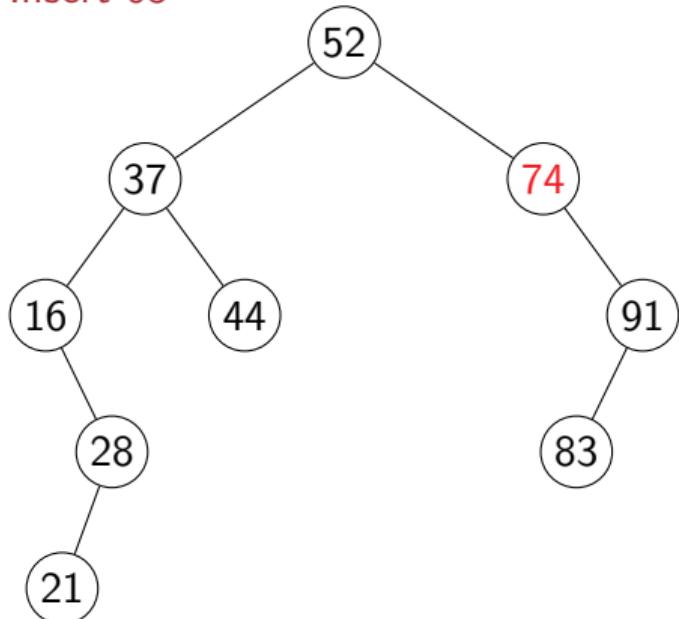


```
class Tree:  
    ...  
    def insert(self,v):  
        if self.isempty():  
            self.value = v  
            self.left = Tree()  
            self.right = Tree()  
  
        if self.value == v:  
            return  
  
        if v < self.value:  
            self.left.insert(v)  
            return  
  
        if v > self.value:  
            self.right.insert(v)  
            return
```

Insert a value v

- Try to find v
- Insert at the position where `find` fails

Insert 65

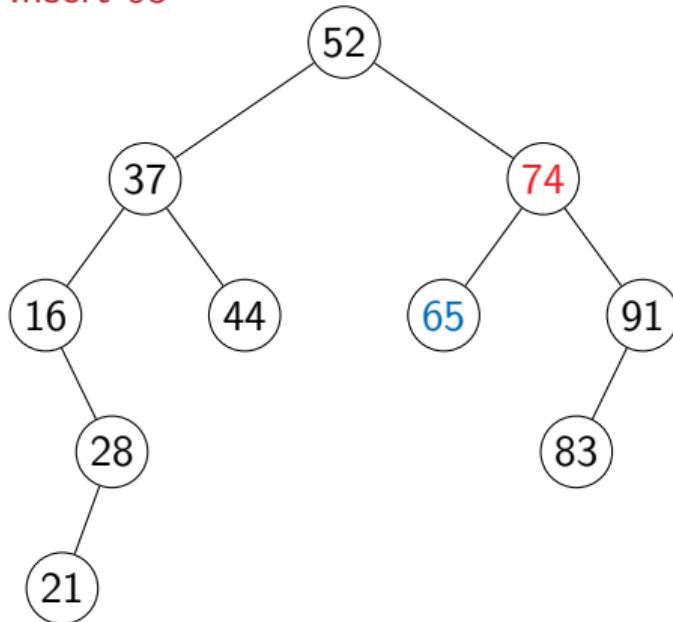


```
class Tree:  
    ...  
    def insert(self,v):  
        if self.isempty():  
            self.value = v  
            self.left = Tree()  
            self.right = Tree()  
  
        if self.value == v:  
            return  
  
        if v < self.value:  
            self.left.insert(v)  
            return  
  
        if v > self.value:  
            self.right.insert(v)  
            return
```

Insert a value v

- Try to find v
- Insert at the position where `find` fails

Insert 65

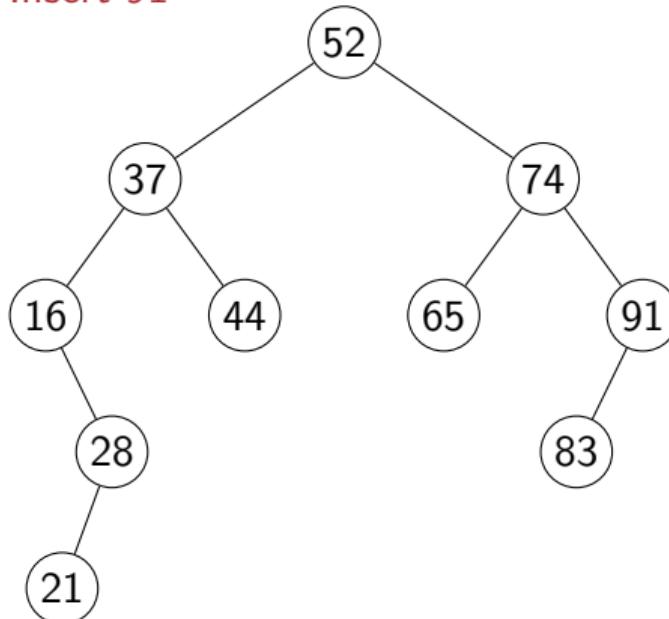


```
class Tree:  
    ...  
    def insert(self,v):  
        if self.isempty():  
            self.value = v  
            self.left = Tree()  
            self.right = Tree()  
  
        if self.value == v:  
            return  
  
        if v < self.value:  
            self.left.insert(v)  
            return  
  
        if v > self.value:  
            self.right.insert(v)  
            return
```

Insert a value v

- Try to find v
- Insert at the position where `find` fails

Insert 91

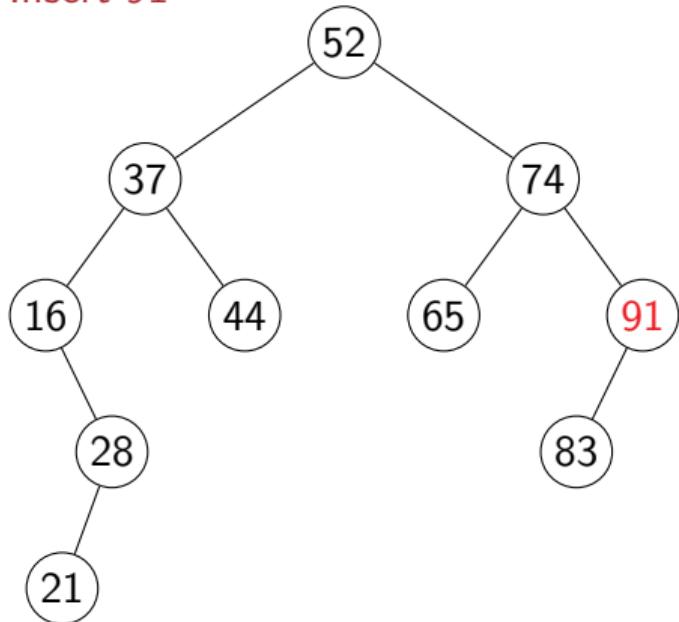


```
class Tree:  
    ...  
    def insert(self,v):  
        if self.isempty():  
            self.value = v  
            self.left = Tree()  
            self.right = Tree()  
  
        if self.value == v:  
            return  
  
        if v < self.value:  
            self.left.insert(v)  
            return  
  
        if v > self.value:  
            self.right.insert(v)  
            return
```

Insert a value v

- Try to find v
- Insert at the position where `find` fails

Insert 91



```
class Tree:  
    ...  
    def insert(self,v):  
        if self.isempty():  
            self.value = v  
            self.left = Tree()  
            self.right = Tree()  
  
        if self.value == v:  
            return  
  
        if v < self.value:  
            self.left.insert(v)  
            return  
  
        if v > self.value:  
            self.right.insert(v)  
            return
```

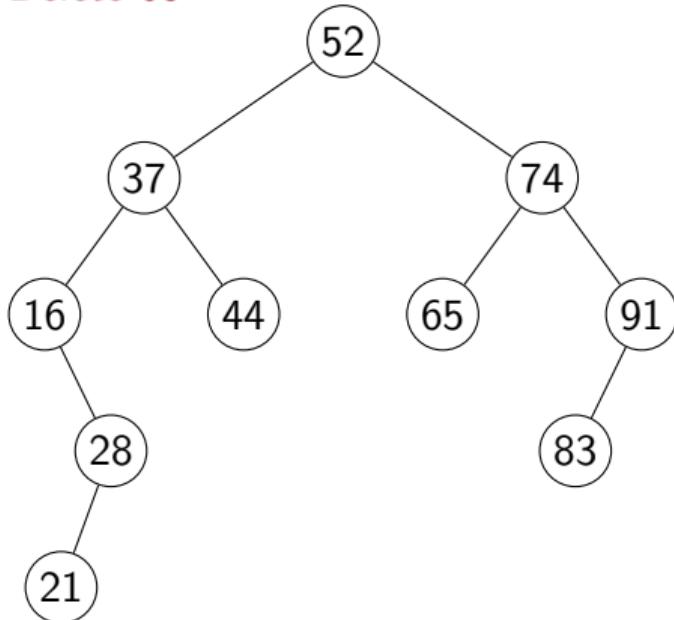
Delete a value v

- If `v` is present, delete
- Leaf node? No problem
- If only one child, promote that subtree
- Otherwise, replace `v` with `self.left.maxval()` and delete `self.left.maxval()`
 - `self.left.maxval()` has no right child

```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.copyright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
return
```

Delete a value v

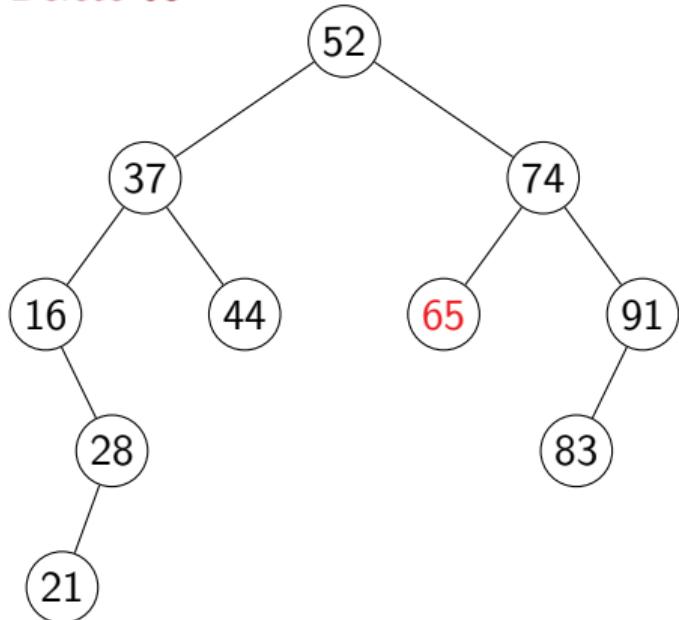
Delete 65



```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.coptright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
    return
```

Delete a value v

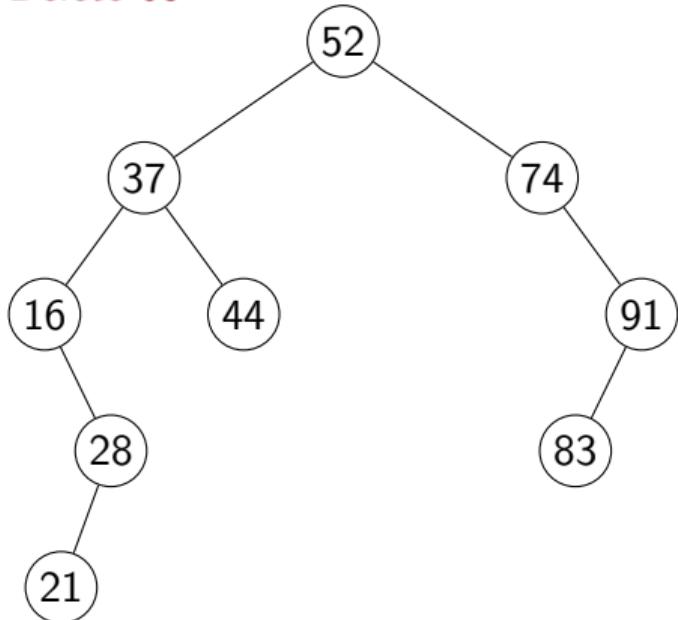
Delete 65



```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.coptright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
        return
```

Delete a value v

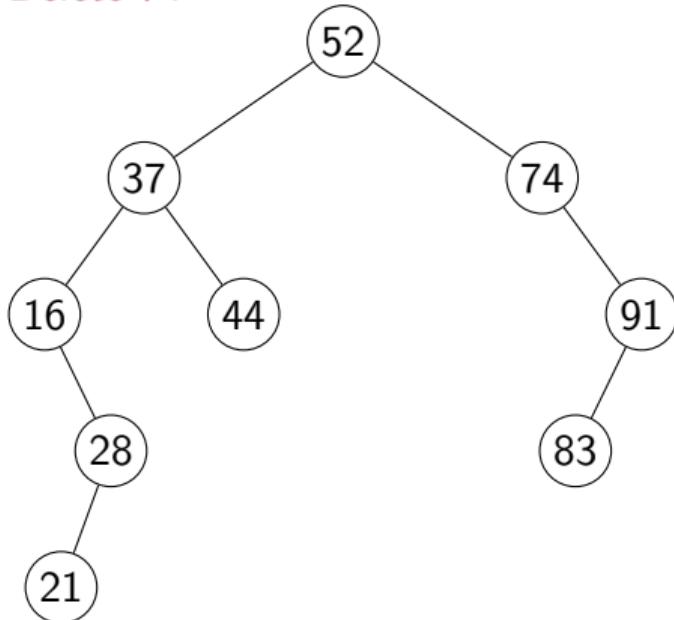
Delete 65



```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.copyright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
    return
```

Delete a value v

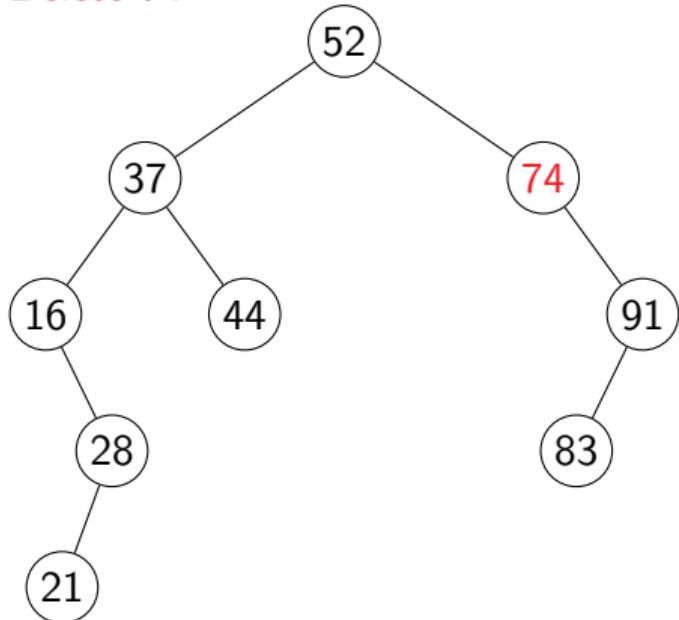
Delete 74



```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.copyright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
    return
```

Delete a value v

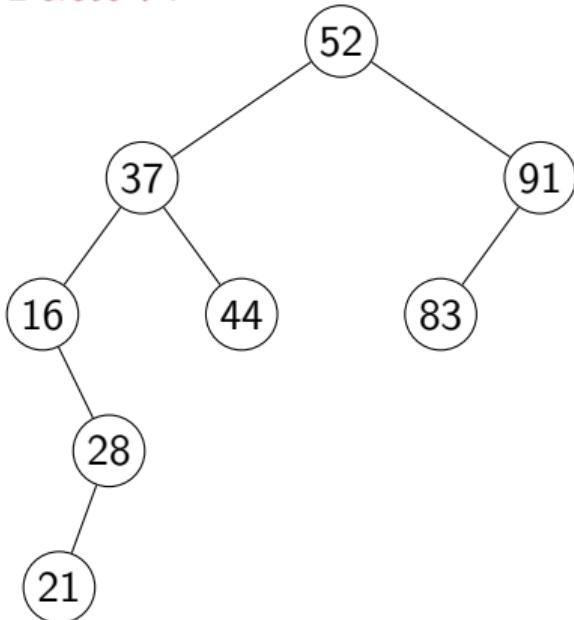
Delete 74



```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.copyright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
        return
```

Delete a value v

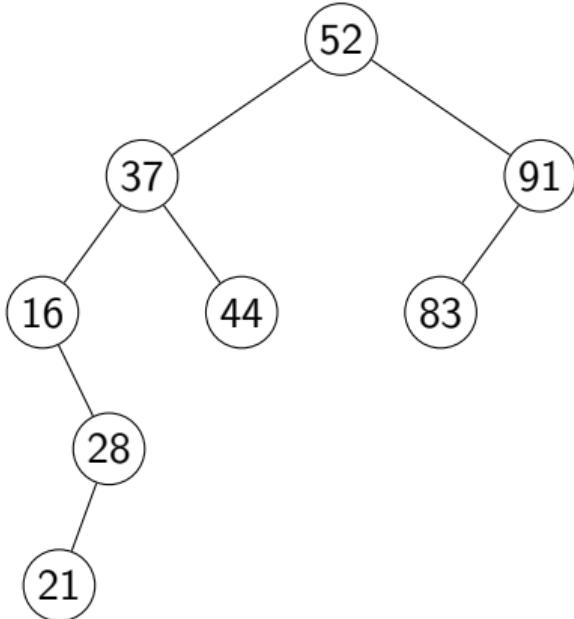
Delete 74



```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.copyright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
    return
```

Delete a value v

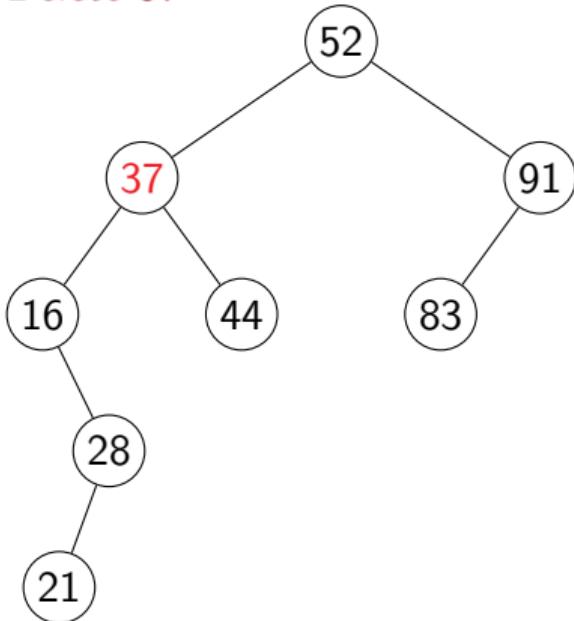
Delete 37



```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.copyright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
    return
```

Delete a value v

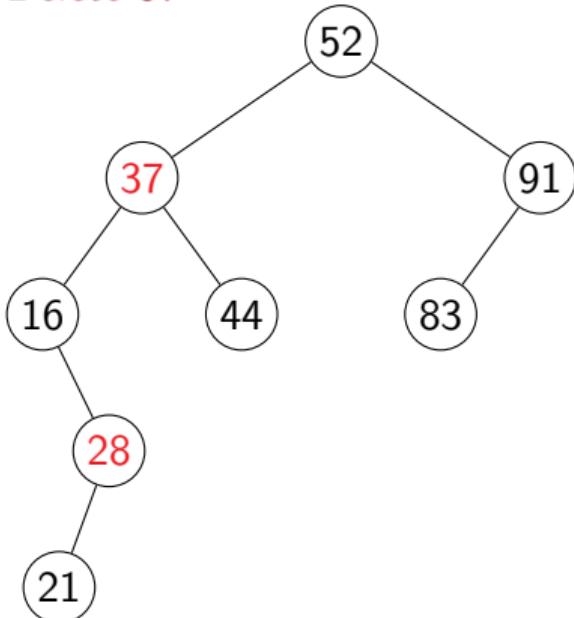
Delete 37



```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.copyright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
    return
```

Delete a value v

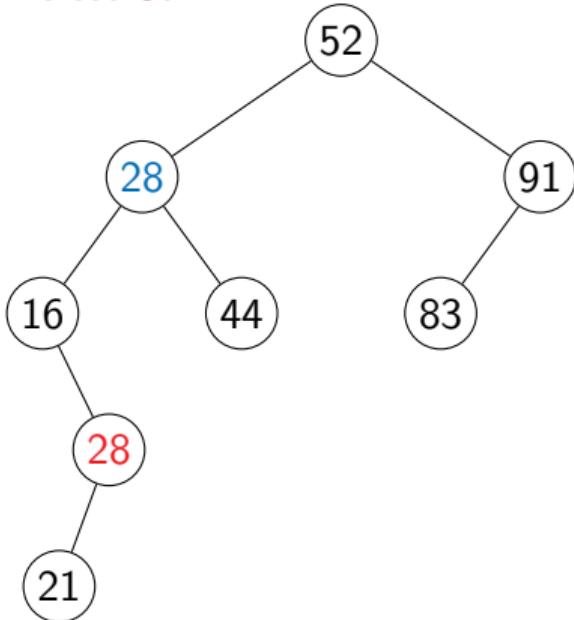
Delete 37



```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.copyright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
    return
```

Delete a value v

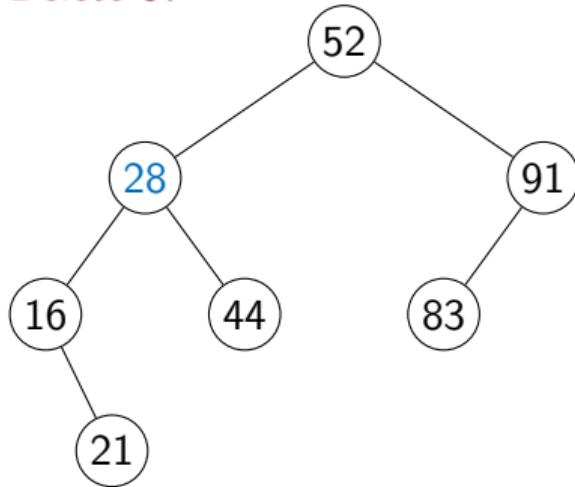
Delete 37



```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.copyright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
    return
```

Delete a value v

Delete 37



```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.copyright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
        return
```

Delete a value v

```
class Tree:  
    ...  
    def delete(self,v):  
        if self.isempty():  
            return  
        if v < self.value:  
            self.left.delete(v)  
            return  
        if v > self.value:  
            self.right.delete(v)  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.left.isempty():  
                self.copyright()  
            elif self.right.isempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
        return  
  
    # Convert leaf node to empty node  
    def makeempty(self):  
        self.value = None  
        self.left = None  
        self.right = None  
        return  
  
    # Promote left child  
    def copyleft(self):  
        self.value = self.left.value  
        self.right = self.left.right  
        self.left = self.left.left  
        return  
  
    # Promote right child  
    def copyright(self):  
        self.value = self.right.value  
        self.left = self.right.left  
        self.right = self.right.right  
        return
```

Complexity

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with n nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$
- Will see how to keep a tree balanced to ensure all operations remain $O(\log n)$

Balanced Search Trees

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 7

Search trees

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with n nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$

Search trees

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with n nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$
- How can we maintain balance as tree grows and shrinks

Defining balance

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with n nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$
- How can we maintain balance as tree grows and shrinks
- Left and right subtrees should be “equal”
 - Two possible measures: `size` and `height`

Defining balance

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with n nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$
- How can we maintain balance as tree grows and shrinks
- Left and right subtrees should be “equal”
 - Two possible measures: `size` and `height`
- `self.left.size()` and `self.right.size()` are equal?
 - Only possible for **complete** binary trees

Defining balance

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with n nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$
- How can we maintain balance as tree grows and shrinks

- Left and right subtrees should be “equal”
 - Two possible measures: `size` and `height`
- `self.left.size()` and `self.right.size()` are equal?
 - Only possible for **complete** binary trees
- `self.left.size()` and `self.right.size()` differ by at most 1?
 - Plausible, but difficult to maintain

Height balanced trees

- `self.height()` — number of nodes on longest path from root to leaf
 - `0` for empty tree
 - `1` for tree with only a root node
 - `1 + max` of heights of left and right subtrees, in general

Height balanced trees

- `self.height()` — number of nodes on longest path from root to leaf
 - 0 for empty tree
 - 1 for tree with only a root node
 - $1 + \max$ of heights of left and right subtrees, in general
- Height balance
 - `self.left.height()` and `self.right.height()` differ by at most 1
 - AVL trees — Adelson-Velskii, Landis

Height balanced trees

- `self.height()` — number of nodes on longest path from root to leaf
 - 0 for empty tree
 - 1 for tree with only a root node
 - $1 + \max$ of heights of left and right subtrees, in general
- Height balance
 - `self.left.height()` and `self.right.height()` differ by at most 1
 - AVL trees — Adelson-Velskii, Landis
- Does height balance guarantee $O(\log n)$ height?

Height balanced trees

- `self.height()` — number of nodes on longest path from root to leaf

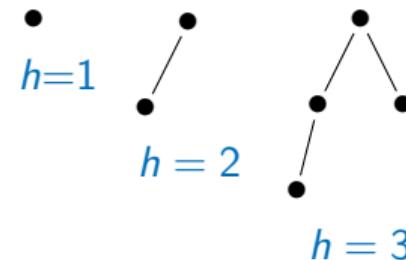
- 0 for empty tree
- 1 for tree with only a root node
- $1 + \max$ of heights of left and right subtrees, in general

- Height balance

- `self.left.height()` and `self.right.height()` differ by at most 1
- AVL trees — Adelson-Velskii, Landis

- Does height balance guarantee $O(\log n)$ height?

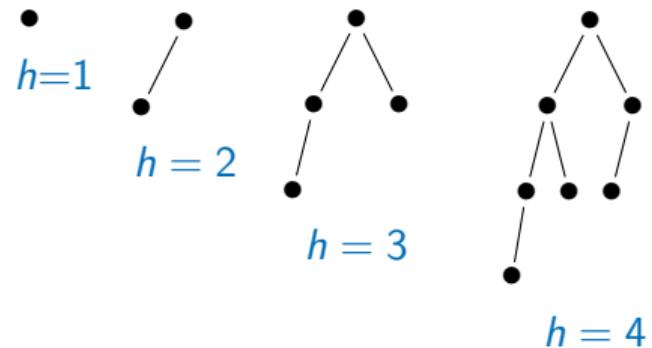
- Minimum size height-balanced trees



Height balanced trees

- `self.height()` — number of nodes on longest path from root to leaf
 - 0 for empty tree
 - 1 for tree with only a root node
 - $1 + \max$ of heights of left and right subtrees, in general
- Height balance
 - `self.left.height()` and `self.right.height()` differ by at most 1
 - AVL trees — Adelson-Velskii, Landis
- Does height balance guarantee $O(\log n)$ height?

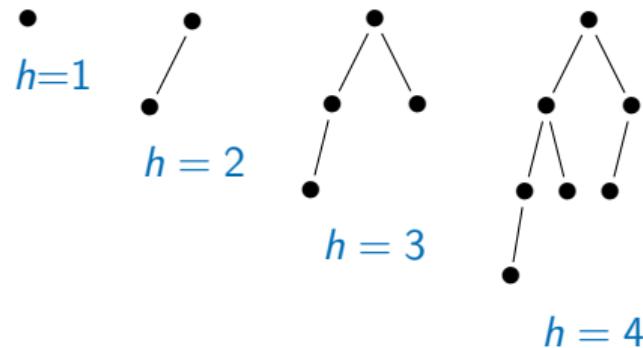
- Minimum size height-balanced trees



- General strategy to build a small balanced tree of height h
 - Smallest balanced tree of height $h-1$ as left subtree
 - Smallest balanced tree of height $h-2$ as right subtree

Height balanced trees

- Minimum size height-balanced trees

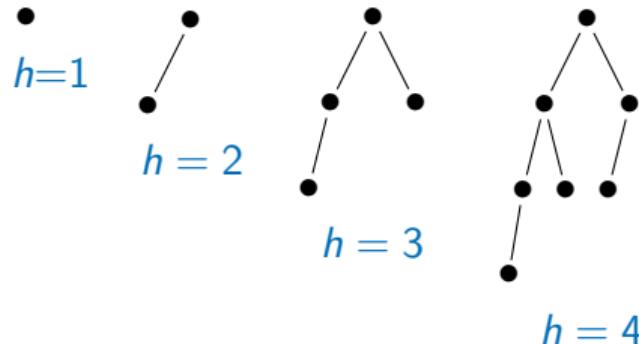


- General strategy to build a small balanced tree of height h

- Smallest balanced tree of height $h - 1$ as left subtree
- Smallest balanced tree of height $h - 2$ as right subtree

Height balanced trees

- Minimum size height-balanced trees



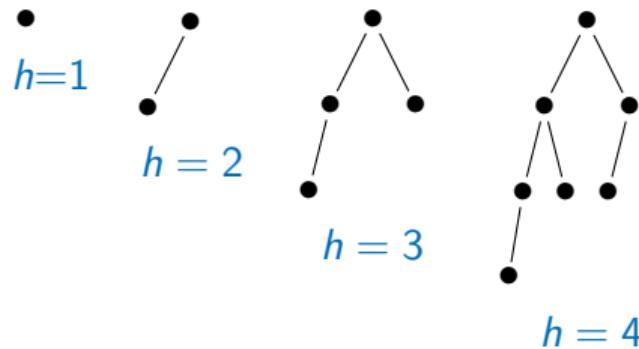
- $S(h)$, size of smallest height-balanced tree of height h

- General strategy to build a small balanced tree of height h

- Smallest balanced tree of height $h - 1$ as left subtree
- Smallest balanced tree of height $h - 2$ as right subtree

Height balanced trees

- Minimum size height-balanced trees



- General strategy to build a small balanced tree of height h

- Smallest balanced tree of height $h - 1$ as left subtree
- Smallest balanced tree of height $h - 2$ as right subtree

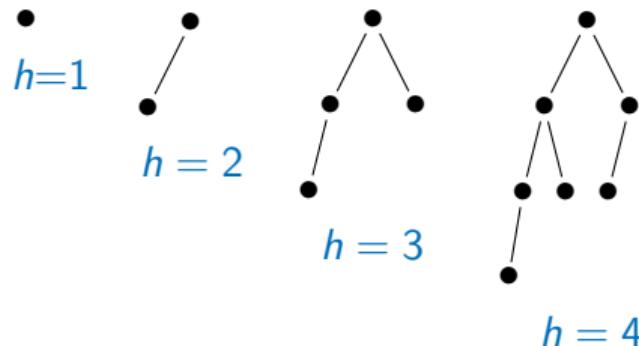
- $S(h)$, size of smallest height-balanced tree of height h

- Recurrence

- $S(0) = 0, S(1) = 1$
- $S(h) = 1 + S(h - 1) + S(h - 2)$

Height balanced trees

- Minimum size height-balanced trees



- General strategy to build a small balanced tree of height h

- Smallest balanced tree of height $h - 1$ as left subtree
- Smallest balanced tree of height $h - 2$ as right subtree

- $S(h)$, size of smallest height-balanced tree of height h

- Recurrence

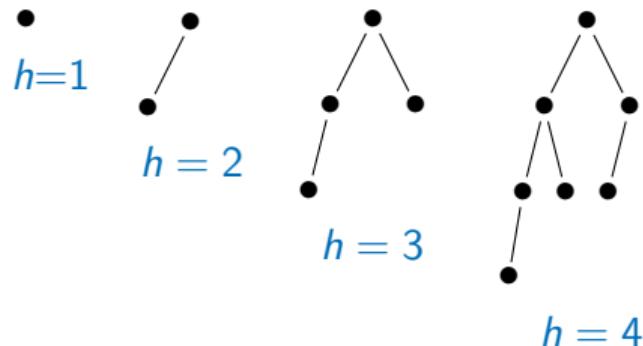
- $S(0) = 0, S(1) = 1$
- $S(h) = 1 + S(h - 1) + S(h - 2)$

- Compare to Fibonacci sequence

- $F(0) = 0, F(1) = 1$
- $F(n) = F(n - 1) + F(n - 2)$

Height balanced trees

- Minimum size height-balanced trees



- General strategy to build a small balanced tree of height h

- Smallest balanced tree of height $h - 1$ as left subtree
- Smallest balanced tree of height $h - 2$ as right subtree

- $S(h)$, size of smallest height-balanced tree of height h

- Recurrence

- $S(0) = 0, S(1) = 1$
- $S(h) = 1 + S(h - 1) + S(h - 2)$

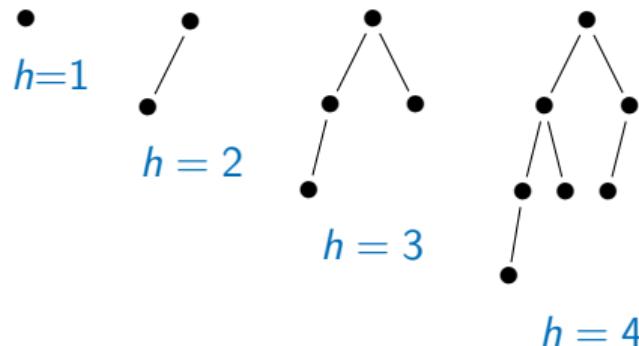
- Compare to Fibonacci sequence

- $F(0) = 0, F(1) = 1$
- $F(n) = F(n - 1) + F(n - 2)$

- $S(h)$ grows exponentially with h

Height balanced trees

- Minimum size height-balanced trees



- General strategy to build a small balanced tree of height h

- Smallest balanced tree of height $h - 1$ as left subtree
- Smallest balanced tree of height $h - 2$ as right subtree

- $S(h)$, size of smallest height-balanced tree of height h

- Recurrence

- $S(0) = 0, S(1) = 1$
- $S(h) = 1 + S(h - 1) + S(h - 2)$

- Compare to Fibonacci sequence

- $F(0) = 0, F(1) = 1$
- $F(n) = F(n - 1) + F(n - 2)$

- $S(h)$ grows exponentially with h

- For size n , h is $O(\log n)$

Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`

Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$

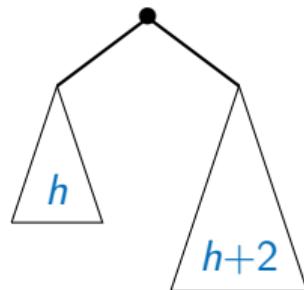
Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to -2 or $+2$

Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to -2 or $+2$

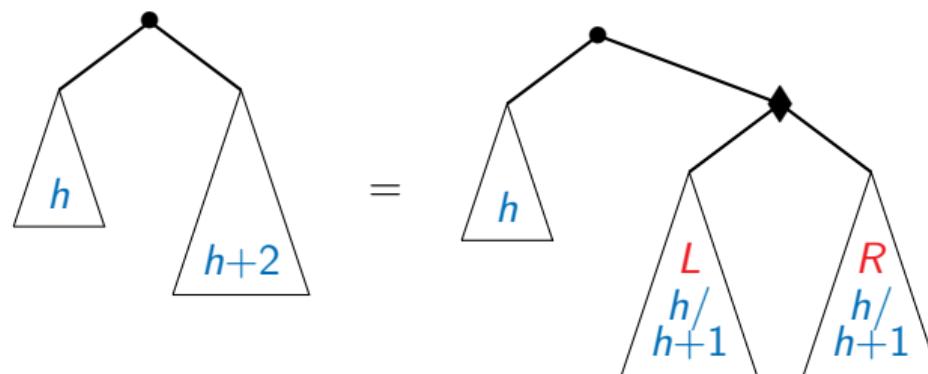
Left rotation



Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to -2 or $+2$

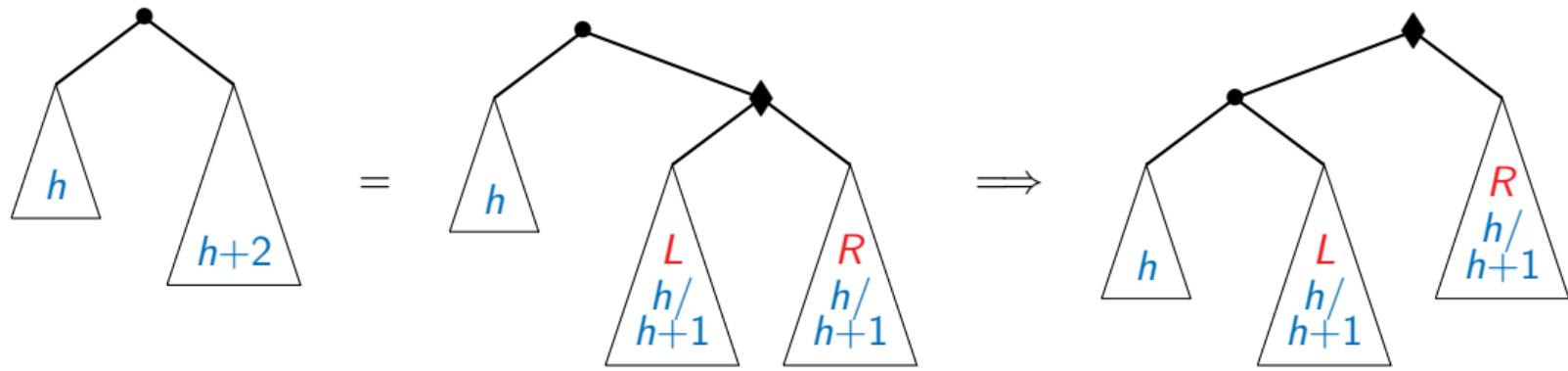
Left rotation



Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to -2 or $+2$

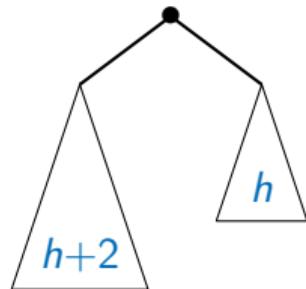
Left rotation — converts slope -2 to $\{0, 1, 2\}$



Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to -2 or $+2$

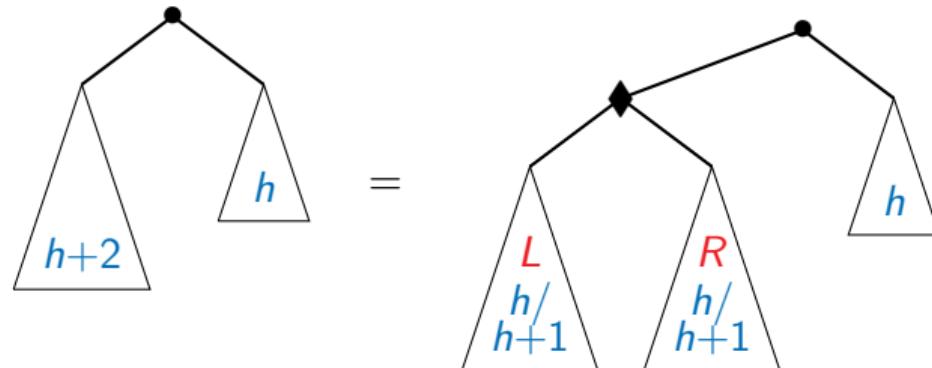
Right rotation



Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to -2 or $+2$

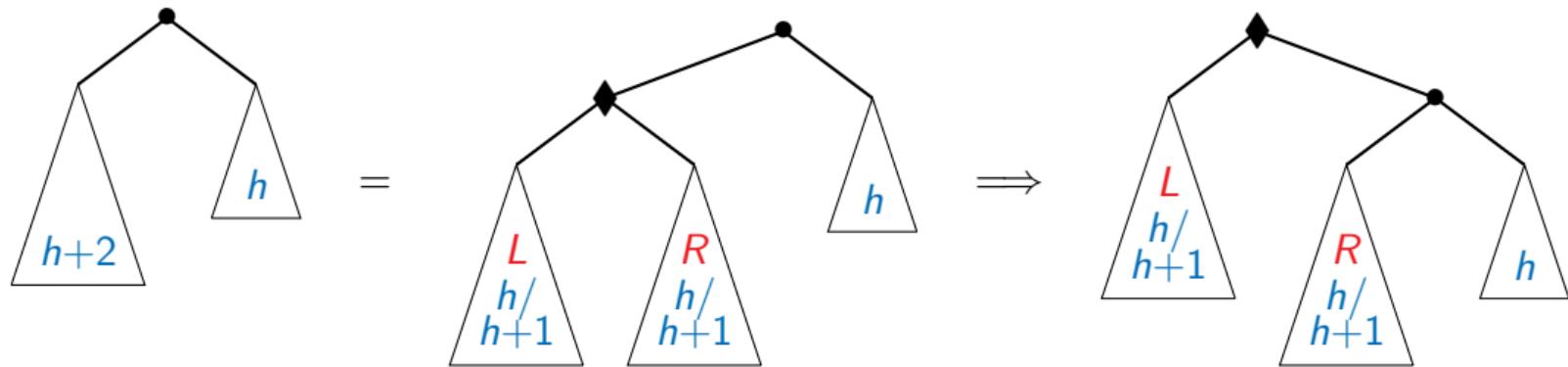
Right rotation



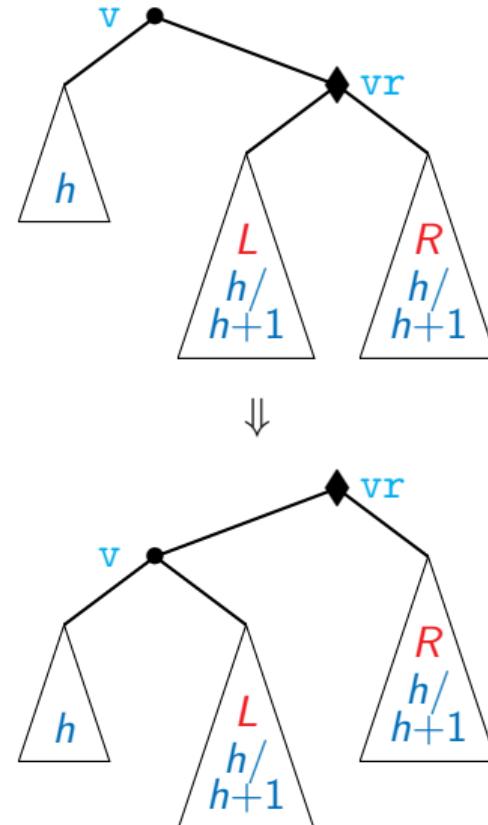
Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to -2 or $+2$

Right rotation — converts slope $+2$ to $\{-2, -1, 0\}$

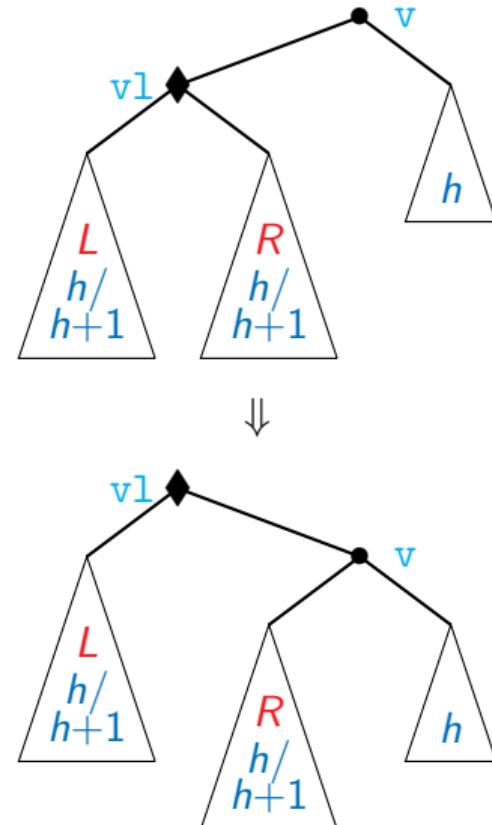


Implementing rotations



```
class Tree:  
    ...  
  
    def leftrotate(self):  
        v = self.value  
        vr = self.right.value  
        tl = self.left  
        trl = self.right.left  
        trr = self.right.right  
  
        newleft = Tree(v)  
        newleft.left = tl  
        newleft.right = trl  
  
        self.value = vr  
        self.right = trr  
        self.left = newleft  
        return
```

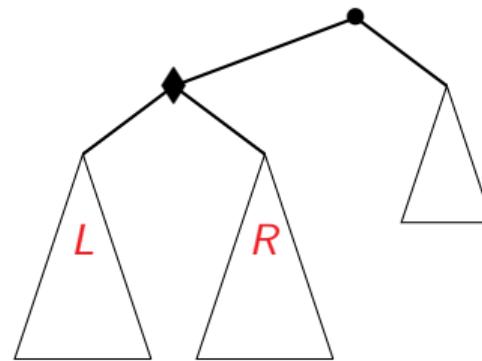
Implementing rotations



```
class Tree:  
    ...  
  
    def rightrotate(self):  
        v = self.value  
        vl = self.left.value  
        tll = self.left.left  
        tlr = self.left.right  
        tr = self.right  
  
        newright = Tree(v)  
        newright.left = tlr  
        newright.right = tr  
  
        self.value = vl  
        self.left = tll  
        self.right = newright  
        return
```

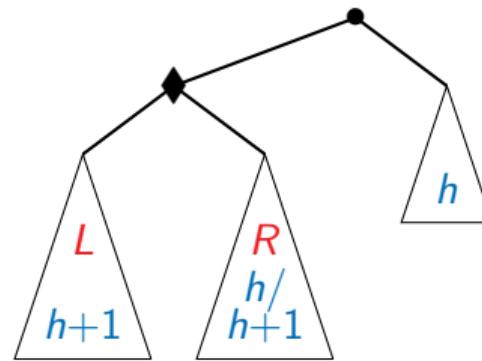
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced



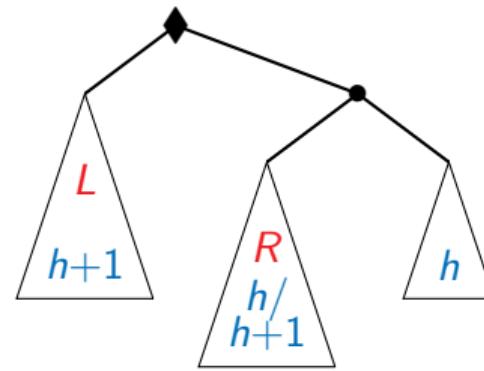
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at \blacklozenge is in $\{0, 1\}$



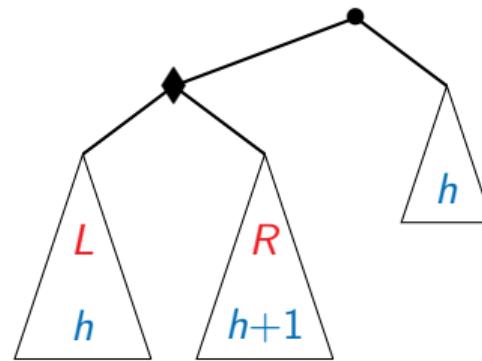
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at \blacklozenge is in $\{0, 1\}$
 - Rotate right at \bullet
 - All nodes are balanced



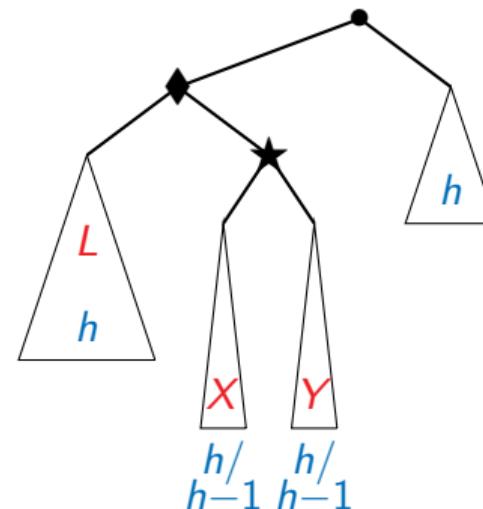
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at \diamond is in $\{0, 1\}$
 - Rotate right at \bullet
 - All nodes are balanced
- Case 2: Slope at \diamond is -1



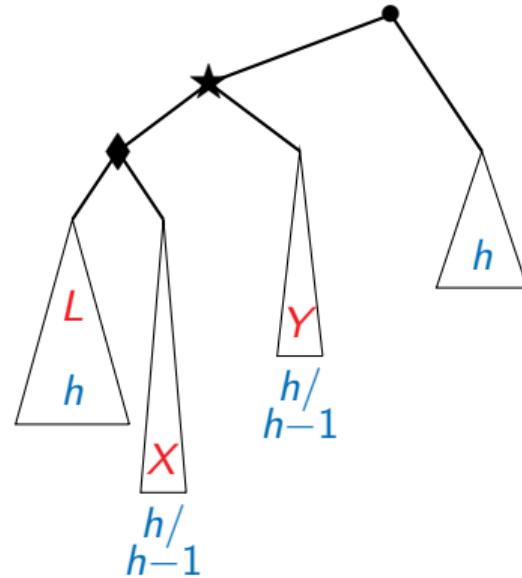
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at \blacklozenge is in $\{0, 1\}$
 - Rotate right at \bullet
 - All nodes are balanced
- Case 2: Slope at \blacklozenge is -1
 - Expand R



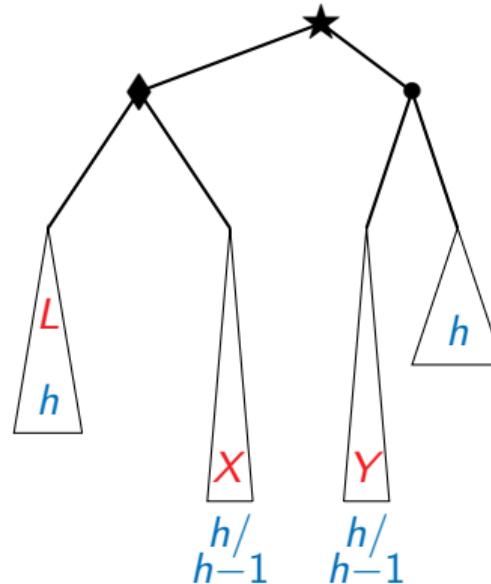
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
 - Case 1: Slope at \diamond is in $\{0, 1\}$
 - Rotate right at \bullet
 - All nodes are balanced
 - Case 2: Slope at \diamond is -1
 - Expand R
 - Rotate left at \diamond



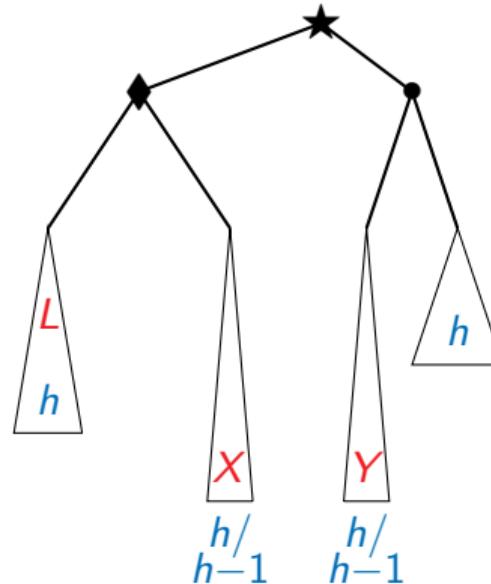
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at \blacklozenge is in $\{0, 1\}$
 - Rotate right at \bullet
 - All nodes are balanced
- Case 2: Slope at \blacklozenge is -1
 - Expand R
 - Rotate left at \blacklozenge
 - Rotate left at \bullet



Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at \blacklozenge is in $\{0, 1\}$
 - Rotate right at \bullet
 - All nodes are balanced
- Case 2: Slope at \blacklozenge is -1
 - Expand R
 - Rotate left at \blacklozenge
 - Rotate left at \bullet
- Rebalance with root slope -2 is symmetric



Update insert() and delete()

- Use the rebalancing strategy to define a function `rebalance()`
- Rebalance each time the tree is modified
- Automatically rebalances bottom up

```
class Tree:  
    ...  
    def insert(self,v):  
        if self.isempty():  
            self.value = v  
            self.left = Tree()  
            self.right = Tree()  
  
        if self.value == v:  
            return  
  
        if v < self.value:  
            self.left.insert(v)  
            self.left.rebalance()  
            return  
  
        if v > self.value:  
            self.right.insert(v)  
            self.right.rebalance()  
            return
```

Update insert() and delete()

- Use the rebalancing strategy to define a function `rebalance()`
- Rebalance each time the tree is modified
- Automatically rebalances bottom up

```
class Tree:  
    ...  
    def delete(self,v):  
        ...  
        if v < self.value:  
            self.left.delete(v)  
            self.left.rebalance()  
            return  
        if v > self.value:  
            self.right.delete(v)  
            self.right.rebalance()  
            return  
        if v == self.value:  
            if self.isleaf():  
                self.makeempty()  
            elif self.leftisempty():  
                self.copyright()  
            elif self.rightisempty():  
                self.copyleft()  
            else:  
                self.value = self.left.maxval()  
                self.left.delete(self.left.maxval())  
        return
```

Computing slope

- To compute the slope we need heights of subtrees
- But, computing height is $O(n)$

```
class Tree:  
    ...  
    def height(self):  
        if self.isempty():  
            return(0)  
        else:  
            return(1 +  
                  max(self.left.height(),  
                       self.right.height()))
```

Computing slope

- To compute the slope we need heights of subtrees
- But, computing height is $O(n)$
- Instead, maintain a field `self.height`

```
class Tree:  
    ...  
    def height(self):  
        if self.isempty():  
            return(0)  
        else:  
            return(1 +  
                  max(self.left.height(),  
                       self.right.height()))
```

Computing slope

- To compute the slope we need heights of subtrees
- But, computing height is $O(n)$
- Instead, maintain a field `self.height`
- After each modification, update `self.height` based on `self.left.height`, `self.right.height`

```
class Tree:  
    ...  
    def insert(self,v):  
        ...  
        if v < self.value:  
            self.left.insert(v)  
            self.left.rebalance()  
            self.height = 1 +  
                max(self.left.height,  
                     self.right.height)  
        return  
  
        if v > self.value:  
            self.right.insert(v)  
            self.right.rebalance()  
            self.height = 1 +  
                max(self.left.height,  
                     self.right.height)  
        return
```

Summary

- Using rotations, we can maintain height balance
- Height balanced trees have height $O(\log n)$
- `find()`, `insert()` and `delete()` all walk down a single path, take time $O(\log n)$

Greedy Algorithms: Interval Scheduling

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 7

Greedy Algorithms

- Need to make a sequence of choices to achieve a global optimum

Greedy Algorithms

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion

Greedy Algorithms

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion
- Never go back and revise an earlier decision

Greedy Algorithms

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion
- Never go back and revise an earlier decision
- Drastically reduces space to search for solutions

Greedy Algorithms

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion
- Never go back and revise an earlier decision
- Drastically reduces space to search for solutions
- How to prove that local choices achieve global optimum?

Greedy Algorithms

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion
- Never go back and revise an earlier decision
- Drastically reduces space to search for solutions
- How to prove that local choices achieve global optimum?

Examples

- Dijkstra's algorithm
 - Local rule: freeze the distance to nearest unvisited vertex
 - Global optimum: distance assigned to each vertex is shortest distance from source

Greedy Algorithms

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion
- Never go back and revise an earlier decision
- Drastically reduces space to search for solutions
- How to prove that local choices achieve global optimum?

Examples

- Prim's algorithm
 - Local rule: add to the spanning tree nearest non-tree vertex
 - Global optimum: final spanning tree is minimum cost spanning tree

Greedy Algorithms

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion
- Never go back and revise an earlier decision
- Drastically reduces space to search for solutions
- How to prove that local choices achieve global optimum?

Examples

- Kruskal's algorithm
 - Local rule: add to the current set of edges the smallest edge that does not form a cycle
 - Global optimum: final spanning tree is minimum cost spanning tree

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slot for instructor i starts at $s(i)$ and finishes at $f(i)$

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slot for instructor i starts at $s(i)$ and finishes at $f(i)$
- Slots may overlap, so not all bookings can be honoured

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slot for instructor i starts at $s(i)$ and finishes at $f(i)$
- Slots may overlap, so not all bookings can be honoured
- Choose a subset of bookings to maximize the number of teachers who get to use the room

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slot for instructor i starts at $s(i)$ and finishes at $f(i)$
- Slots may overlap, so not all bookings can be honoured
- Choose a subset of bookings to maximize the number of teachers who get to use the room

Greedy approach

- Pick the next booking to allot based on a local strategy

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slot for instructor i starts at $s(i)$ and finishes at $f(i)$
- Slots may overlap, so not all bookings can be honoured
- Choose a subset of bookings to maximize the number of teachers who get to use the room

Greedy approach

- Pick the next booking to allot based on a local strategy
- Remove all bookings that overlap with the chosen slot

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slot for instructor i starts at $s(i)$ and finishes at $f(i)$
- Slots may overlap, so not all bookings can be honoured
- Choose a subset of bookings to maximize the number of teachers who get to use the room

Greedy approach

- Pick the next booking to allot based on a local strategy
- Remove all bookings that overlap with the chosen slot
- Argue that this sequence of bookings will maximize the number of teachers who get to use the room

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slot for instructor i starts at $s(i)$ and finishes at $f(i)$
- Slots may overlap, so not all bookings can be honoured
- Choose a subset of bookings to maximize the number of teachers who get to use the room

Greedy approach

- Pick the next booking to allot based on a local strategy
- Remove all bookings that overlap with the chosen slot
- Argue that this sequence of bookings will maximize the number of teachers who get to use the room
- What is a sound local strategy?

Greedy strategies for interval scheduling

■ Strategy 1

Choose the booking whose starting time
is earliest

Greedy strategies for interval scheduling

■ Strategy 1

Choose the booking whose starting time is earliest

Counterexample



Greedy strategies for interval scheduling

- **Strategy 1**

Choose the booking whose starting time
is earliest

- **Strategy 2**

Choose the booking spanning the
shortest interval

Greedy strategies for interval scheduling

- **Strategy 1**

Choose the booking whose starting time is earliest

Counterexample



- **Strategy 2**

Choose the booking spanning the shortest interval

Greedy strategies for interval scheduling

- **Strategy 1**

Choose the booking whose starting time is earliest

- **Strategy 2**

Choose the booking spanning the shortest interval

- **Strategy 3**

Choose the booking that overlaps with minimum number of other bookings

Greedy strategies for interval scheduling

- **Strategy 1**

Choose the booking whose starting time is earliest

- **Strategy 2**

Choose the booking spanning the shortest interval

- **Strategy 3**

Choose the booking that overlaps with minimum number of other bookings

Counterexample



Greedy strategies for interval scheduling

- **Strategy 1**

Choose the booking whose starting time is earliest

- **Strategy 2**

Choose the booking spanning the shortest interval

- **Strategy 3**

Choose the booking that overlaps with minimum number of other bookings

- **Strategy 4**

Choose the booking whose finish time is the earliest

Greedy strategies for interval scheduling

- **Strategy 1**

Choose the booking whose starting time is earliest

- **Strategy 2**

Choose the booking spanning the shortest interval

- **Strategy 3**

Choose the booking that overlaps with minimum number of other bookings

- **Strategy 4**

Choose the booking whose finish time is the earliest

- Counterexample? Proof of correctness?

Greedy algorithm for interval scheduling

- B is the set of bookings

Greedy algorithm for interval scheduling

- B is the set of bookings
- A is the set of accepted bookings
 - Initially, A is empty

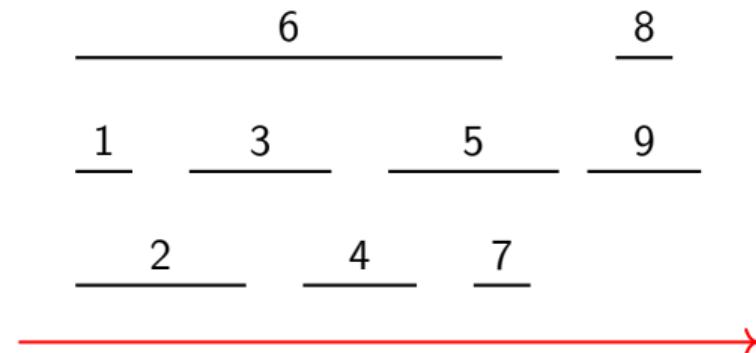
Greedy algorithm for interval scheduling

- B is the set of bookings
- A is the set of accepted bookings
 - Initially, A is empty
- While B is not empty
 - Pick $b \in B$ with earliest finishing time
 - Add b to A
 - Remove from B all bookings that overlap with b

Greedy algorithm for interval scheduling

- B is the set of bookings
- A is the set of accepted bookings
 - Initially, A is empty
- While B is not empty
 - Pick $b \in B$ with earliest finishing time
 - Add b to A
 - Remove from B all bookings that overlap with b

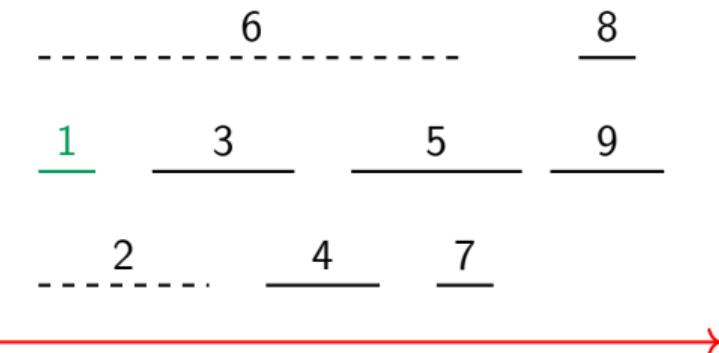
The algorithm in action



Greedy algorithm for interval scheduling

- B is the set of bookings
- A is the set of accepted bookings
 - Initially, A is empty
- While B is not empty
 - Pick $b \in B$ with earliest finishing time
 - Add b to A
 - Remove from B all bookings that overlap with b

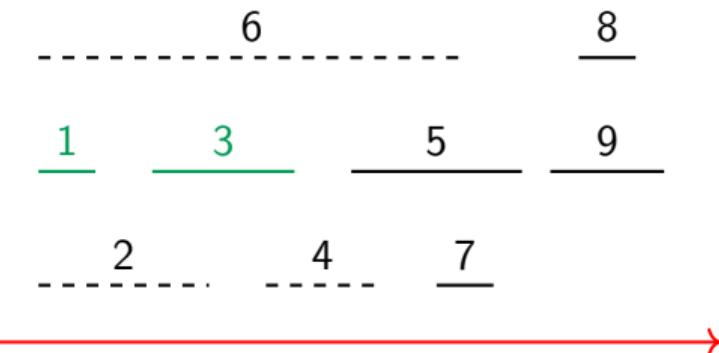
The algorithm in action



Greedy algorithm for interval scheduling

- B is the set of bookings
- A is the set of accepted bookings
 - Initially, A is empty
- While B is not empty
 - Pick $b \in B$ with earliest finishing time
 - Add b to A
 - Remove from B all bookings that overlap with b

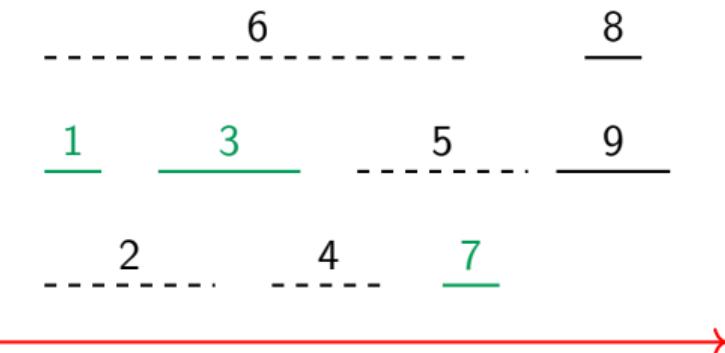
The algorithm in action



Greedy algorithm for interval scheduling

- B is the set of bookings
- A is the set of accepted bookings
 - Initially, A is empty
- While B is not empty
 - Pick $b \in B$ with earliest finishing time
 - Add b to A
 - Remove from B all bookings that overlap with b

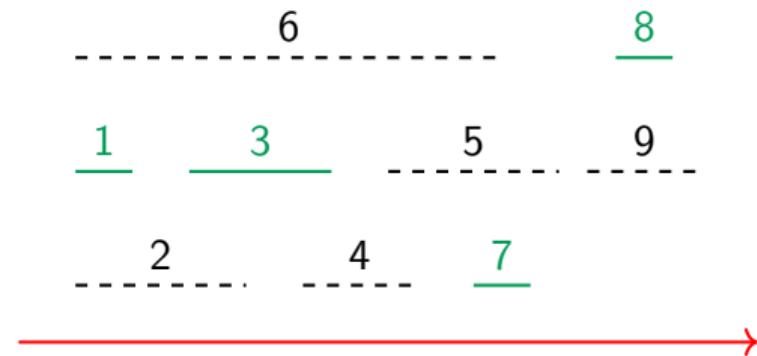
The algorithm in action



Greedy algorithm for interval scheduling

- B is the set of bookings
- A is the set of accepted bookings
 - Initially, A is empty
- While B is not empty
 - Pick $b \in B$ with earliest finishing time
 - Add b to A
 - Remove from B all bookings that overlap with b

The algorithm in action



Correctness

- Our algorithm produces a solution A

Correctness

- Our algorithm produces a solution A
- Let O be an optimal set of accepted bookings

Correctness

- Our algorithm produces a solution A
- Let O be an optimal set of accepted bookings
- A and O need not be identical
 - Could have multiple allocations of the same size

- Our algorithm produces a solution A
- Let O be an optimal set of accepted bookings
- A and O need not be identical
 - Could have multiple allocations of the same size
- Just show that $|A| = |O|$
 - Both sets of bookings are of the same size

Correctness

- Our algorithm produces a solution A
 - Let O be an optimal set of accepted bookings
 - A and O need not be identical
 - Could have multiple allocations of the same size
 - Just show that $|A| = |O|$
 - Both sets of bookings are of the same size
-
- Let $A = \{i_1, i_2, \dots, i_k\}$
 - Assume sorted
 - $f(i_1) \leq s(i_2), f(i_2) \leq s(i_3), \dots$

Correctness

- Our algorithm produces a solution A
 - Let O be an optimal set of accepted bookings
 - A and O need not be identical
 - Could have multiple allocations of the same size
 - Just show that $|A| = |O|$
 - Both sets of bookings are of the same size
-
- Let $A = \{i_1, i_2, \dots, i_k\}$
 - Assume sorted
 - $f(i_1) \leq s(i_2), f(i_2) \leq s(i_3), \dots$
 - Let $O = \{j_1, j_2, \dots, j_m\}$
 - Also sorted
 - $f(j_1) \leq s(j_2), f(j_2) \leq s(j_3), \dots$

Correctness

- Our algorithm produces a solution A
 - Let O be an optimal set of accepted bookings
 - A and O need not be identical
 - Could have multiple allocations of the same size
 - Just show that $|A| = |O|$
 - Both sets of bookings are of the same size
-
- Let $A = \{i_1, i_2, \dots, i_k\}$
 - Assume sorted
 - $f(i_1) \leq s(i_2), f(i_2) \leq s(i_3), \dots$
 - Let $O = \{j_1, j_2, \dots, j_m\}$
 - Also sorted
 - $f(j_1) \leq s(j_2), f(j_2) \leq s(j_3), \dots$
 - Our goal is to show that $k = m$

Greedy algorithm stays ahead

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$

Greedy algorithm stays ahead

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- **Claim** For each $\ell \leq k$, $f(i_\ell) \leq f(j_\ell)$

Greedy algorithm stays ahead

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- **Claim** For each $\ell \leq k$, $f(i_\ell) \leq f(j_\ell)$
- **Proof** By induction of ℓ

Greedy algorithm stays ahead

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- **Claim** For each $\ell \leq k$, $f(i_\ell) \leq f(j_\ell)$
- **Proof** By induction of ℓ
- Base case: $\ell = 1$

By greedy strategy, i_1 has earliest overall finish time

Greedy algorithm stays ahead

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- **Claim** For each $\ell \leq k$, $f(i_\ell) \leq f(j_\ell)$
- **Proof** By induction of ℓ
- Base case: $\ell = 1$

By greedy strategy, i_1 has earliest overall finish time

- Induction step: Assume $f(i_{\ell-1}) \leq f(j_{\ell-1})$

Greedy algorithm stays ahead

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- **Claim** For each $\ell \leq k$, $f(i_\ell) \leq f(j_\ell)$
- **Proof** By induction of ℓ
- Base case: $\ell = 1$

By greedy strategy, i_1 has earliest overall finish time

- Induction step: Assume $f(i_{\ell-1}) \leq f(j_{\ell-1})$
 - $f(i_{\ell-1}) \leq f(j_{\ell-1}) \leq s(j_\ell)$

Greedy algorithm stays ahead

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- **Claim** For each $\ell \leq k$, $f(i_\ell) \leq f(j_\ell)$
- **Proof** By induction of ℓ
- Base case: $\ell = 1$

By greedy strategy, i_1 has earliest overall finish time

- Induction step: Assume $f(i_{\ell-1}) \leq f(j_{\ell-1})$
 - $f(i_{\ell-1}) \leq f(j_{\ell-1}) \leq s(j_\ell)$
 - If $f(j_\ell) < f(i_\ell)$, greedy strategy would pick j_ℓ

Greedy algorithm stays ahead

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- **Claim** For each $\ell \leq k$, $f(i_\ell) \leq f(j_\ell)$
- **Proof** By induction of ℓ
- Base case: $\ell = 1$

By greedy strategy, i_1 has earliest overall finish time

- Induction step: Assume $f(i_{\ell-1}) \leq f(j_{\ell-1})$
 - $f(i_{\ell-1}) \leq f(j_{\ell-1}) \leq s(j_\ell)$
 - If $f(j_\ell) < f(i_\ell)$, greedy strategy would pick j_ℓ
 - We must have $f(i_\ell) \leq f(j_\ell)$

Greedy strategy is optimal

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$

Greedy strategy is optimal

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- Suppose $m > k$

Greedy strategy is optimal

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- Suppose $m > k$
- We know $f(i_k) \leq f(j_k)$

Greedy strategy is optimal

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- Suppose $m > k$
- We know $f(i_k) \leq f(j_k)$
- Greedy strategy stops when B is empty

Greedy strategy is optimal

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- Suppose $m > k$
- We know $f(i_k) \leq f(j_k)$
- Greedy strategy stops when B is empty
 - Consider request j_{k+1}
 - Since $f(i_k) \leq f(j_k) \leq s(j_{k+1})$, this request is compatible with A

Greedy strategy is optimal

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- Suppose $m > k$
- We know $f(i_k) \leq f(j_k)$
- Greedy strategy stops when B is empty
 - Consider request j_{k+1}
 - Since $f(i_k) \leq f(j_k) \leq s(j_{k+1})$, this request is compatible with A
 - j_{k+1} must still be in B

Greedy strategy is optimal

- $A = \{i_1, i_2, \dots, i_k\}$
- $O = \{j_1, j_2, \dots, j_m\}$
- Suppose $m > k$
- We know $f(i_k) \leq f(j_k)$
- Greedy strategy stops when B is empty
 - Consider request j_{k+1}
 - Since $f(i_k) \leq f(j_k) \leq s(j_{k+1})$, this request is compatible with A
 - j_{k+1} must still be in B
 - B is not empty after choosing A , contradiction!

Implementation

- Initially, sort n bookings by finish time — $O(n \log n)$
 - Renumber bookings to reflect this sorted order

Implementation

- Initially, sort n bookings by finish time — $O(n \log n)$
 - Renumber bookings to reflect this sorted order
- Record start and finish times in array/dictionary
 - $S[i]$ is starting time of request i
 - $F[i]$ is finish time of request i

Implementation

- Initially, sort n bookings by finish time — $O(n \log n)$
 - Renumber bookings to reflect this sorted order
- Record start and finish times in array/dictionary
 - $S[i]$ is starting time of request i
 - $F[i]$ is finish time of request i
- Add first booking to A

Implementation

- Initially, sort n bookings by finish time — $O(n \log n)$
 - Renumber bookings to reflect this sorted order
- Record start and finish times in array/dictionary
 - $S[i]$ is starting time of request i
 - $F[i]$ is finish time of request i
- Add first booking to A
- In general, after adding booking j to A , Find the smallest r with $S[r] > F[j]$
 - Single scan, $O(n)$ overall

Summary

- A greedy algorithm makes a sequence of locally optimal choices to achieve a global optimum

Summary

- A greedy algorithm makes a sequence of locally optimal choices to achieve a global optimum
- The algorithm never goes back and reconsiders on a choice

Summary

- A greedy algorithm makes a sequence of locally optimal choices to achieve a global optimum
- The algorithm never goes back and reconsiders on a choice
- Drastically reduces space to search for solutions, but need to show prove that local strategy is globally optimal

Summary

- A greedy algorithm makes a sequence of locally optimal choices to achieve a global optimum
- The algorithm never goes back and reconsiders on a choice
- Drastically reduces space to search for solutions, but need to show prove that local strategy is globally optimal
- Interval scheduling — many “natural” greedy strategies

Summary

- A greedy algorithm makes a sequence of locally optimal choices to achieve a global optimum
- The algorithm never goes back and reconsiders on a choice
- Drastically reduces space to search for solutions, but need to show prove that local strategy is globally optimal
- Interval scheduling — many “natural” greedy strategies
- Most of them are wrong!

Summary

- A greedy algorithm makes a sequence of locally optimal choices to achieve a global optimum
- The algorithm never goes back and reconsiders on a choice
- Drastically reduces space to search for solutions, but need to show prove that local strategy is globally optimal
- Interval scheduling — many “natural” greedy strategies
- Most of them are wrong!
- Correct strategy needs a proof

Summary

- A greedy algorithm makes a sequence of locally optimal choices to achieve a global optimum
- The algorithm never goes back and reconsiders on a choice
- Drastically reduces space to search for solutions, but need to show prove that local strategy is globally optimal
- Interval scheduling — many “natural” greedy strategies
- Most of them are wrong!
- Correct strategy needs a proof
- One way is to show that greedy solution “stays ahead”, step by step, of any optimal solutions

Greedy Algorithms: Minimizing Lateness

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 7

Greedy Algorithms

- Make a sequence of local choices to achieve a global optimum
- Never go back and revise an earlier decision
- How to prove that local choices achieve global optimum?

Greedy Algorithms

- Make a sequence of local choices to achieve a global optimum
- Never go back and revise an earlier decision
- How to prove that local choices achieve global optimum?

Strategy 1

- Greedy solution and optimal may not be identical — interval scheduling
- Incrementally show that the greedy solution is at least as good as an optimal one
- The greedy algorithm “stays ahead” of the optimal

Greedy Algorithms

- Make a sequence of local choices to achieve a global optimum
- Never go back and revise an earlier decision
- How to prove that local choices achieve global optimum?

Strategy 1

- Greedy solution and optimal may not be identical — interval scheduling
- Incrementally show that the greedy solution is at least as good as an optimal one
- The greedy algorithm “stays ahead” of the optimal

Strategy 2

- Greedy solution and optimal have a common structure
- Transform the optimal solution to match the greedy one, preserving optimality

Minimizing lateness

- IIT Madras has a single 3D printer

Minimizing lateness

- IIT Madras has a single 3D printer
- A number of users need to use this printer

Minimizing lateness

- IIT Madras has a single 3D printer
- A number of users need to use this printer
- User i 's item takes time $T(i)$ to print

Minimizing lateness

- IIT Madras has a single 3D printer
- A number of users need to use this printer
- User i 's item takes time $T(i)$ to print
- User i has a deadline $D(i)$

Minimizing lateness

- IIT Madras has a single 3D printer
- A number of users need to use this printer
- User i 's item takes time $T(i)$ to print
- User i has a deadline $D(i)$
- Each user will get access to the printer, but may not finish before deadline

Minimizing lateness

- IIT Madras has a single 3D printer
 - A number of users need to use this printer
 - User i 's item takes time $T(i)$ to print
 - User i has a deadline $D(i)$
 - Each user will get access to the printer, but may not finish before deadline
- User i starts at time $S(i)$ and finishes at time $F(i) = S(i) + T(i)$

Minimizing lateness

- IIT Madras has a single 3D printer
 - A number of users need to use this printer
 - User i 's item takes time $T(i)$ to print
 - User i has a deadline $D(i)$
 - Each user will get access to the printer, but may not finish before deadline
- User i starts at time $S(i)$ and finishes at time $F(i) = S(i) + T(i)$
 - If $F(i) > D(i)$, user i is late by $L(i) = F(i) - D(i)$

Minimizing lateness

- IIT Madras has a single 3D printer
 - A number of users need to use this printer
 - User i 's item takes time $T(i)$ to print
 - User i has a deadline $D(i)$
 - Each user will get access to the printer, but may not finish before deadline
-
- User i starts at time $S(i)$ and finishes at time $F(i) = S(i) + T(i)$
 - If $F(i) > D(i)$, user i is late by $L(i) = F(i) - D(i)$
 - Maximum lateness: $\max_i L(i)$

Minimizing lateness

- IIT Madras has a single 3D printer
 - A number of users need to use this printer
 - User i 's item takes time $T(i)$ to print
 - User i has a deadline $D(i)$
 - Each user will get access to the printer, but may not finish before deadline
-
- User i starts at time $S(i)$ and finishes at time $F(i) = S(i) + T(i)$
 - If $F(i) > D(i)$, user i is late by $L(i) = F(i) - D(i)$
 - Maximum lateness: $\max_i L(i)$
 - **Goal** Minimize the maximum lateness

Greedy strategies

Strategy 1 Schedule requests in increasing order of length — $T(i)$

Greedy strategies

Strategy 1 Schedule requests in increasing order of length — $T(i)$

Counterexample

- Two jobs
 - $T(1) = 1, D(1) = 100$
 - $T(2) = 10, D(2) = 10$

Greedy strategies

Strategy 1 Schedule requests in increasing order of length — $T(i)$

Counterexample

- Two jobs
 - $T(1) = 1, D(1) = 100$
 - $T(2) = 10, D(2) = 10$

Strategy 2 Give priority to requests with smaller slack time — $D(i) - T(i)$

Greedy strategies

Strategy 1 Schedule requests in increasing order of length — $T(i)$

Counterexample

- Two jobs
 - $T(1) = 1, D(1) = 100$
 - $T(2) = 10, D(2) = 10$

Strategy 2 Give priority to requests with smaller slack time — $D(i) - T(i)$

Counterexample

- Two jobs
 - $T(1) = 1, D(1) = 2$
 - $T(2) = 10, D(2) = 10$

Greedy strategies

Strategy 1 Schedule requests in increasing order of length — $T(i)$

Counterexample

- Two jobs
 - $T(1) = 1, D(1) = 100$
 - $T(2) = 10, D(2) = 10$

Strategy 2 Give priority to requests with smaller slack time — $D(i) - T(i)$

Counterexample

- Two jobs
 - $T(1) = 1, D(1) = 2$
 - $T(2) = 10, D(2) = 10$

Strategy 3 Schedule requests in increasing order of deadlines — $D(i)$

Greedy strategies

Strategy 1 Schedule requests in increasing order of length — $T(i)$

Counterexample

- Two jobs
 - $T(1) = 1, D(1) = 100$
 - $T(2) = 10, D(2) = 10$

Strategy 2 Give priority to requests with smaller slack time — $D(i) - T(i)$

Counterexample

- Two jobs
 - $T(1) = 1, D(1) = 2$
 - $T(2) = 10, D(2) = 10$

Strategy 3 Schedule requests in increasing order of deadlines — $D(i)$

- This works, but how do you prove it is correct?

Correctness

- Renumber the jobs so that they are sorted by deadline
 - $D(1) \leq D(2) \leq \dots \leq D(n)$

Correctness

- Renumber the jobs so that they are sorted by deadline

- $D(1) \leq D(2) \leq \dots \leq D(n)$

- Final schedule is simply $1, 2, \dots, n$

- Job 1 starts at $S(1) = 0$, ends at $F(1) = T(1)$

- Job 2 starts at $S(2) = F(1)$, ends at $F(2) = S(2) + T(2)$

- ...

- Job n starts at $S(n) = F(n-1)$, ends at $F(n) = S(n) + T(n)$

Correctness

- Renumber the jobs so that they are sorted by deadline
 - $D(1) \leq D(2) \leq \dots \leq D(n)$
- Final schedule is simply $1, 2, \dots, n$
 - Job 1 starts at $S(1) = 0$, ends at $F(1) = T(1)$
 - Job 2 starts at $S(2) = F(1)$, ends at $F(2) = S(2) + T(2)$
 - ...
 - Job n starts at $S(n) = F(n-1)$, ends at $F(n) = S(n) + T(n)$
- Our schedule has no gaps
 - 3D printer is continuously in use from $S(1) = 0$ to $F(n)$
 - No idle time

Correctness

- Renumber the jobs so that they are sorted by deadline

- $D(1) \leq D(2) \leq \dots \leq D(n)$

- Final schedule is simply $1, 2, \dots, n$

- Job 1 starts at $S(1) = 0$, ends at $F(1) = T(1)$
 - Job 2 starts at $S(2) = F(1)$, ends at $F(2) = S(2) + T(2)$
 - ...
 - Job n starts at $S(n) = F(n-1)$, ends at $F(n) = S(n) + T(n)$

- Our schedule has no gaps

- 3D printer is continuously in use from $S(1) = 0$ to $F(n)$
 - No idle time

Claim

There is an optimal schedule with no idle time

Correctness

- Renumber the jobs so that they are sorted by deadline
 - $D(1) \leq D(2) \leq \dots \leq D(n)$
- Final schedule is simply $1, 2, \dots, n$
 - Job 1 starts at $S(1) = 0$, ends at $F(1) = T(1)$
 - Job 2 starts at $S(2) = F(1)$, ends at $F(2) = S(2) + T(2)$
 - ...
 - Job n starts at $S(n) = F(n-1)$, ends at $F(n) = S(n) + T(n)$

- Our schedule has no gaps
 - 3D printer is continuously in use from $S(1) = 0$ to $F(n)$
 - No idle time

Claim

There is an optimal schedule with no idle time

- Eliminate idle time by shifting jobs earlier
- Can only reduce lateness

Correctness

Exchange argument

- Let O be some other optimal schedule
- Transform O to be identical to greedy schedule $1, 2, \dots, n$

Correctness

Exchange argument

- Let O be some other optimal schedule
- Transform O to be identical to greedy schedule $1, 2, \dots, n$

Inversions

- O has an inversion if i appears before j but $D(j) < D(i)$
- Greedy schedule $1, 2, \dots, n$ has no inversions, by construction

Correctness

Exchange argument

- Let O be some other optimal schedule
- Transform O to be identical to greedy schedule $1, 2, \dots, n$

Claim

Any two schedules with no inversions and no idle time have same maximum lateness

Inversions

- O has an inversion if i appears before j but $D(j) < D(i)$
- Greedy schedule $1, 2, \dots, n$ has no inversions, by construction

Correctness

Exchange argument

- Let O be some other optimal schedule
- Transform O to be identical to greedy schedule $1, 2, \dots, n$

Inversions

- O has an inversion if i appears before j but $D(j) < D(i)$
- Greedy schedule $1, 2, \dots, n$ has no inversions, by construction

Claim

Any two schedules with no inversions and no idle time have same maximum lateness

- No inversions, idle time — only difference can be the order of requests with same deadline
- Reordering jobs with same deadline produces same lateness

Correctness

Exchange argument

- Let O be some other optimal schedule
- Transform O to be identical to greedy schedule $1, 2, \dots, n$

Inversions

- O has an inversion if i appears before j but $D(j) < D(i)$
- Greedy schedule $1, 2, \dots, n$ has no inversions, by construction

Claim

Any two schedules with no inversions and no idle time have same maximum lateness

- No inversions, idle time — only difference can be the order of requests with same deadline
- Reordering jobs with same deadline produces same lateness



Correctness

Exchange argument

- Let O be some other optimal schedule
- Transform O to be identical to greedy schedule $1, 2, \dots, n$

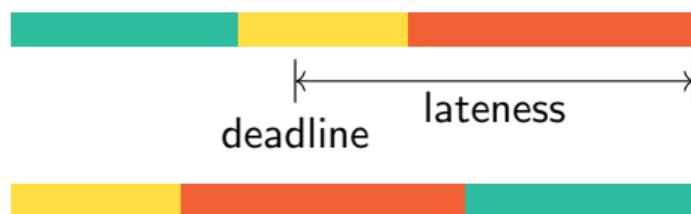
Inversions

- O has an inversion if i appears before j but $D(j) < D(i)$
- Greedy schedule $1, 2, \dots, n$ has no inversions, by construction

Claim

Any two schedules with no inversions and no idle time have same maximum lateness

- No inversions, idle time — only difference can be the order of requests with same deadline
- Reordering jobs with same deadline produces same lateness



Correctness

Claim

There is an optimal schedule with no inversions and no idle time

Correctness

Claim

There is an optimal schedule with no inversions and no idle time

Let O be an optimal schedule with no idle time

Correctness

Claim

There is an optimal schedule with no inversions and no idle time

Let O be an optimal schedule with no idle time

- A If O has an inversion, there are jobs i, j such that j is scheduled immediately after i and $D(j) < D(i)$

- Find the first point where deadline decreases

Correctness

Claim

There is an optimal schedule with no inversions and no idle time

Let O be an optimal schedule with no idle time

- A If O has an inversion, there are jobs i, j such that j is scheduled immediately after i and $D(j) < D(i)$

- Find the first point where deadline decreases

- B Swap i and j to get one less inversion
 - Obvious

Correctness

Claim

There is an optimal schedule with no inversions and no idle time

Let O be an optimal schedule with no idle time

- A If O has an inversion, there are jobs i, j such that j is scheduled immediately after i and $D(j) < D(i)$

- Find the first point where deadline decreases

- B Swap i and j to get one less inversion
- Obvious

- C Swapping i and j does not increase lateness of O

- Not so obvious

Correctness

Claim

There is an optimal schedule with no inversions and no idle time

Let O be an optimal schedule with no idle time

- A If O has an inversion, there are jobs i, j such that j is scheduled immediately after i and $D(j) < D(i)$

- Find the first point where deadline decreases

- B Swap i and j to get one less inversion
- Obvious

- C Swapping i and j does not increase lateness of O

- Not so obvious



Correctness

Claim

There is an optimal schedule with no inversions and no idle time

Let O be an optimal schedule with no idle time

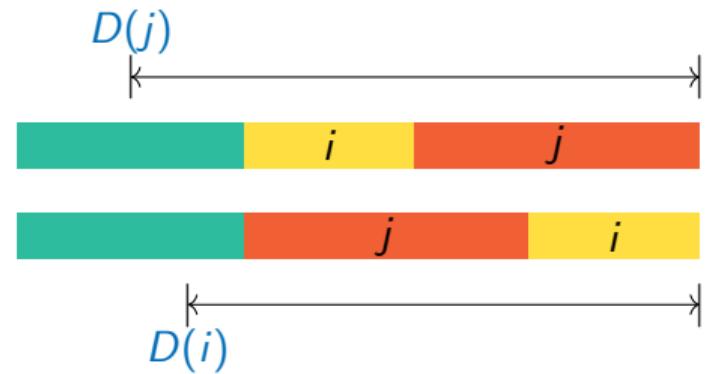
- A If O has an inversion, there are jobs i, j such that j is scheduled immediately after i and $D(j) < D(i)$

- Find the first point where deadline decreases

- B Swap i and j to get one less inversion
- Obvious

- C Swapping i and j does not increase lateness of O

- Not so obvious
- Recall that $D(j) < D(i)$



Correctness

Claim

There is an optimal schedule with no inversions and no idle time

Let O be an optimal schedule with no idle time

- A If O has an inversion, there are jobs i, j such that j is scheduled immediately after i and $D(j) < D(i)$
- B Swap i and j to get one less inversion
- C Swapping i and j does not increase lateness of O

Correctness

Claim

There is an optimal schedule with no inversions and no idle time

Let O be an optimal schedule with no idle time

- From C we can remove each adjacent inversion without increasing lateness

- A If O has an inversion, there are jobs i, j such that j is scheduled immediately after i and $D(j) < D(i)$
- B Swap i and j to get one less inversion
- C Swapping i and j does not increase lateness of O

Correctness

Claim

There is an optimal schedule with no inversions and no idle time

Let O be an optimal schedule with no idle time

- A If O has an inversion, there are jobs i, j such that j is scheduled immediately after i and $D(j) < D(i)$
- B Swap i and j to get one less inversion
- C Swapping i and j does not increase lateness of O

- From C we can remove each adjacent inversion without increasing lateness
- At most $n(n - 1)/2$ inversions in O to start with

Correctness

Claim

There is an optimal schedule with no inversions and no idle time

Let O be an optimal schedule with no idle time

- A If O has an inversion, there are jobs i, j such that j is scheduled immediately after i and $D(j) < D(i)$
- B Swap i and j to get one less inversion
- C Swapping i and j does not increase lateness of O

- From C we can remove each adjacent inversion without increasing lateness
- At most $n(n - 1)/2$ inversions in O to start with
- Repeatedly remove adjacent inversions to get an optimal schedule with no inversions, no idle time

Summary

- Schedule requests with start times $T(i)$, deadlines $D(i)$, to minimize maximum lateness

Summary

- Schedule requests with start times $T(i)$, deadlines $D(i)$, to minimize maximum lateness
- Simple greedy algorithm with complexity $O(n \log n)$
 - Sort the requests by $D(i)$ — $O(n \log n)$
 - Read off schedule in sorted order — $O(n)$

Summary

- Schedule requests with start times $T(i)$, deadlines $D(i)$, to minimize maximum lateness
- Simple greedy algorithm with complexity $O(n \log n)$
 - Sort the requests by $D(i)$ — $O(n \log n)$
 - Read off schedule in sorted order — $O(n)$
- Correctness follows from an “exchange argument”
 - Consider any optimal solution O
 - Transform it, step by step, to be equal to the greedy solution

Greedy Algorithms: Huffman Coding

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 7

Efficient communication

- Send messages in English or Hindi or Tamil or ...

Efficient communication

- Send messages in English or Hindi or Tamil or ...
- Each language has its own alphabet

Efficient communication

- Send messages in English or Hindi or Tamil or ...
- Each language has its own alphabet
- Digital communication uses $\{0, 1\}$

Efficient communication

- Send messages in English or Hindi or Tamil or ...
- Each language has its own alphabet
- Digital communication uses $\{0, 1\}$
- Encode $\{a, b, \dots, z\}$ using $\{0, 1\}$
 - Use binary strings of length 5
 - 26 letters, $2^4 < 26 \leq 2^5$

Efficient communication

- Send messages in English or Hindi or Tamil or ...
- Each language has its own alphabet
- Digital communication uses $\{0, 1\}$
- Encode $\{a, b, \dots, z\}$ using $\{0, 1\}$
 - Use binary strings of length 5
 - 26 letters, $2^4 < 26 \leq 2^5$
- Can we do better?
 - Use shorter strings for more frequent letters
 - Optimize data transfer

Efficient communication

- Send messages in English or Hindi or Tamil or ...
- Each language has its own alphabet
- Digital communication uses $\{0, 1\}$
- Encode $\{a, b, \dots, z\}$ using $\{0, 1\}$
 - Use binary strings of length 5
 - 26 letters, $2^4 < 26 \leq 2^5$
- Can we do better?
 - Use shorter strings for more frequent letters
 - Optimize data transfer

Variable length encoding
Morse Code

Efficient communication

- Send messages in English or Hindi or Tamil or ...
- Each language has its own alphabet
- Digital communication uses $\{0, 1\}$
- Encode $\{a, b, \dots, z\}$ using $\{0, 1\}$
 - Use binary strings of length 5
 - 26 letters, $2^4 < 26 \leq 2^5$
- Can we do better?
 - Use shorter strings for more frequent letters
 - Optimize data transfer

Variable length encoding
Morse Code

- Encode letters using dots (0) and dashes (1)

Efficient communication

- Send messages in English or Hindi or Tamil or ...
- Each language has its own alphabet
- Digital communication uses $\{0, 1\}$
- Encode $\{a, b, \dots, z\}$ using $\{0, 1\}$
 - Use binary strings of length 5
 - 26 letters, $2^4 < 26 \leq 2^5$
- Can we do better?
 - Use shorter strings for more frequent letters
 - Optimize data transfer

Variable length encoding Morse Code

- Encode letters using dots (0) and dashes (1)
- Encoding of e is 0, t is 1, a is 01

Efficient communication

- Send messages in English or Hindi or Tamil or ...
- Each language has its own alphabet
- Digital communication uses $\{0, 1\}$
- Encode $\{a, b, \dots, z\}$ using $\{0, 1\}$
 - Use binary strings of length 5
 - 26 letters, $2^4 < 26 \leq 2^5$
- Can we do better?
 - Use shorter strings for more frequent letters
 - Optimize data transfer

Variable length encoding Morse Code

- Encode letters using dots (0) and dashes (1)
- Encoding of e is 0, t is 1, a is 01
- Decode 0101

Efficient communication

- Send messages in English or Hindi or Tamil or ...
- Each language has its own alphabet
- Digital communication uses $\{0, 1\}$
- Encode $\{a, b, \dots, z\}$ using $\{0, 1\}$
 - Use binary strings of length 5
 - 26 letters, $2^4 < 26 \leq 2^5$
- Can we do better?
 - Use shorter strings for more frequent letters
 - Optimize data transfer

Variable length encoding Morse Code

- Encode letters using dots (0) and dashes (1)
- Encoding of e is 0, t is 1, a is 01
- Decode 0101
 - aa

Efficient communication

- Send messages in English or Hindi or Tamil or ...
- Each language has its own alphabet
- Digital communication uses $\{0, 1\}$
- Encode $\{a, b, \dots, z\}$ using $\{0, 1\}$
 - Use binary strings of length 5
 - 26 letters, $2^4 < 26 \leq 2^5$
- Can we do better?
 - Use shorter strings for more frequent letters
 - Optimize data transfer

Variable length encoding Morse Code

- Encode letters using dots (0) and dashes (1)
- Encoding of e is 0, t is 1, a is 01
- Decode 0101
 - aa , etet, aet, eta?

Efficient communication

- Send messages in English or Hindi or Tamil or ...
- Each language has its own alphabet
- Digital communication uses $\{0, 1\}$
- Encode $\{a, b, \dots, z\}$ using $\{0, 1\}$
 - Use binary strings of length 5
 - 26 letters, $2^4 < 26 \leq 2^5$
- Can we do better?
 - Use shorter strings for more frequent letters
 - Optimize data transfer

Variable length encoding Morse Code

- Encode letters using dots (0) and dashes (1)
- Encoding of e is 0, t is 1, a is 01
- Decode 0101
 - aa, etet, aet, eta?
- Use pauses between letters
 - Like adding a third symbol for encoding

Prefix codes

- Encoding of x , $E(x)$, is not a **prefix** of $E(y)$ for any x, y
 - In Morse Code, $E(e) = 0$ is a prefix of $E(a) = 01$

Prefix codes

- Encoding of x , $E(x)$, is not a prefix of $E(y)$ for any x, y
 - In Morse Code, $E(e) = 0$ is a prefix of $E(a) = 01$
- Example of a prefix code

x	a	b	c	d	e
$E(x)$	11	01	001	10	000

Prefix codes

- Encoding of x , $E(x)$, is not a prefix of $E(y)$ for any x, y
 - In Morse Code, $E(e) = 0$ is a prefix of $E(a) = 01$
- Example of a prefix code

x	a	b	c	d	e
$E(x)$	11	01	001	10	000

- Decode

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

Prefix codes

- Encoding of x , $E(x)$, is not a prefix of $E(y)$ for any x, y
 - In Morse Code, $E(e) = 0$ is a prefix of $E(a) = 01$
- Example of a prefix code

x	a	b	c	d	e
$E(x)$	11	01	001	10	000

- Decode

0 0 1 | 0 0 0 0 1 1 1 0 1
c

Prefix codes

- Encoding of x , $E(x)$, is not a prefix of $E(y)$ for any x, y
 - In Morse Code, $E(e) = 0$ is a prefix of $E(a) = 01$
- Example of a prefix code

x	a	b	c	d	e
$E(x)$	11	01	001	10	000

- Decode

0 0 1 | 0 0 0 | 0 0 1 1 0 1
 c e

Prefix codes

- Encoding of x , $E(x)$, is not a prefix of $E(y)$ for any x, y
 - In Morse Code, $E(e) = 0$ is a prefix of $E(a) = 01$
- Example of a prefix code

x	a	b	c	d	e
$E(x)$	11	01	001	10	000

- Decode

0 0 1 | 0 0 | 0 0 1 | 1 1 0 1
 c e c

Prefix codes

- Encoding of x , $E(x)$, is not a prefix of $E(y)$ for any x, y
 - In Morse Code, $E(e) = 0$ is a prefix of $E(a) = 01$
- Example of a prefix code

x	a	b	c	d	e
$E(x)$	11	01	001	10	000

- Decode

0 0 1 | 0 0 | 0 0 1 | 1 1 | 0 1
 c e c a

Prefix codes

- Encoding of x , $E(x)$, is not a prefix of $E(y)$ for any x, y
 - In Morse Code, $E(e) = 0$ is a prefix of $E(a) = 01$
- Example of a prefix code

x	a	b	c	d	e
$E(x)$	11	01	001	10	000

- Decode

0 0 1 | 0 0 | 0 0 1 | 1 1 | 0 1 |
 c e c a b

Optimal prefix codes

- Measure frequency $f(x)$ of each letter
 - Fraction of occurrences of x over large text corpus
 - Number of times x appears divided by total number of letters

Optimal prefix codes

- Measure frequency $f(x)$ of each letter
 - Fraction of occurrences of x over large text corpus
 - Number of times x appears divided by total number of letters
- $A = \{x_1, x_2, \dots, x_m\}$
 - $f(x_1) + f(x_2) + \dots + f(x_m) = 1$
 - $f(x)$ is “probability” that next letter is x corpus

Optimal prefix codes

- Measure frequency $f(x)$ of each letter
 - Fraction of occurrences of x over large text corpus
 - Number of times x appears divided by total number of letters
- $A = \{x_1, x_2, \dots, x_m\}$
 - $f(x_1) + f(x_2) + \dots + f(x_m) = 1$
 - $f(x)$ is “probability” that next letter is x corpus
- Message M to be transmitted has n symbols
 - Each letter x occurs $n \cdot f(x)$ times

Optimal prefix codes

- Measure frequency $f(x)$ of each letter
 - Fraction of occurrences of x over large text corpus
 - Number of times x appears divided by total number of letters
- $A = \{x_1, x_2, \dots, x_m\}$
 - $f(x_1) + f(x_2) + \dots + f(x_m) = 1$
 - $f(x)$ is “probability” that next letter is x corpus
- Message M to be transmitted has n symbols
 - Each letter x occurs $n \cdot f(x)$ times

Optimal prefix codes

- Measure frequency $f(x)$ of each letter
 - Fraction of occurrences of x over large text corpus
 - Number of times x appears divided by total number of letters
- $A = \{x_1, x_2, \dots, x_m\}$
 - $f(x_1) + f(x_2) + \dots + f(x_m) = 1$
 - $f(x)$ is “probability” that next letter is x corpus
- Message M to be transmitted has n symbols
 - Each letter x occurs $n \cdot f(x)$ times
- Each x is encoded as $E(x)$ with length $|E(x)|$
- Total message length is
$$\sum_{x \in A} n \cdot f(x) \cdot |E(x)|$$

Optimal prefix codes

- Measure frequency $f(x)$ of each letter
 - Fraction of occurrences of x over large text corpus
 - Number of times x appears divided by total number of letters
- $A = \{x_1, x_2, \dots, x_m\}$
 - $f(x_1) + f(x_2) + \dots + f(x_m) = 1$
 - $f(x)$ is “probability” that next letter is x corpus
- Message M to be transmitted has n symbols
 - Each letter x occurs $n \cdot f(x)$ times
- Each x is encoded as $E(x)$ with length $|E(x)|$
- Total message length is
$$\sum_{x \in A} n \cdot f(x) \cdot |E(x)|$$
- Average number of bits per letter in encoding
$$\sum_{x \in A} f(x) \cdot |E(x)|$$

Optimal prefix codes

- Suppose we have the following frequencies for our earlier example

x	a	b	c	d	e
$E(x)$	11	01	001	10	000
$f(x)$	0.32	0.25	0.20	0.18	0.05

Optimal prefix codes

- Suppose we have the following frequencies for our earlier example

<i>x</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
$E(x)$	11	01	001	10	000
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Average number of bits per letter is
2.25

- $$(0.32 \cdot 2) + (0.25 \cdot 2) + (0.20 \cdot 3) + (0.18 \cdot 2) + (0.05 \cdot 3)$$

Optimal prefix codes

- Suppose we have the following frequencies for our earlier example

x	a	b	c	d	e
$E(x)$	11	01	001	10	000
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Average number of bits per letter is
2.25
 - $(0.32 \cdot 2) + (0.25 \cdot 2) + (0.20 \cdot 3)$
 $+(0.18 \cdot 2) + (0.05 \cdot 3)$
- Fixed length encoding would require 3 bits per letter — $2^2 < 5 \leq 2^3$
 - 25% saving using variable length code

Optimal prefix codes

- Suppose we have the following frequencies for our earlier example

x	a	b	c	d	e
$E(x)$	11	01	001	10	000
$f(x)$	0.32	0.25	0.20	0.18	0.05

- A better encoding

x	a	b	c	d	e
$E(x)$	11	10	01	001	000
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Average number of bits per letter is 2.25

- $$(0.32 \cdot 2) + (0.25 \cdot 2) + (0.20 \cdot 3) + (0.18 \cdot 2) + (0.05 \cdot 3)$$

- Fixed length encoding would require 3 bits per letter — $2^2 < 5 \leq 2^3$
 - 25% saving using variable length code

Optimal prefix codes

- Suppose we have the following frequencies for our earlier example

x	a	b	c	d	e
$E(x)$	11	01	001	10	000
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Average number of bits per letter is 2.25

- $$(0.32 \cdot 2) + (0.25 \cdot 2) + (0.20 \cdot 3) + (0.18 \cdot 2) + (0.05 \cdot 3)$$

- Fixed length encoding would require 3 bits per letter — $2^2 < 5 \leq 2^3$
 - 25% saving using variable length code

- A better encoding

x	a	b	c	d	e
$E(x)$	11	10	01	001	000
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Average number of bits per letter is 2.23

- $$(0.32 \cdot 2) + (0.25 \cdot 2) + (0.20 \cdot 2) + (0.18 \cdot 3) + (0.05 \cdot 3)$$

Optimal prefix codes

- Suppose we have the following frequencies for our earlier example

x	a	b	c	d	e
$E(x)$	11	01	001	10	000
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Average number of bits per letter is 2.25

- $$(0.32 \cdot 2) + (0.25 \cdot 2) + (0.20 \cdot 3) + (0.18 \cdot 2) + (0.05 \cdot 3)$$

- Fixed length encoding would require 3 bits per letter — $2^2 < 5 \leq 2^3$
 - 25% saving using variable length code

- A better encoding

x	a	b	c	d	e
$E(x)$	11	10	01	001	000
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Average number of bits per letter is 2.23

- $$(0.32 \cdot 2) + (0.25 \cdot 2) + (0.20 \cdot 2) + (0.18 \cdot 3) + (0.05 \cdot 3)$$

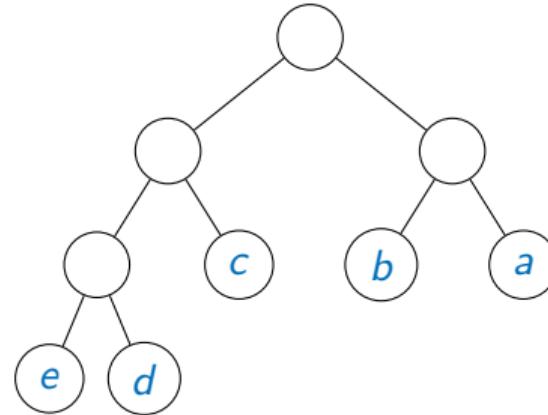
- Given a set of letters A and frequencies $f(x)$ for each x , produce the most efficient prefix code possible

- Minimize $ABL(A)$ — Average Bits per Letter

Codes as trees

- Encoding can be viewed as a binary tree

x	a	b	c	d	e
$E(x)$	11	10	01	001	000

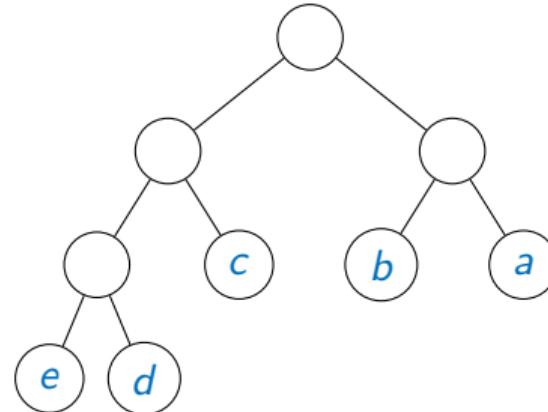


Codes as trees

- Encoding can be viewed as a binary tree

x	a	b	c	d	e
$E(x)$	11	10	01	001	000

- Letters are at the leaves

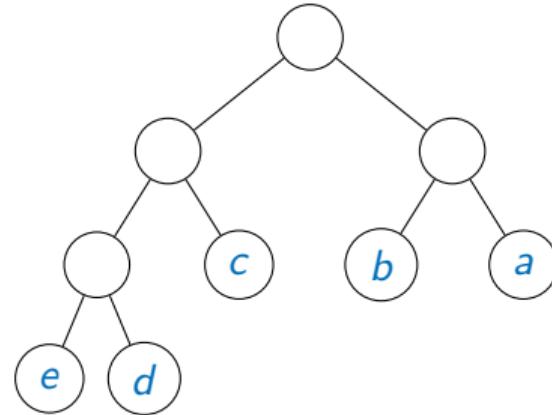


Codes as trees

- Encoding can be viewed as a binary tree

x	a	b	c	d	e
$E(x)$	11	10	01	001	000

- Letters are at the leaves
- Path to leaf describes encoding — 0 is left, 1 is right

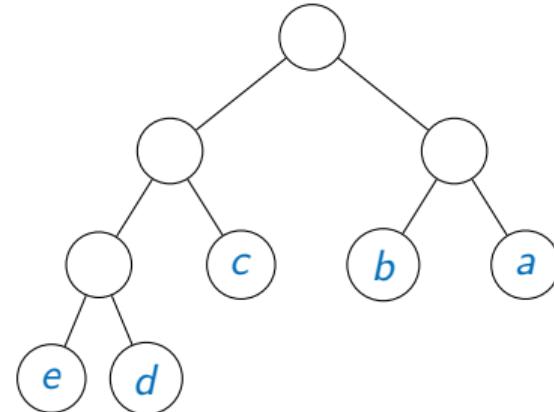


Codes as trees

- Encoding can be viewed as a binary tree

x	a	b	c	d	e
$E(x)$	11	10	01	001	000

- Letters are at the leaves
- Path to leaf describes encoding — 0 is left, 1 is right
- Prefix code — no internal nodes encode letters

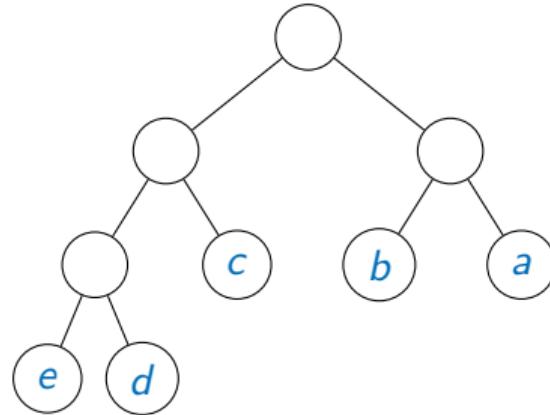


Codes as trees

Claim 1

Any optimal prefix code produces a full tree

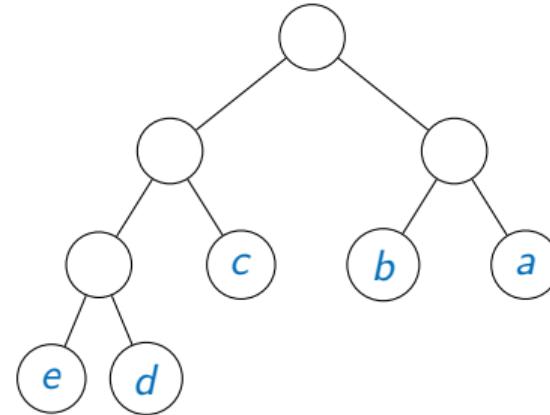
- **Full tree** Each node has 0 or 2 children



Claim 1

Any optimal prefix code produces a full tree

- **Full tree** Each node has 0 or 2 children
- For a node with only one child, “promote” child to get a shorter tree



Codes as trees

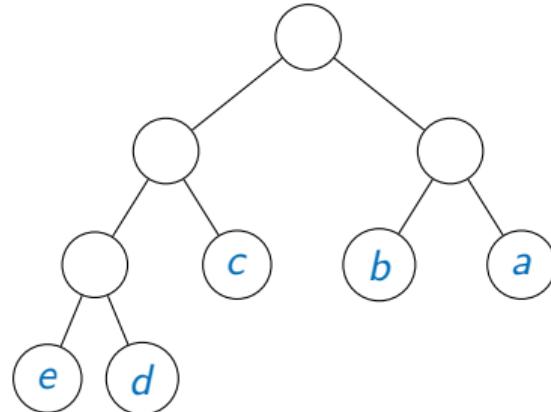
Claim 1

Any optimal prefix code produces a full tree

- **Full tree** Each node has 0 or 2 children
- For a node with only one child, “promote” child to get a shorter tree

Claim 2

In an optimal tree, if leaf labelled x is at smaller depth (higher) than leaf labelled y ,
 $f(x) \geq f(y)$



Codes as trees

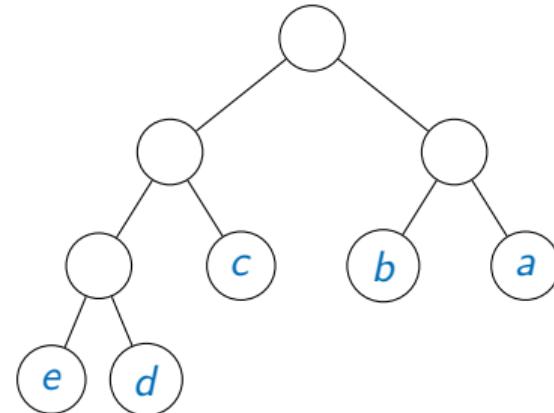
Claim 1

Any optimal prefix code produces a full tree

- **Full tree** Each node has 0 or 2 children
- For a node with only one child, “promote” child to get a shorter tree

Claim 2

In an optimal tree, if leaf labelled x is at smaller depth (higher) than leaf labelled y ,
 $f(x) \geq f(y)$

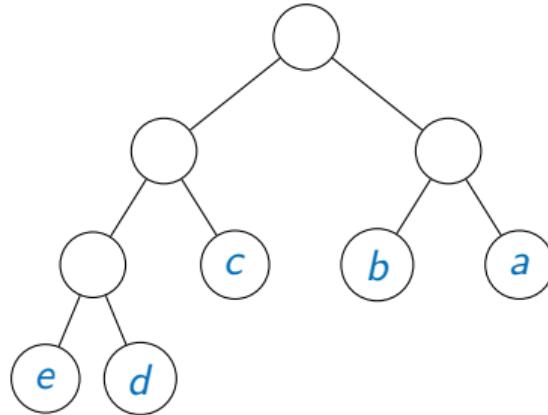


- If $f(y) > f(x)$, exchange labels, improve tree

Codes as trees

Claim 3

In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf ✕

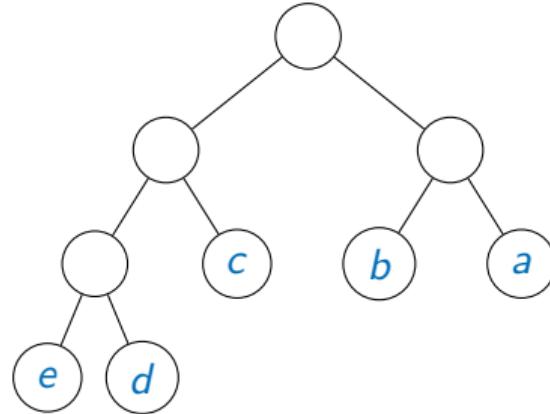


Codes as trees

Claim 3

In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf ✗

- If not, the sibling of this leaf has children

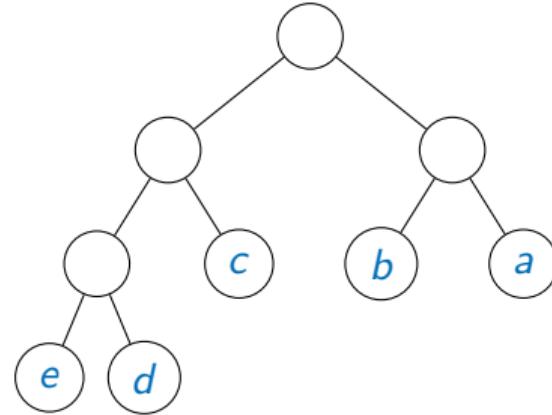


Codes as trees

Claim 3

In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf \times

- If not, the sibling of this leaf has children
- There is a leaf at lower depth

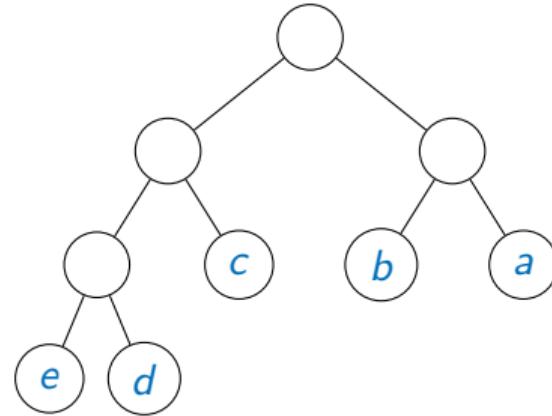


Codes as trees

Claim 3

In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf ✗

- If not, the sibling of this leaf has children
- There is a leaf at lower depth
- The leaf we started with is not a maximum depth!



A recursive algorithm

Claim 1

Any optimal prefix code produces a full tree

Claim 2

In an optimal tree, if leaf labelled x is at smaller depth (higher) than leaf labelled y ,
 $f(x) \geq f(y)$

Claim 3

In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf x

A recursive algorithm

- From Claim 3, leaves at maximum depth occur in pairs

Claim 1

Any optimal prefix code produces a full tree

Claim 2

In an optimal tree, if leaf labelled x is at smaller depth (higher) than leaf labelled y ,
 $f(x) \geq f(y)$

Claim 3

In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf x

A recursive algorithm

- From Claim 3, leaves at maximum depth occur in pairs
- From Claim 2, these must have lowest frequencies

Claim 1

Any optimal prefix code produces a full tree

Claim 2

In an optimal tree, if leaf labelled x is at smaller depth (higher) than leaf labelled y ,
 $f(x) \geq f(y)$

Claim 3

In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf x

A recursive algorithm

- From Claim 3, leaves at maximum depth occur in pairs
- From Claim 2, these must have lowest frequencies
- Pick x, y with smallest $f(x), f(y)$

Claim 1

Any optimal prefix code produces a full tree

Claim 2

In an optimal tree, if leaf labelled x is at smaller depth (higher) than leaf labelled y ,
 $f(x) \geq f(y)$

Claim 3

In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf x

A recursive algorithm

- From Claim 3, leaves at maximum depth occur in pairs
- From Claim 2, these must have lowest frequencies
- Pick x, y with smallest $f(x), f(y)$
- Assign x, y to lowest pair of leaves (left/right does not matter)

Claim 1

Any optimal prefix code produces a full tree

Claim 2

In an optimal tree, if leaf labelled x is at smaller depth (higher) than leaf labelled y ,
 $f(x) \geq f(y)$

Claim 3

In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf x

A recursive algorithm

- From Claim 3, leaves at maximum depth occur in pairs
- From Claim 2, these must have lowest frequencies
- Pick x, y with smallest $f(x), f(y)$
- Assign x, y to lowest pair of leaves (left/right does not matter)
- “Combine” x, y into new letter xy with $f(xy) = f(x) + f(y)$

Claim 1

Any optimal prefix code produces a full tree

Claim 2

In an optimal tree, if leaf labelled x is at smaller depth (higher) than leaf labelled y ,
 $f(x) \geq f(y)$

Claim 3

In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf x

A recursive algorithm

- From Claim 3, leaves at maximum depth occur in pairs
- From Claim 2, these must have lowest frequencies
- Pick x, y with smallest $f(x), f(y)$
- Assign x, y to lowest pair of leaves (left/right does not matter)
- “Combine” x, y into new letter xy with $f(xy) = f(x) + f(y)$
- Update alphabet
$$A' = (A \setminus \{x, y\}) \cup \{xy\}$$

Claim 1

Any optimal prefix code produces a full tree

Claim 2

In an optimal tree, if leaf labelled x is at smaller depth (higher) than leaf labelled y ,
 $f(x) \geq f(y)$

Claim 3

In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf x

A recursive algorithm

- From Claim 3, leaves at maximum depth occur in pairs
- From Claim 2, these must have lowest frequencies
- Pick x, y with smallest $f(x), f(y)$
- Assign x, y to lowest pair of leaves (left/right does not matter)
- “Combine” x, y into new letter xy with $f(xy) = f(x) + f(y)$
- Update alphabet
$$A' = (A \setminus \{x, y\}) \cup \{xy\}$$
- Recursively find an optimal tree T' for A'

A recursive algorithm

- From Claim 3, leaves at maximum depth occur in pairs
- From Claim 2, these must have lowest frequencies
- Pick x, y with smallest $f(x), f(y)$
- Assign x, y to lowest pair of leaves (left/right does not matter)
- “Combine” x, y into new letter xy with $f(xy) = f(x) + f(y)$
- Update alphabet
$$A' = (A \setminus \{x, y\}) \cup \{xy\}$$
- Recursively find an optimal tree T' for A'
- T' will have a leaf labelled xy

A recursive algorithm

- From Claim 3, leaves at maximum depth occur in pairs
- From Claim 2, these must have lowest frequencies
- Pick x, y with smallest $f(x), f(y)$
- Assign x, y to lowest pair of leaves (left/right does not matter)
- “Combine” x, y into new letter xy with $f(xy) = f(x) + f(y)$
- Update alphabet
$$A' = (A \setminus \{x, y\}) \cup \{xy\}$$
- Recursively find an optimal tree T' for A'
- T' will have a leaf labelled xy
- Replace this leaf by internal node with two children labelled x, y

A recursive algorithm

- From Claim 3, leaves at maximum depth occur in pairs
- From Claim 2, these must have lowest frequencies
- Pick x, y with smallest $f(x), f(y)$
- Assign x, y to lowest pair of leaves (left/right does not matter)
- “Combine” x, y into new letter xy with $f(xy) = f(x) + f(y)$
- Update alphabet
$$A' = (A \setminus \{x, y\}) \cup \{xy\}$$
- Recursively find an optimal tree T' for A'
- T' will have a leaf labelled xy
- Replace this leaf by internal node with two children labelled x, y
- **Huffman coding** — David E Huffman

Huffman's algorithm

x	a	b	c	d	e
$f(x)$	0.32	0.25	0.20	0.18	0.05

Huffman's algorithm

x	a	b	c	d	e
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Combine d, e as de

x	a	b	c	de
$f(x)$	0.32	0.25	0.20	0.23

Huffman's algorithm

x	a	b	c	d	e
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Combine d, e as de

x	a	b	c	de
$f(x)$	0.32	0.25	0.20	0.23

- Combine c, de as cde

x	a	b	cde
$f(x)$	0.32	0.25	0.43

Huffman's algorithm

x	a	b	c	d	e
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Combine d, e as de

x	a	b	c	de
$f(x)$	0.32	0.25	0.20	0.23

- Combine c, de as cde

x	a	b	cde
$f(x)$	0.32	0.25	0.43

- Combine a, b as ab

x	ab	cde
$f(x)$	0.57	0.43

Huffman's algorithm

x	a	b	c	d	e
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Combine d, e as de

x	a	b	c	de
$f(x)$	0.32	0.25	0.20	0.23

- Combine c, de as cde

x	a	b	cde
$f(x)$	0.32	0.25	0.43

- Combine a, b as ab

x	ab	cde
$f(x)$	0.57	0.43

- Two letters, base case, build a tree

Huffman's algorithm

x	a	b	c	d	e
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Combine d, e as de

x	a	b	c	de
$f(x)$	0.32	0.25	0.20	0.23

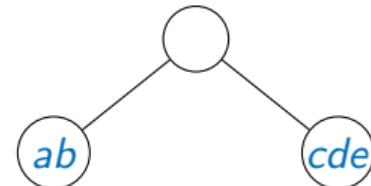
- Combine c, de as cde

x	a	b	cde
$f(x)$	0.32	0.25	0.43

- Combine a, b as ab

x	ab	cde
$f(x)$	0.57	0.43

- Two letters, base case, build a tree



Huffman's algorithm

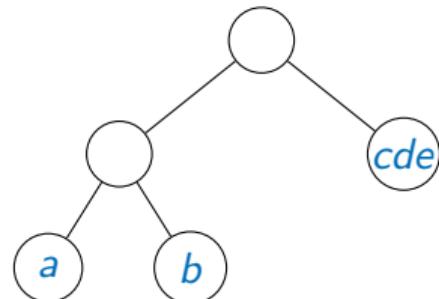
x	a	b	c	d	e
$f(x)$	0.32	0.25	0.20	0.18	0.05

x	a	b	c	de
$f(x)$	0.32	0.25	0.20	0.23

x	a	b	cde
$f(x)$	0.32	0.25	0.43

x	ab	cde
$f(x)$	0.57	0.43

- Combine d, e as de
- Combine c, de as cde
- Combine a, b as ab
- Two letters, base case, build a tree
- Repeatedly split compound leaves



Huffman's algorithm

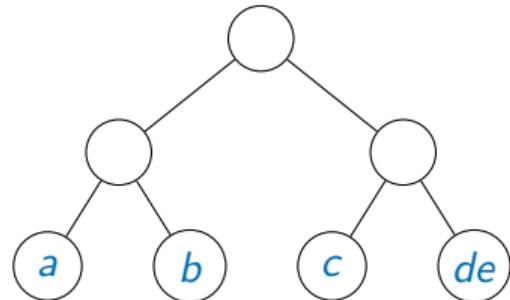
x	a	b	c	d	e
$f(x)$	0.32	0.25	0.20	0.18	0.05

x	a	b	c	de
$f(x)$	0.32	0.25	0.20	0.23

x	a	b	cde
$f(x)$	0.32	0.25	0.43

x	ab	cde
$f(x)$	0.57	0.43

- Combine d, e as de
- Combine c, de as cde
- Combine a, b as ab
- Two letters, base case, build a tree
- Repeatedly split compound leaves



Huffman's algorithm

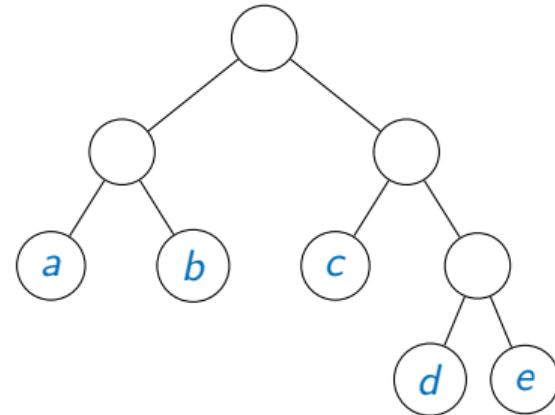
x	a	b	c	d	e
$f(x)$	0.32	0.25	0.20	0.18	0.05

x	a	b	c	de
$f(x)$	0.32	0.25	0.20	0.23

x	a	b	cde
$f(x)$	0.32	0.25	0.43

x	ab	cde
$f(x)$	0.57	0.43

- Combine d, e as de
- Combine c, de as cde
- Combine a, b as ab
- Two letters, base case, build a tree
- Repeatedly split compound leaves



Optimality of Huffman's algorithm

By induction on the size of alphabet A

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

Claim

$$ABL(T) - ABL(T') = f(xy)$$

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

Claim

$$ABL(T) - ABL(T') = f(xy)$$

- From T' to T , only change to ABL is due to xy, x, y

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

Claim

$$ABL(T) - ABL(T') = f(xy)$$

- From T' to T , only change to ABL is due to xy, x, y
- Subtract $depth(xy)f(xy)$, add $depth(x)f(x) + depth(y)f(y)$

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

Claim

$$ABL(T) - ABL(T') = f(xy)$$

- From T' to T , only change to ABL is due to xy, x, y
- Subtract $depth(xy)f(xy)$, add $depth(x)f(x) + depth(y)f(y)$
- $f(xy) = f(x) + f(y)$,
 $depth(x) = depth(y) = 1 + depth(xy)$

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

Claim

$$ABL(T) - ABL(T') = f(xy)$$

- From T' to T , only change to ABL is due to xy, x, y
- Subtract $depth(xy)f(xy)$, add $depth(x)f(x) + depth(y)f(y)$
- $f(xy) = f(x) + f(y)$,
 $depth(x) = depth(y) = 1 + depth(xy)$
- Net increase is $f(x) + f(y)$, which is $f(xy)$

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

Claim

$$ABL(T) - ABL(T') = f(xy)$$

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

- Suppose there is an optimal tree S for A with $ABL(S) < ABL(T)$

Claim

$$ABL(T) - ABL(T') = f(xy)$$

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

Claim

$$ABL(T) - ABL(T') = f(xy)$$

- Suppose there is an optimal tree S for A with $ABL(S) < ABL(T)$
- Shuffle the labels in S so that lowest frequency x, y are siblings

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

- Suppose there is an optimal tree S for A with $ABL(S) < ABL(T)$
- Shuffle the labels in S so that lowest frequency x, y are siblings
- Merge x, y as xy , contract S to S'

Claim

$$ABL(T) - ABL(T') = f(xy)$$

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

- Suppose there is an optimal tree S for A with $ABL(S) < ABL(T)$
- Shuffle the labels in S so that lowest frequency x, y are siblings
- Merge x, y as xy , contract S to S'
- S' is over same A' as T' , T' is optimal for A' , so $ABL(T') \leq ABL(S')$

Claim

$$ABL(T) - ABL(T') = f(xy)$$

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

- Suppose there is an optimal tree S for A with $ABL(S) < ABL(T)$
- Shuffle the labels in S so that lowest frequency x, y are siblings
- Merge x, y as xy , contract S to S'
- S' is over same A' as T' , T' is optimal for A' , so $ABL(T') \leq ABL(S')$
- $ABL(S) - ABL(S') = ABL(T) - ABL(T') = f(xy)$

Claim

$$ABL(T) - ABL(T') = f(xy)$$

Optimality of Huffman's algorithm

By induction on the size of alphabet A

- Base case, $|A| = 2$
 - Single letter code $\{0, 1\}$ is optimal
- Assume optimality for $|A| = k-1$
- For $|A| = k$
 - Combine lowest frequency x, y as xy
 - Construct tree T' for A'
 - $ABL(T')$ is optimal by induction
 - Expand leaf xy to get T

Claim

$$ABL(T) - ABL(T') = f(xy)$$

- Suppose there is an optimal tree S for A with $ABL(S) < ABL(T)$
- Shuffle the labels in S so that lowest frequency x, y are siblings
- Merge x, y as xy , contract S to S'
- S' is over same A' as T' , T' is optimal for A' , so $ABL(T') \leq ABL(S')$
- $ABL(S) - ABL(S') = ABL(T) - ABL(T') = f(xy)$
- Hence $ABL(T) \leq ABL(S)$, a contradiction

Implementation, complexity

- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency

Implementation, complexity

- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency
- Store frequencies in an array
 - Linear scan to find minimum values
 - $|A| = k$, number of recursive calls is $k-1$
 - Complexity is $O(k^2)$

Implementation, complexity

- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency
- Store frequencies in an array
 - Linear scan to find minimum values
 - $|A| = k$, number of recursive calls is $k-1$
 - Complexity is $O(k^2)$
- Instead, maintain frequencies in an heap
 - Extracting two minimum frequency letters and adding back compound letter are both $O(\log k)$
 - Complexity drops to $O(k \log k)$

Summary

Why is Huffman coding greedy?

- Recursively combine letters with lowest frequencies

Summary

Why is Huffman coding greedy?

- Recursively combine letters with lowest frequencies
- Locally optimal choice

Summary

Why is Huffman coding greedy?

- Recursively combine letters with lowest frequencies
- Locally optimal choice
- Never go back and consider other pairings of letters

Summary

Why is Huffman coding greedy?

- Recursively combine letters with lowest frequencies
- Locally optimal choice
- Never go back and consider other pairings of letters

Historical notes

Summary

Why is Huffman coding greedy?

- Recursively combine letters with lowest frequencies
- Locally optimal choice
- Never go back and consider other pairings of letters

Historical notes

- Claude Shannon invented **information theory**
 - Mathematical lower bounds on the size of optimal encodings
 - Does not describe how to construct optimal codes

Summary

Why is Huffman coding greedy?

- Recursively combine letters with lowest frequencies
- Locally optimal choice
- Never go back and consider other pairings of letters

Historical notes

- Claude Shannon invented **information theory**
 - Mathematical lower bounds on the size of optimal encodings
 - Does not describe how to construct optimal codes
- Shannon and Robert Fano came up with a recursive solution (Shannon-Fano codes) that was not optimal

Summary

Why is Huffman coding greedy?

- Recursively combine letters with lowest frequencies
- Locally optimal choice
- Never go back and consider other pairings of letters

Historical notes

- Claude Shannon invented **information theory**
 - Mathematical lower bounds on the size of optimal encodings
 - Does not describe how to construct optimal codes
- Shannon and Robert Fano came up with a recursive solution (Shannon-Fano codes) that was not optimal
- Huffman was a graduate student in Fano's class and discovered his algorithm during the course

Divide and Conquer: Counting Inversions

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 8

Divide and Conquer

- Break up a problem into disjoint subproblems
- Combine these subproblem solutions efficiently

Divide and Conquer

- Break up a problem into disjoint subproblems
- Combine these subproblem solutions efficiently

Examples

Divide and Conquer

- Break up a problem into disjoint subproblems
- Combine these subproblem solutions efficiently

Examples

- Merge sort
 - Split into left and right half and sort each half separately
 - Merge the sorted halves

Divide and Conquer

- Break up a problem into disjoint subproblems
- Combine these subproblem solutions efficiently

Examples

- Merge sort
 - Split into left and right half and sort each half separately
 - Merge the sorted halves
- Quicksort
 - Rearrange into lower and upper partitions, sort each partition separately
 - Place pivot between sorted lower and upper partitions

Divide and Conquer

- Break up a problem into disjoint subproblems
- Combine these subproblem solutions efficiently

Examples

- Merge sort
 - Split into left and right half and sort each half separately
 - Merge the sorted halves
- Quicksort
 - Rearrange into lower and upper partitions, sort each partition separately
 - Place pivot between sorted lower and upper partitions
- Other examples?

Recommender systems

- Online services recommend items to you

Recommender systems

- Online services recommend items to you
- Compare your profile with other customers

Recommender systems

- Online services recommend items to you
- Compare your profile with other customers
- Identify people who share your likes and dislikes

Recommender systems

- Online services recommend items to you
- Compare your profile with other customers
- Identify people who share your likes and dislikes
- Recommend items that they like

Recommender systems

- Online services recommend items to you
- Compare your profile with other customers
- Identify people who share your likes and dislikes
- Recommend items that they like
- Comparing profiles: how similar are your rankings to those of others?

Recommender systems

- Online services recommend items to you
- Compare your profile with other customers
- Identify people who share your likes and dislikes
- Recommend items that they like
- Comparing profiles: how similar are your rankings to those of others?

Comparing rankings

Recommender systems

- Online services recommend items to you
- Compare your profile with other customers
- Identify people who share your likes and dislikes
- Recommend items that they like
- Comparing profiles: how similar are your rankings to those of others?

Comparing rankings

- You and your friend rank 5 movies, $\{A, B, C, D, E\}$
 - Your ranking: D, B, C, A, E
 - Your friend's ranking: B, A, C, D, E

Recommender systems

- Online services recommend items to you
- Compare your profile with other customers
- Identify people who share your likes and dislikes
- Recommend items that they like
- Comparing profiles: how similar are your rankings to those of others?

Comparing rankings

- You and your friend rank 5 movies, {A, B, C, D, E}
 - Your ranking: D, B, C, A, E
 - Your friend's ranking: B, A, C, D, E
- How to measure how similar these rankings are?

Recommender systems

- Online services recommend items to you
- Compare your profile with other customers
- Identify people who share your likes and dislikes
- Recommend items that they like
- Comparing profiles: how similar are your rankings to those of others?

Comparing rankings

- You and your friend rank 5 movies, {A, B, C, D, E}
 - Your ranking: D, B, C, A, E
 - Your friend's ranking: B, A, C, D, E
- How to measure how similar these rankings are?
- For each pair of movies, compare preferences
 - You rank B above C, so does your friend
 - You rank D above B, your friend ranks B above D

Compare based on inversions

Inversions

- Pair of movies ranked in opposite order
 - You rank D above B, your friend ranks B above D

Compare based on inversions

Inversions

- Pair of movies ranked in opposite order
 - You rank D above B, your friend ranks B above D
- No inversions — rankings identical

Compare based on inversions

Inversions

- Pair of movies ranked in opposite order
 - You rank D above B, your friend ranks B above D
- No inversions — rankings identical
- Every pair inverted — maximally dissimilar

Compare based on inversions

Inversions

- Pair of movies ranked in opposite order
 - You rank D above B, your friend ranks B above D
- No inversions — rankings identical
- Every pair inverted — maximally dissimilar
- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity

Compare based on inversions

Inversions

Permutations

- Pair of movies ranked in opposite order
 - You rank D above B, your friend ranks B above D
- No inversions — rankings identical
- Every pair inverted — maximally dissimilar
- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity

Compare based on inversions

Inversions

- Pair of movies ranked in opposite order
 - You rank D above B, your friend ranks B above D
- No inversions — rankings identical
- Every pair inverted — maximally dissimilar
- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity

Permutations

- Fix the order of one ranking as a sorted sequence $1, 2, \dots, n$

Compare based on inversions

Inversions

- Pair of movies ranked in opposite order
 - You rank D above B, your friend ranks B above D
- No inversions — rankings identical
- Every pair inverted — maximally dissimilar
- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity

Permutations

- Fix the order of one ranking as a sorted sequence $1, 2, \dots, n$
- The other ranking is a permutation of $1, 2, \dots, n$

Compare based on inversions

Inversions

- Pair of movies ranked in opposite order
 - You rank D above B, your friend ranks B above D
- No inversions — rankings identical
- Every pair inverted — maximally dissimilar
- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity

Permutations

- Fix the order of one ranking as a sorted sequence $1, 2, \dots, n$
- The other ranking is a permutation of $1, 2, \dots, n$
- An inversion is a pair (i, j) , $i < j$, where j appears before i

Counting inversions

- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity

Counting inversions

- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity
- Your ranking: D, B, C, A, E
 $D = 1, B = 2, C = 3, A = 4, E = 5$

Counting inversions

- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity
- Your ranking: D, B, C, A, E
 $D = 1, B = 2, C = 3, A = 4, E = 5$
- Your friend's ranking: B, A, C, D, E
 $= 2, 4, 3, 1, 5$

Counting inversions

- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity
- Your ranking: D, B, C, A, E
 $D = 1, B = 2, C = 3, A = 4, E = 5$
- Your friend's ranking: B, A, C, D, E
 $= 2, 4, 3, 1, 5$
- Inversions in $2, 4, 3, 1, 5$?

Counting inversions

- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity
- Your ranking: D, B, C, A, E
 $D = 1, B = 2, C = 3, A = 4, E = 5$
- Your friend's ranking: B, A, C, D, E
 $= 2, 4, 3, 1, 5$
- Inversions in $2, 4, 3, 1, 5$?
- $(1, 2), (1, 3), (1, 4), (3, 4)$

Counting inversions

- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity Graphically
- Your ranking: D, B, C, A, E
 $D = 1, B = 2, C = 3, A = 4, E = 5$
- Your friend's ranking: B, A, C, D, E
 $= 2, 4, 3, 1, 5$
- Inversions in $2, 4, 3, 1, 5$?
 - $(1, 2), (1, 3), (1, 4), (3, 4)$

Counting inversions

- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity
- Your ranking: D, B, C, A, E
 $D = 1, B = 2, C = 3, A = 4, E = 5$
- Your friend's ranking: B, A, C, D, E
 $= 2, 4, 3, 1, 5$
- Inversions in $2, 4, 3, 1, 5$?
- $(1, 2), (1, 3), (1, 4), (3, 4)$

Graphically

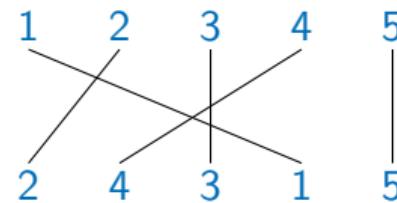
- Write the two permutations as two rows of nodes
- Connect every pair (j, j) between the two rows

Counting inversions

- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity
- Your ranking: D, B, C, A, E
 $D = 1, B = 2, C = 3, A = 4, E = 5$
- Your friend's ranking: B, A, C, D, E
 $= 2, 4, 3, 1, 5$
- Inversions in $2, 4, 3, 1, 5$?
- $(1, 2), (1, 3), (1, 4), (3, 4)$

Graphically

- Write the two permutations as two rows of nodes
- Connect every pair (j, j) between the two rows

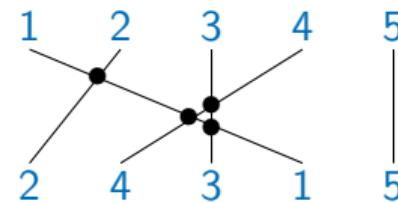


Counting inversions

- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity
- Your ranking: D, B, C, A, E
 $D = 1, B = 2, C = 3, A = 4, E = 5$
- Your friend's ranking: B, A, C, D, E
 $= 2, 4, 3, 1, 5$
- Inversions in $2, 4, 3, 1, 5$?
- $(1, 2), (1, 3), (1, 4), (3, 4)$

Graphically

- Write the two permutations as two rows of nodes
- Connect every pair (j, j') between the two rows



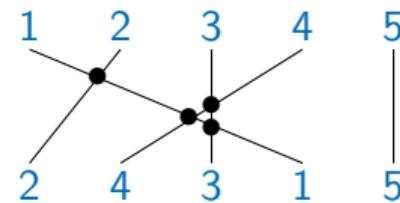
- Every crossing is an inversion

Counting inversions

- Number of inversions ranges from 0 to $n(n - 1)/2$ — measure of dissimilarity
- Your ranking: D, B, C, A, E
 $D = 1, B = 2, C = 3, A = 4, E = 5$
- Your friend's ranking: B, A, C, D, E
 $= 2, 4, 3, 1, 5$
- Inversions in $2, 4, 3, 1, 5$?
- $(1, 2), (1, 3), (1, 4), (3, 4)$

Graphically

- Write the two permutations as two rows of nodes
- Connect every pair (j, j) between the two rows



- Every crossing is an inversion
- Brute force — check every (i, j) , $O(n^2)$

Divide and conquer

- Friend's permutation is i_1, i_2, \dots, i_n

Divide and conquer

- Friend's permutation is i_1, i_2, \dots, i_n
- Divide into two lists
 - $L = [i_1, i_2, \dots, i_{n/2}]$
 - $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$

Divide and conquer

- Friend's permutation is i_1, i_2, \dots, i_n
- Divide into two lists
 - $L = [i_1, i_2, \dots, i_{n/2}]$
 - $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$
- Recursively count inversions in L and R

Divide and conquer

- Friend's permutation is i_1, i_2, \dots, i_n
- Divide into two lists
 - $L = [i_1, i_2, \dots, i_{n/2}]$
 - $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$
- Recursively count inversions in L and R
- Add inversions across the boundary between L and R
 - $i \in L, j \in R, i > j$
 - How many elements in L are bigger than elements in R ?

Divide and conquer

- Friend's permutation is i_1, i_2, \dots, i_n
- Divide into two lists
 - $L = [i_1, i_2, \dots, i_{n/2}]$
 - $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$
- Recursively count inversions in L and R
- Add inversions across the boundary between L and R
 - $i \in L, j \in R, i > j$
 - How many elements in L are bigger than elements in R ?

Divide and conquer

- Friend's permutation is i_1, i_2, \dots, i_n
- Divide into two lists
 - $L = [i_1, i_2, \dots, i_{n/2}]$
 - $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$
- Recursively count inversions in L and R
- Add inversions across the boundary between L and R
 - $i \in L, j \in R, i > j$
 - How many elements in L are bigger than elements in R ?
- How to count inversions across the boundary?
- Adapt merge sort

Divide and conquer

- Friend's permutation is i_1, i_2, \dots, i_n
- Divide into two lists
 - $L = [i_1, i_2, \dots, i_{n/2}]$
 - $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$
- Recursively count inversions in L and R
- Add inversions across the boundary between L and R
 - $i \in L, j \in R, i > j$
 - How many elements in L are bigger than elements in R ?
- How to count inversions across the boundary?
- Adapt merge sort
- Recursively **sort and count** inversions in L and R

Divide and conquer

- Friend's permutation is i_1, i_2, \dots, i_n
- Divide into two lists
 - $L = [i_1, i_2, \dots, i_{n/2}]$
 - $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$
- Recursively count inversions in L and R
- Add inversions across the boundary between L and R
 - $i \in L, j \in R, i > j$
 - How many elements in L are bigger than elements in R ?
- How to count inversions across the boundary?
- Adapt merge sort
- Recursively **sort and count** inversions in L and R
- Count inversions while merging — **merge and count**

Divide and conquer

Merge and count

- Merge $L = [i_1, i_2, \dots, i_{n/2}]$ and $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$, sorted

Divide and conquer

Merge and count

- Merge $L = [i_1, i_2, \dots, i_{n/2}]$ and $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$, sorted
- Count inversions while merging
 - If we add i_m from R to the output, i_m is smaller than elements currently in L
 - i_m is hence inverted with respect to elements currently in L
 - Add current size of L to inversion count

Divide and conquer

Merge and count

- Merge $L = [i_1, i_2, \dots, i_{n/2}]$ and $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$, sorted
- Count inversions while merging
 - If we add i_m from R to the output, i_m is smaller than elements currently in L
 - i_m is hence inverted with respect to elements currently in L
 - Add current size of L to inversion count

```
def mergeAndCount(A,B):  
    (m,n) = (len(A),len(B))  
    (C,i,j,k,count) = ([],0,0,0,0)  
    while k < m+n:  
        if i == m:  
            C.append(B[j])  
            (j,k) = (j+1,k+1)  
        elif j == n:  
            C.append(A[i])  
            (i,k) = (i+1,k+1)  
        elif A[i] < B[j]:  
            C.append(A[i])  
            (i,k) = (i+1,k+1)  
        else:  
            C.append(B[j])  
            (j,k,count) = (j+1,k+1,count+(m-i))  
    return(C,count)
```

Divide and conquer

Merge and count

- Merge $L = [i_1, i_2, \dots, i_{n/2}]$ and $R = [i_{n/2+1}, i_{n/2+2}, \dots, i_n]$, sorted
- Count inversions while merging
 - If we add i_m from R to the output, i_m is smaller than elements currently in L
 - i_m is hence inverted with respect to elements currently in L
 - Add current size of L to inversion count
- `sortAndCount` is merge sort with `mergeAndCount`

```
def sortAndCount(A):  
    n = len(A)  
  
    if n <= 1:  
        return(A,0)  
  
(L,countL) = sortAndCount(A[:n//2])  
(R,countR) = sortAndCount(A[n//2:])  
  
(B,countB) = mergeAndCount(L,R)  
  
return(B,countL+countR+countB)
```

Analysis

- Recurrence is similar to merge sort

- $\blacksquare T(0) = T(1) = 1$

- $\blacksquare T(n) = 2T(n/2) + n$

Analysis

- Recurrence is similar to merge sort
 - $T(0) = T(1) = 1$
 - $T(n) = 2T(n/2) + n$
- Solve to get $T(n) = O(n \log n)$

Analysis

- Recurrence is similar to merge sort
 - $T(0) = T(1) = 1$
 - $T(n) = 2T(n/2) + n$
- Solve to get $T(n) = O(n \log n)$
- Note that the number of inversions can still be $O(n^2)$
 - Number ranges from 0 to $n(n - 1)/2$

Analysis

- Recurrence is similar to merge sort
 - $T(0) = T(1) = 1$
 - $T(n) = 2T(n/2) + n$
- Solve to get $T(n) = O(n \log n)$
- Note that the number of inversions can still be $O(n^2)$
 - Number ranges from 0 to $n(n - 1)/2$
- We are counting them efficiently without enumerating each one

Divide and Conquer: Closest Pair of Points

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 8

Recall: Video Game

- Several objects on screen
- Basic step: find closest pair of objects

Recall: Video Game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
 - For each pair of objects, compute their distance
 - Report minimum distance across all pairs

Recall: Video Game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
 - For each pair of objects, compute their distance
 - Report minimum distance across all pairs
- There is a clever algorithm that takes time $n \log n$

Recall: Video Game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
 - For each pair of objects, compute their distance
 - Report minimum distance across all pairs
- There is a clever algorithm that takes time $n \log n$
- Use divide and conquer

The problem statement

- Points p in 2D — $p = (x, y)$

The problem statement

- Points p in 2D — $p = (x, y)$
- Usual Euclidean distance between $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$
 - $d(p_1, p_2) = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$

The problem statement

- Points p in 2D — $p = (x, y)$
- Usual Euclidean distance between $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$
 - $d(p_1, p_2) = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$
- Given n points p_1, p_2, \dots, p_n , find the closest pair
 - Assume no two points have same x or y coordinate
 - Can always rotate points slightly to ensure this

The problem statement

- Points p in 2D — $p = (x, y)$
- Usual Euclidean distance between $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$
 - $d(p_1, p_2) = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$
- Given n points p_1, p_2, \dots, p_n , find the closest pair
 - Assume no two points have same x or y coordinate
 - Can always rotate points slightly to ensure this
 - ... or modify the algorithm slightly!

The problem statement

- Points p in 2D — $p = (x, y)$
- Usual Euclidean distance between $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$
 - $d(p_1, p_2) = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$
- Given n points p_1, p_2, \dots, p_n , find the closest pair
 - Assume no two points have same x or y coordinate
 - Can always rotate points slightly to ensure this
 - ... or modify the algorithm slightly!
- Brute force
 - Compute $d(p_i, p_j)$ for every pair of points
 - $O(n^2)$

Finding the closest pair of points

In 1 dimension

Finding the closest pair of points

In 1 dimension

- Given n 1D points x_1, x_2, \dots, x_n , find the closest pair
 - $d(p_i, p_j) = |x_j - x_i|$

Finding the closest pair of points

In 1 dimension

- Given n 1D points x_1, x_2, \dots, x_n , find the closest pair
 - $d(p_i, p_j) = |x_j - x_i|$
- Sort the points — $O(n \log n)$

Finding the closest pair of points

In 1 dimension

- Given n 1D points x_1, x_2, \dots, x_n , find the closest pair
 - $d(p_i, p_j) = |x_j - x_i|$
- Sort the points — $O(n \log n)$
- In sorted order, nearest points to p are its neighbours
 - $O(n)$ scan to find minimum separation between adjacent points

Finding the closest pair of points

In 1 dimension

- Given n 1D points x_1, x_2, \dots, x_n , find the closest pair
 - $d(p_i, p_j) = |x_j - x_i|$
- Sort the points — $O(n \log n)$
- In sorted order, nearest points to p are its neighbours
 - $O(n)$ scan to find minimum separation between adjacent points

In 2 dimensions

- Divide and conquer

Finding the closest pair of points

In 1 dimension

- Given n 1D points x_1, x_2, \dots, x_n , find the closest pair
 - $d(p_i, p_j) = |x_j - x_i|$
- Sort the points — $O(n \log n)$
- In sorted order, nearest points to p are its neighbours
 - $O(n)$ scan to find minimum separation between adjacent points

In 2 dimensions

- Divide and conquer
- Split the points into two halves by vertical line

Finding the closest pair of points

In 1 dimension

- Given n 1D points x_1, x_2, \dots, x_n , find the closest pair
 - $d(p_i, p_j) = |x_j - x_i|$
- Sort the points — $O(n \log n)$
- In sorted order, nearest points to p are its neighbours
 - $O(n)$ scan to find minimum separation between adjacent points

In 2 dimensions

- Divide and conquer
- Split the points into two halves by vertical line
- Recursively compute closest pair in each half

Finding the closest pair of points

In 1 dimension

- Given n 1D points x_1, x_2, \dots, x_n , find the closest pair
 - $d(p_i, p_j) = |x_j - x_i|$
- Sort the points — $O(n \log n)$
- In sorted order, nearest points to p are its neighbours
 - $O(n)$ scan to find minimum separation between adjacent points

In 2 dimensions

- Divide and conquer
- Split the points into two halves by vertical line
- Recursively compute closest pair in each half
- Compare shortest distance in each half to shortest distance across the dividing line

Finding the closest pair of points

In 1 dimension

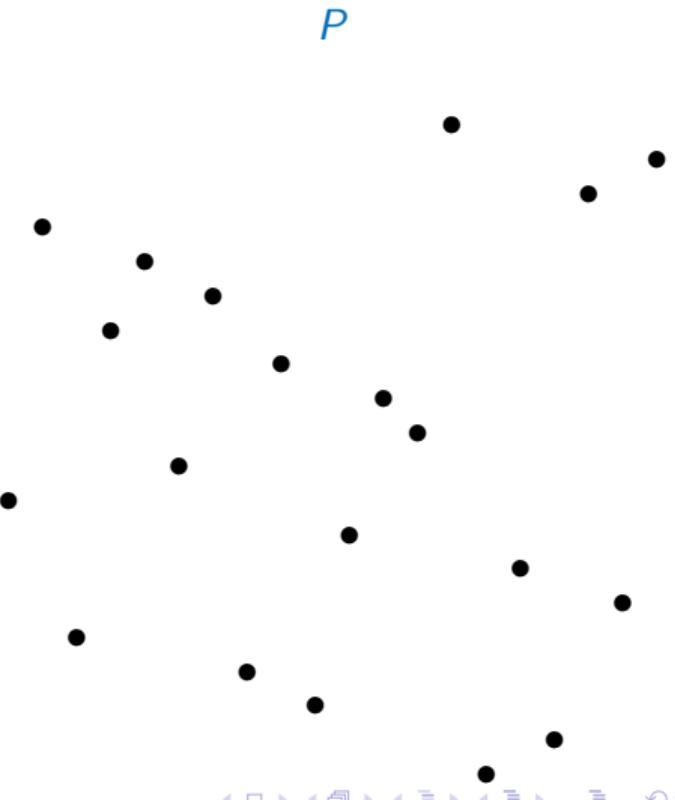
- Given n 1D points x_1, x_2, \dots, x_n , find the closest pair
 - $d(p_i, p_j) = |x_j - x_i|$
- Sort the points — $O(n \log n)$
- In sorted order, nearest points to p are its neighbours
 - $O(n)$ scan to find minimum separation between adjacent points

In 2 dimensions

- Divide and conquer
- Split the points into two halves by vertical line
- Recursively compute closest pair in each half
- Compare shortest distance in each half to shortest distance across the dividing line
- How to do this efficiently?

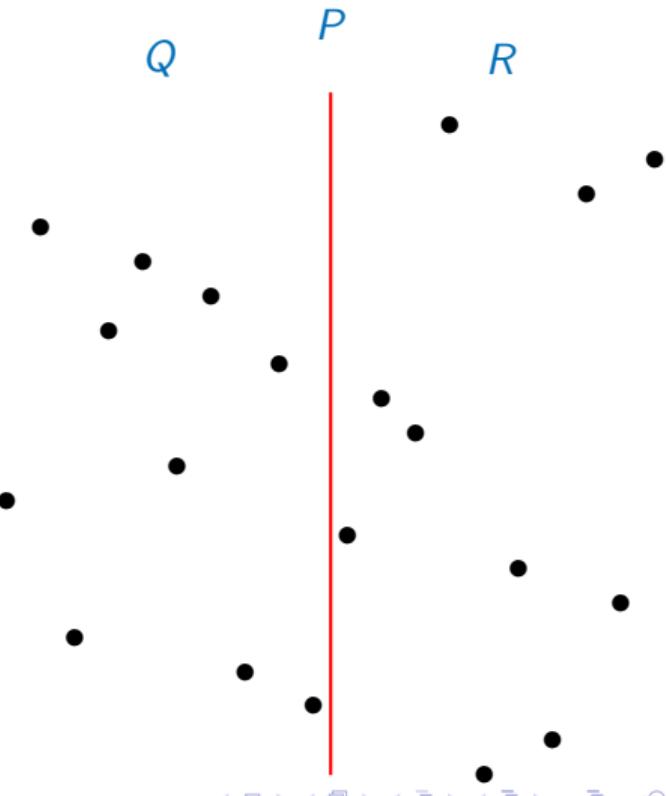
Dividing points

- Given n points $P = \{p_1, p_2, \dots, p_n\}$, compute
 - P_x , P sorted by x -coordinate
 - P_y , P sorted by y -coordinate



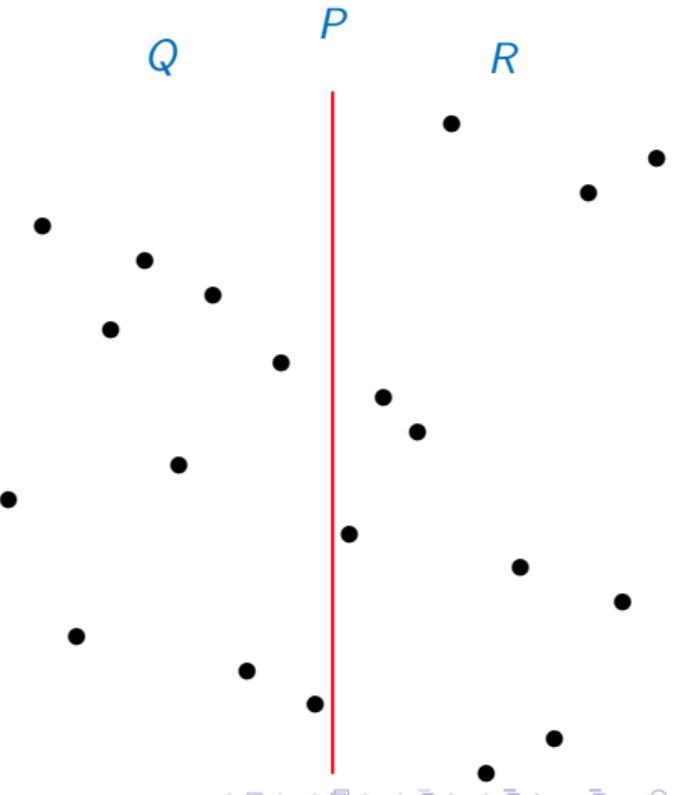
Dividing points

- Given n points $P = \{p_1, p_2, \dots, p_n\}$, compute
 - P_x , P sorted by x -coordinate
 - P_y , P sorted by y -coordinate
- Divide P by vertical line into equal size Q , R



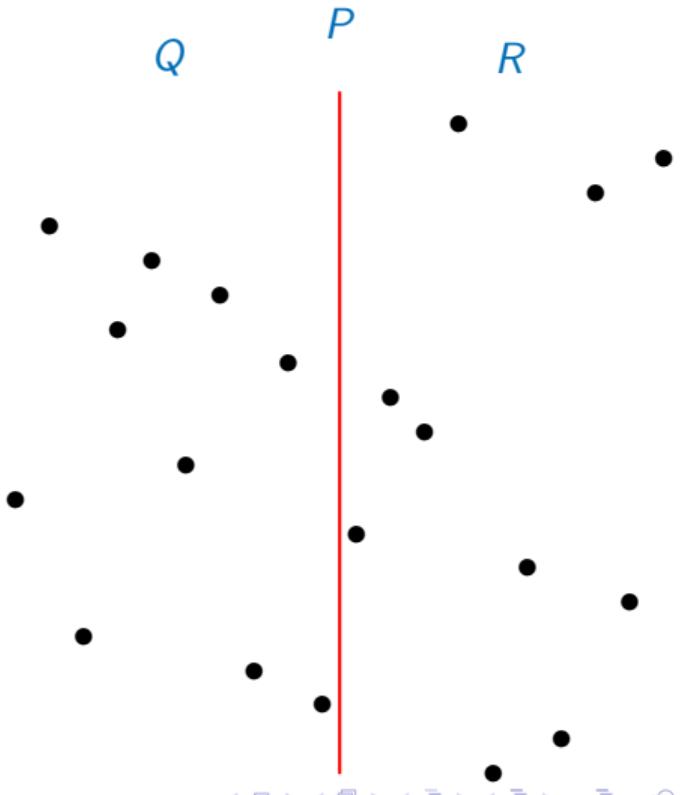
Dividing points

- Given n points $P = \{p_1, p_2, \dots, p_n\}$, compute
 - P_x , P sorted by x -coordinate
 - P_y , P sorted by y -coordinate
- Divide P by vertical line into equal size Q , R
- How to compute Q_x , Q_y , R_x , R_y efficiently?



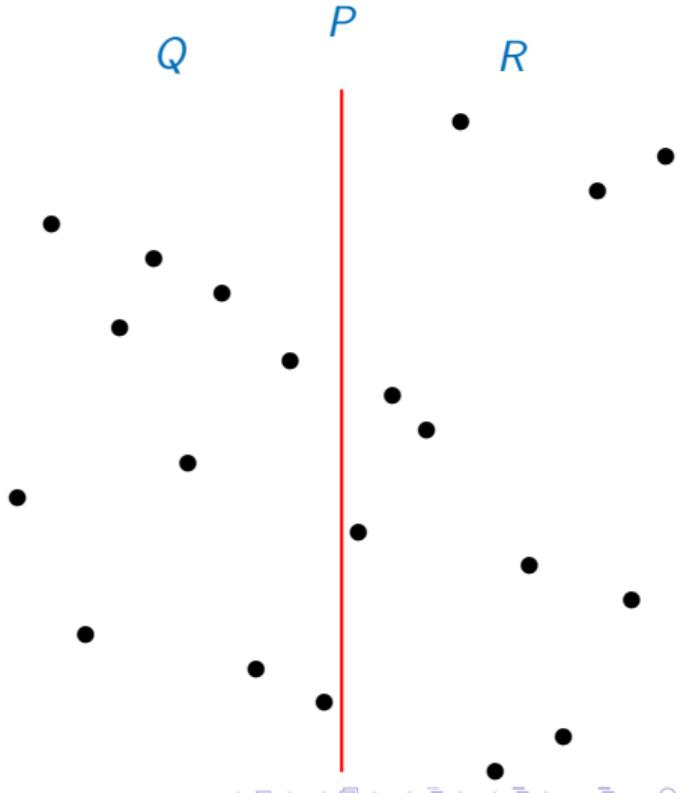
Dividing points

- Given n points $P = \{p_1, p_2, \dots, p_n\}$, compute
 - P_x , P sorted by x -coordinate
 - P_y , P sorted by y -coordinate
- Divide P by vertical line into equal size Q , R
- How to compute Q_x , Q_y , R_x , R_y efficiently?
- Q_x is first half of P_x , R_x is second half of P_x



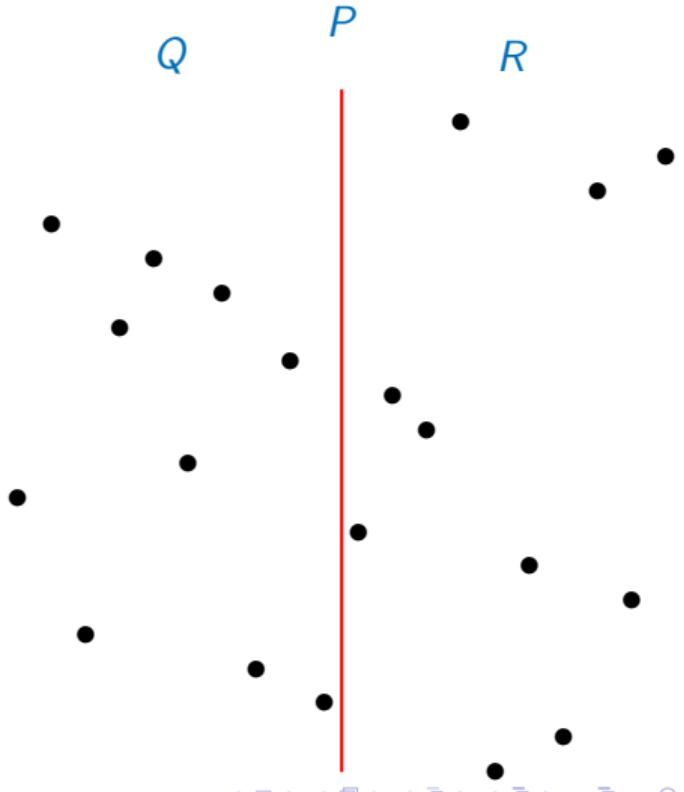
Dividing points

- Given n points $P = \{p_1, p_2, \dots, p_n\}$, compute
 - P_x , P sorted by x -coordinate
 - P_y , P sorted by y -coordinate
- Divide P by vertical line into equal size Q , R
- How to compute Q_x , Q_y , R_x , R_y efficiently?
- Q_x is first half of P_x , R_x is second half of P_x
- Let x_R be smallest x coordinate in R



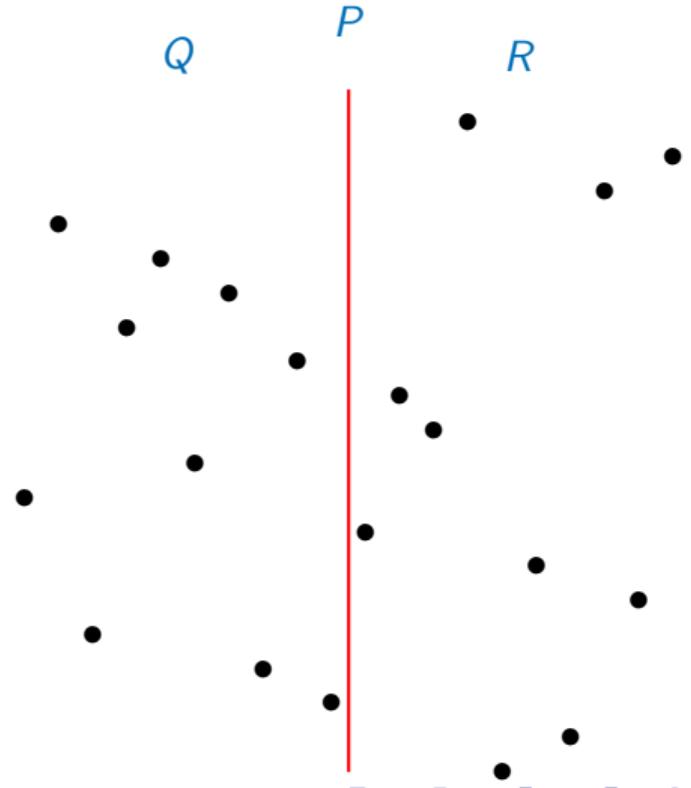
Dividing points

- Given n points $P = \{p_1, p_2, \dots, p_n\}$, compute
 - P_x , P sorted by x -coordinate
 - P_y , P sorted by y -coordinate
- Divide P by vertical line into equal size Q , R
- How to compute Q_x , Q_y , R_x , R_y efficiently?
- Q_x is first half of P_x , R_x is second half of P_x
- Let x_R be smallest x coordinate in R
- For $p \in P_y$, if x coordinate of p less than x_R , move p to Q_y , else R_y



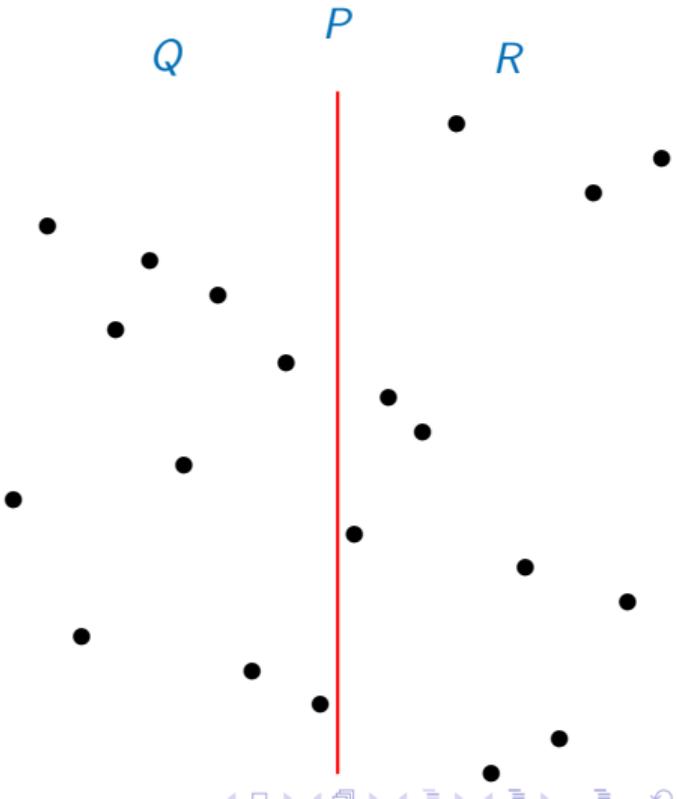
Dividing points

- Given n points $P = \{p_1, p_2, \dots, p_n\}$, compute
 - P_x , P sorted by x -coordinate
 - P_y , P sorted by y -coordinate
- Divide P by vertical line into equal size Q , R
- How to compute Q_x , Q_y , R_x , R_y efficiently?
- Q_x is first half of P_x , R_x is second half of P_x
- Let x_R be smallest x coordinate in R
- For $p \in P_y$, if x coordinate of p less than x_R , move p to Q_y , else R_y
- All of this can be done in $O(n)$



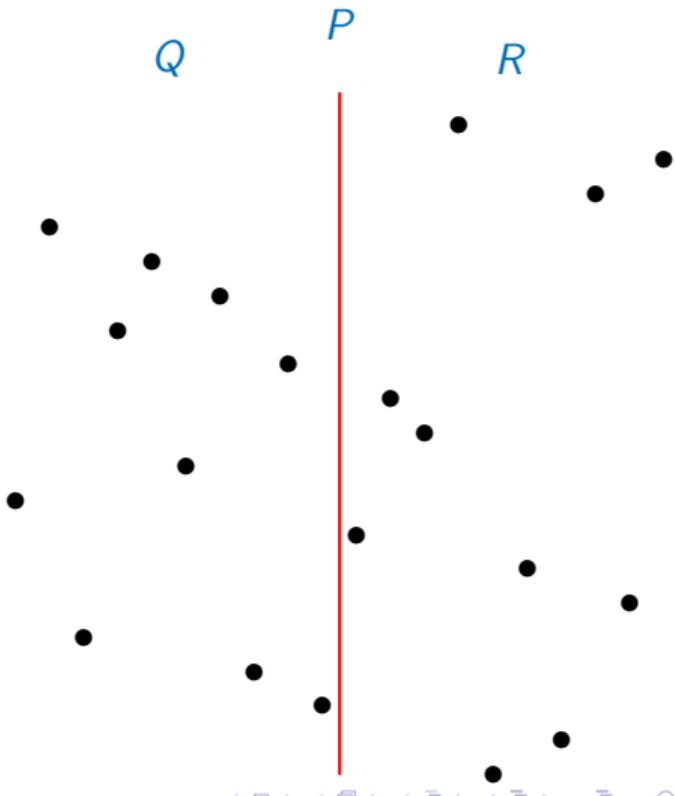
Divide and conquer

- Want to compute $\text{ClosestPair}(P_x, P_y)$



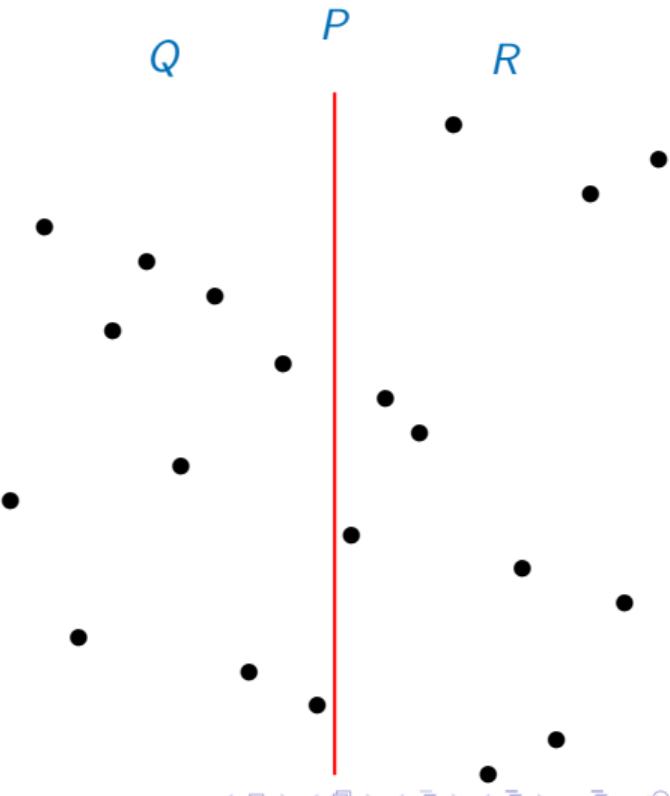
Divide and conquer

- Want to compute $\text{ClosestPair}(P_x, P_y)$
- Split (P_x, P_y) as $(Q_x, Q_y), (R_x, R_y)$



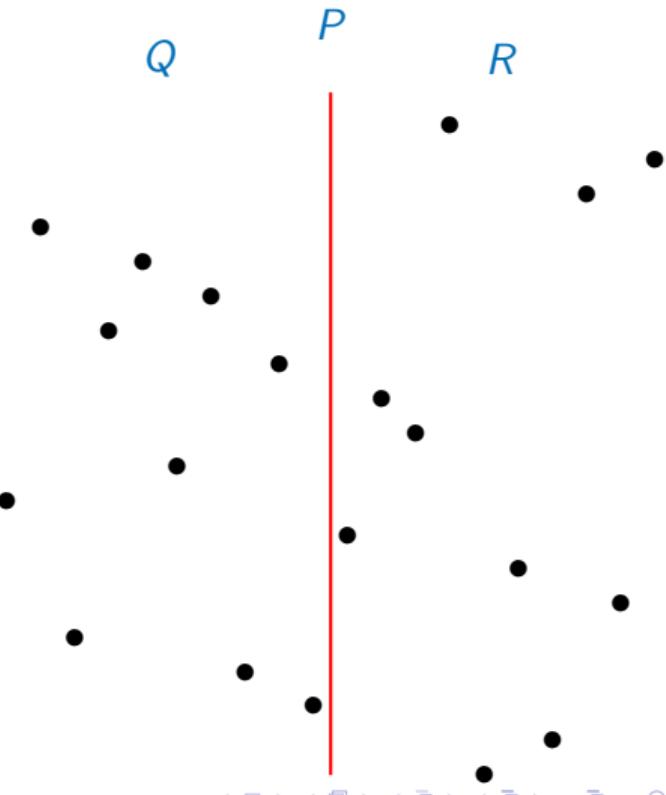
Divide and conquer

- Want to compute $\text{ClosestPair}(P_x, P_y)$
- Split (P_x, P_y) as $(Q_x, Q_y), (R_x, R_y)$
- Recursively compute $\text{ClosestPair}(Q_x, Q_y)$ and $\text{ClosestPair}(R_x, R_y)$



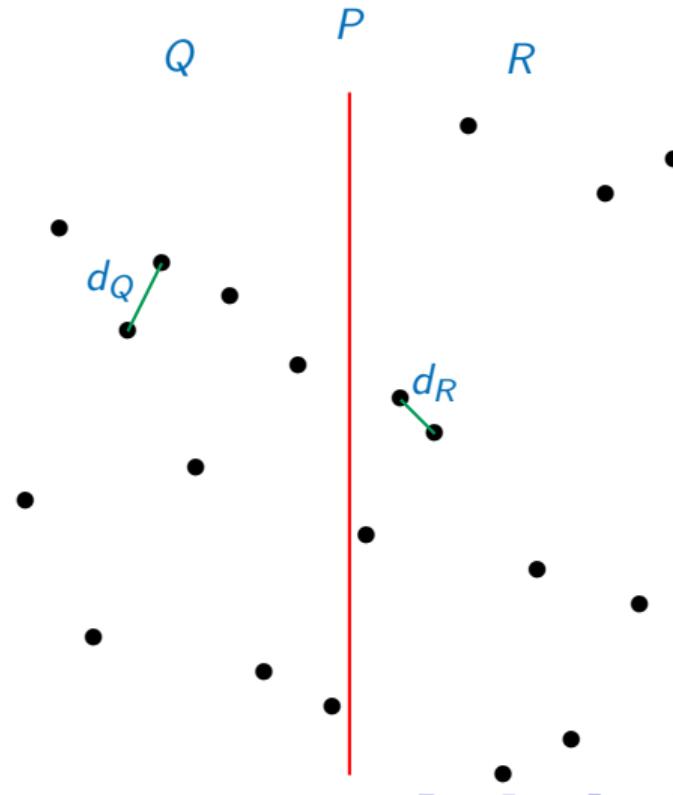
Divide and conquer

- Want to compute $\text{ClosestPair}(P_x, P_y)$
- Split (P_x, P_y) as $(Q_x, Q_y), (R_x, R_y)$
- Recursively compute $\text{ClosestPair}(Q_x, Q_y)$ and $\text{ClosestPair}(R_x, R_y)$
- How to combine these recursive solutions?



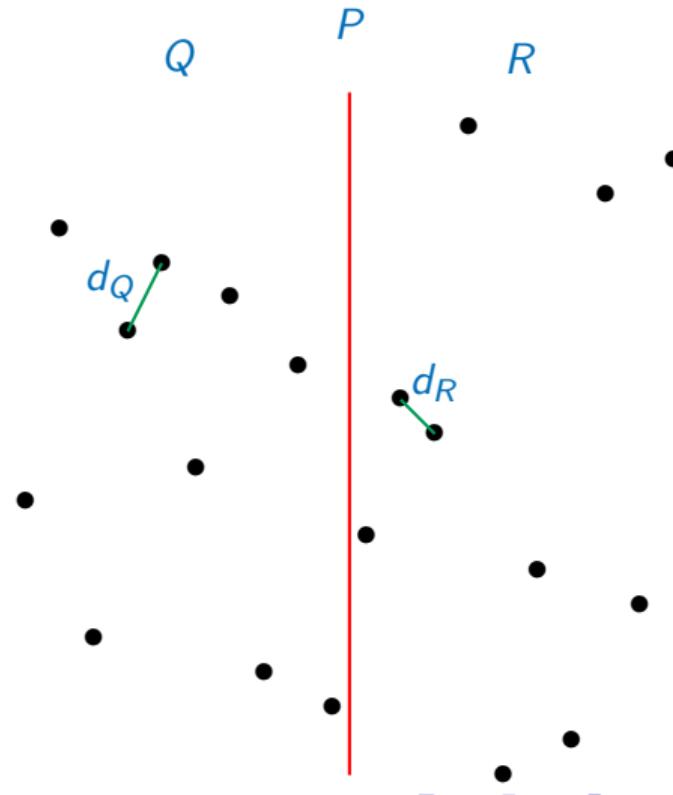
Combining solutions

- Let d_Q, d_R be closest distances in Q , R , respectively



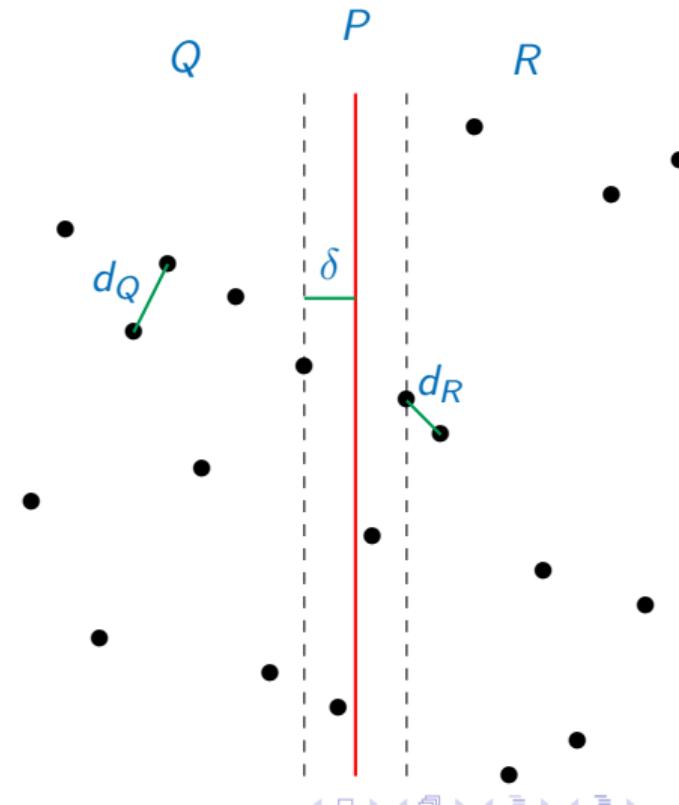
Combining solutions

- Let d_Q, d_R be closest distances in Q , R , respectively
- Set $\delta = \min(d_Q, d_R)$



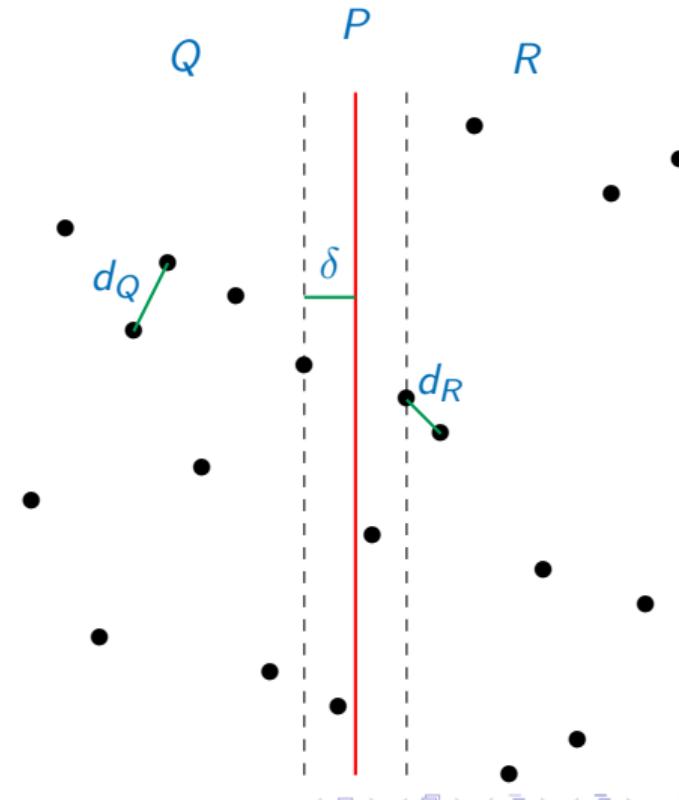
Combining solutions

- Let d_Q, d_R be closest distances in Q , R , respectively
- Set $\delta = \min(d_Q, d_R)$
- Only need to consider points within distance δ on either side of the separator



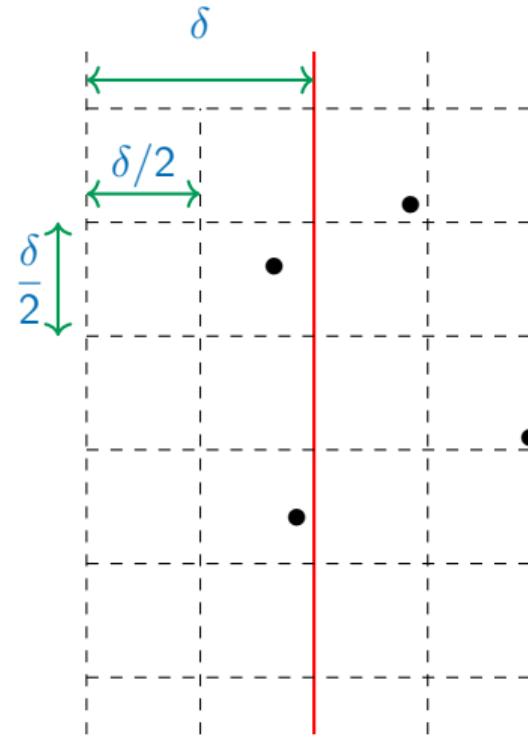
Combining solutions

- Let d_Q, d_R be closest distances in Q , R , respectively
- Set $\delta = \min(d_Q, d_R)$
- Only need to consider points within distance δ on either side of the separator
- No pair outside this band can be closer than δ



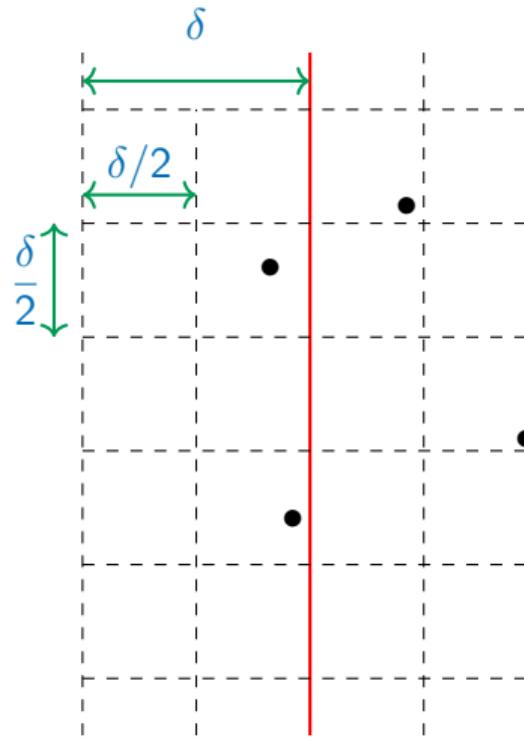
Combining solutions

- Divide the distance δ band into boxes of side $\delta/2$



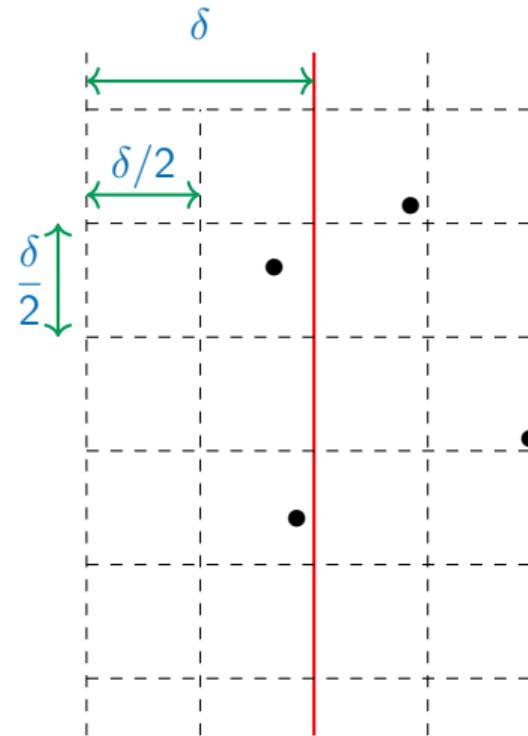
Combining solutions

- Divide the distance δ band into boxes of side $\delta/2$
- Cannot have two points inside the same box
 - Box diagonal is $\delta/\sqrt{2}$



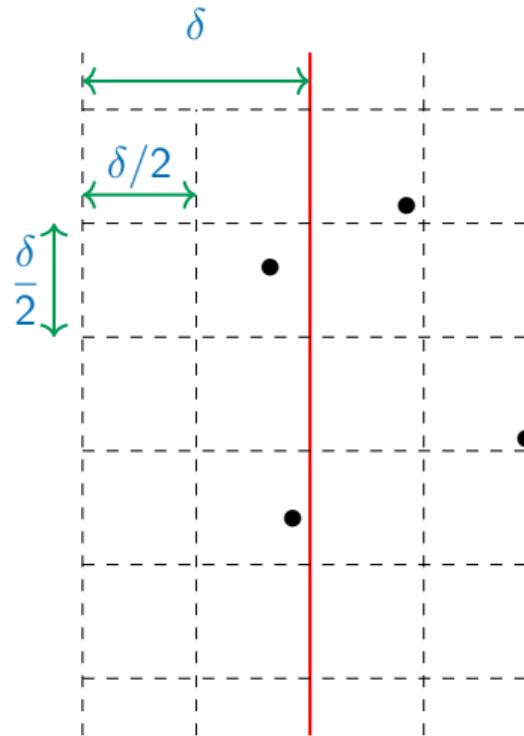
Combining solutions

- Divide the distance δ band into boxes of side $\delta/2$
- Cannot have two points inside the same box
 - Box diagonal is $\delta/\sqrt{2}$
- Any point within distance δ must lie in a 4×4 neighbourhood of boxes
 - Check each point against 15 others



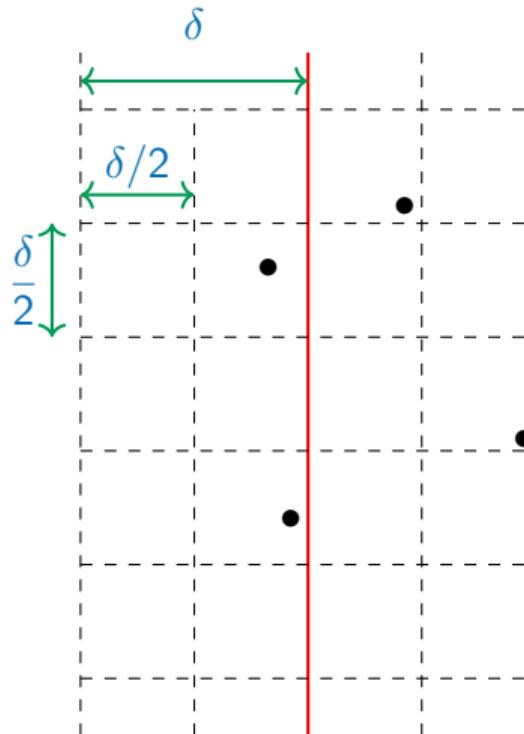
Combining solutions

- Divide the distance δ band into boxes of side $\delta/2$
- Cannot have two points inside the same box
 - Box diagonal is $\delta/\sqrt{2}$
- Any point within distance δ must lie in a 4×4 neighbourhood of boxes
 - Check each point against 15 others
- From Q_y , R_y , extract S_y , points in δ band sorted by y



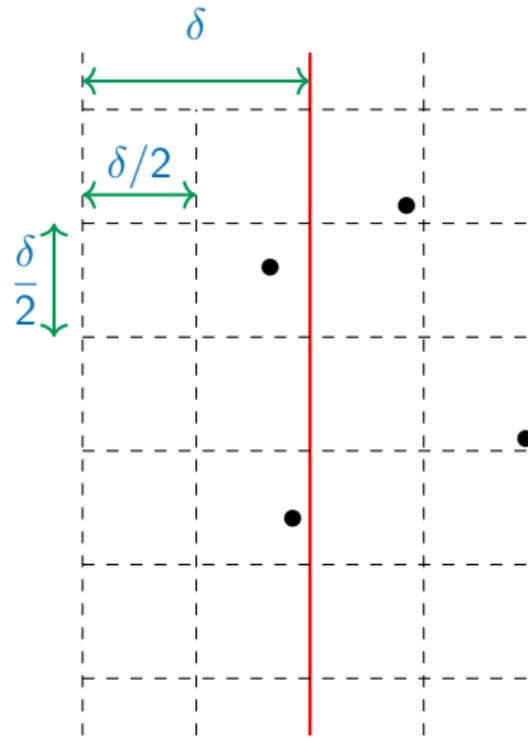
Combining solutions

- Divide the distance δ band into boxes of side $\delta/2$
- Cannot have two points inside the same box
 - Box diagonal is $\delta/\sqrt{2}$
- Any point within distance δ must lie in a 4×4 neighbourhood of boxes
 - Check each point against 15 others
- From Q_y , R_y , extract S_y , points in δ band sorted by y
- Scan S_y from bottom to top, comparing each p with next 15 points in S_y



Combining solutions

- Divide the distance δ band into boxes of side $\delta/2$
- Cannot have two points inside the same box
 - Box diagonal is $\delta/\sqrt{2}$
- Any point within distance δ must lie in a 4×4 neighbourhood of boxes
 - Check each point against 15 others
- From Q_y , R_y , extract S_y , points in δ band sorted by y
- Scan S_y from bottom to top, comparing each p with next 15 points in S_y
- Linear scan



Algorithm and analysis

Pseudocode

```
def ClosestPair(Px,Py):  
  
    if len(Px) <= 3:  
        compute pairwise distances  
        return closest pair and distance  
  
    Construct (Qx,Qy), (Rx,Ry)  
  
    (q1,q2,dQ) = ClosestPair(Qx,Qy)  
    (r1,r2,dR) = ClosestPair(Rx,Ry)  
  
    Construct Sy from Qy,Ry  
    Scan Sy, find (s1,s2,dS)  
  
    return (q1,q2,dQ), (r1,r2,dR), (s1,s2,dS)  
    depending on which of dQ, dR, dS is minimum
```

Algorithm and analysis

Pseudocode

Analysis

```
def ClosestPair(Px,Py):  
  
    if len(Px) <= 3:  
        compute pairwise distances  
        return closest pair and distance  
  
    Construct (Qx,Qy), (Rx,Ry)  
  
    (q1,q2,dQ) = ClosestPair(Qx,Qy)  
    (r1,r2,dR) = ClosestPair(Rx,Ry)  
  
    Construct Sy from Qy,Ry  
    Scan Sy, find (s1,s2,dS)  
  
    return (q1,q2,dQ), (r1,r2,dR), (s1,s2,dS)  
    depending on which of dQ, dR, dS is minimum
```

Algorithm and analysis

Pseudocode

```
def ClosestPair(Px,Py):  
  
    if len(Px) <= 3:  
        compute pairwise distances  
        return closest pair and distance  
  
    Construct (Qx,Qy), (Rx,Ry)  
  
    (q1,q2,dQ) = ClosestPair(Qx,Qy)  
    (r1,r2,dR) = ClosestPair(Rx,Ry)  
  
    Construct Sy from Qy,Ry  
    Scan Sy, find (s1,s2,dS)  
  
    return (q1,q2,dQ), (r1,r2,dR), (s1,s2,dS)  
depending on which of dQ, dR, dS is minimum
```

Analysis

- Sort P to get P_x, P_y — $O(n \log n)$

Algorithm and analysis

Pseudocode

```
def ClosestPair(Px,Py):  
  
    if len(Px) <= 3:  
        compute pairwise distances  
        return closest pair and distance  
  
    Construct (Qx,Qy), (Rx,Ry)  
  
    (q1,q2,dQ) = ClosestPair(Qx,Qy)  
    (r1,r2,dR) = ClosestPair(Rx,Ry)  
  
    Construct Sy from Qy,Ry  
    Scan Sy, find (s1,s2,dS)  
  
    return (q1,q2,dQ), (r1,r2,dR), (s1,s2,dS)  
    depending on which of dQ, dR, dS is minimum
```

Analysis

- Sort P to get P_x, P_y — $O(n \log n)$
- Recursive algorithm
 - Construct $(Q_x, Q_y), (R_x, R_y)$ — $O(n)$
 - Construct S_y from Q_y, R_y — $O(n)$
 - Scan S_y — $O(n)$

Algorithm and analysis

Pseudocode

```
def ClosestPair(Px,Py):  
  
    if len(Px) <= 3:  
        compute pairwise distances  
        return closest pair and distance  
  
    Construct (Qx,Qy), (Rx,Ry)  
  
    (q1,q2,dQ) = ClosestPair(Qx,Qy)  
    (r1,r2,dR) = ClosestPair(Rx,Ry)  
  
    Construct Sy from Qy,Ry  
    Scan Sy, find (s1,s2,dS)  
  
    return (q1,q2,dQ), (r1,r2,dR), (s1,s2,dS)  
    depending on which of dQ, dR, dS is minimum
```

Analysis

- Sort P to get P_x, P_y — $O(n \log n)$
- Recursive algorithm
 - Construct $(Q_x, Q_y), (R_x, R_y)$ — $O(n)$
 - Construct S_y from Q_y, R_y — $O(n)$
 - Scan S_y — $O(n)$
- Recurrence: $T(n) = 2T(n/2) + O(n)$, like merge sort

Algorithm and analysis

Pseudocode

```
def ClosestPair(Px,Py):  
  
    if len(Px) <= 3:  
        compute pairwise distances  
        return closest pair and distance  
  
    Construct (Qx,Qy), (Rx,Ry)  
  
    (q1,q2,dQ) = ClosestPair(Qx,Qy)  
    (r1,r2,dR) = ClosestPair(Rx,Ry)  
  
    Construct Sy from Qy,Ry  
    Scan Sy, find (s1,s2,dS)  
  
    return (q1,q2,dQ), (r1,r2,dR), (s1,s2,dS)  
    depending on which of dQ, dR, dS is minimum
```

Analysis

- Sort P to get P_x, P_y — $O(n \log n)$
- Recursive algorithm
 - Construct $(Q_x, Q_y), (R_x, R_y)$ — $O(n)$
 - Construct S_y from Q_y, R_y — $O(n)$
 - Scan S_y — $O(n)$
- Recurrence: $T(n) = 2T(n/2) + O(n)$,
like merge sort
- Overall, $O(n \log n)$

Divide and Conquer: Integer Multiplication

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 8

Integer Multiplication

- How do we multiply two integers x , y ?

Integer Multiplication

- How do we multiply two integers x , y ?
- Form **partial products** — multiply each digit of y separately by x

$$\begin{array}{r} 12 \\ \times 13 \\ \hline 36 \\ 12 \\ \hline 156 \end{array}$$

Integer Multiplication

- How do we multiply two integers x , y ?
- Form **partial products** — multiply each digit of y separately by x
- Add up all the partial products

$$\begin{array}{r} 12 \\ \times 13 \\ \hline 36 \\ 12 \\ \hline 156 \end{array}$$

Integer Multiplication

- How do we multiply two integers x, y ?
- Form **partial products** — multiply each digit of y separately by x
- Add up all the partial products
- Works the same in any base — e.g., binary

$$\begin{array}{r} 12 \\ \times 13 \\ \hline 36 \\ 12 \\ \hline 156 \end{array} \qquad \begin{array}{r} 1100 \\ \times 1101 \\ \hline \end{array}$$
$$\begin{array}{r} 1100 \\ 0000 \\ 1100 \\ \hline 10011100 \end{array}$$

Integer Multiplication

- How do we multiply two integers x, y ?
- Form **partial products** — multiply each digit of y separately by x
- Add up all the partial products
- Works the same in any base — e.g., binary
- To multiply two n -bit numbers

$$\begin{array}{r} 12 \\ \times 13 \\ \hline 36 \\ 12 \\ \hline 156 \end{array} \quad \begin{array}{r} 1100 \\ \times 1101 \\ \hline \end{array}$$
$$\begin{array}{r} 1100 \\ 0000 \\ \hline 1100 \\ 1100 \\ \hline 10011100 \end{array}$$

- n partial products
- Adding each partial product to cumulative sum is $O(n)$
- Overall $O(n^2)$

Integer Multiplication

- How do we multiply two integers x, y ?
- Form **partial products** — multiply each digit of y separately by x
- Add up all the partial products
- Works the same in any base — e.g., binary
- To multiply two n -bit numbers
 - n partial products
 - Adding each partial product to cumulative sum is $O(n)$
 - Overall $O(n^2)$
- Can we improve on this?
 - Each partial product seems “necessary”

$$\begin{array}{r} 12 \\ \times 13 \\ \hline 36 \\ 12 \\ \hline 156 \end{array} \quad \begin{array}{r} 1100 \\ \times 1101 \\ \hline \end{array}$$
$$\begin{array}{r} 1100 \\ 0000 \\ 1100 \\ 1100 \\ \hline 10011100 \end{array}$$

Divide and conquer

- Split the n bits into two groups of $n/2$

$$\begin{array}{ll} x_1 & x_0 \\ \times & b_{n-1} b_{n-2} \cdots b_{\frac{n}{2}} \quad b_{\frac{n}{2}-1} b_{\frac{n}{2}-2} \cdots b_0 \\ y_1 & y_0 \\ \times & b'_{n-1} b'_{n-2} \cdots b'_{\frac{n}{2}} \quad b'_{\frac{n}{2}-1} b'_{\frac{n}{2}-2} \cdots b'_0 \end{array}$$

$$\begin{array}{r} 12 \\ \times 13 \\ \hline 36 \\ 12 \\ \hline 156 \\ \hline 10011100 \end{array} \qquad \begin{array}{r} 1100 \\ \times 1101 \\ \hline 1100 \\ 0000 \\ 1100 \\ \hline 1100 \end{array}$$

Divide and conquer

- Split the n bits into two groups of $n/2$

$$\begin{array}{ll} x_1 & x_0 \\ \times & b_{n-1} b_{n-2} \cdots b_{\frac{n}{2}} \quad b_{\frac{n}{2}-1} b_{\frac{n}{2}-2} \cdots b_0 \\ y_1 & y_0 \\ y & b'_{n-1} b'_{n-2} \cdots b'_{\frac{n}{2}} \quad b'_{\frac{n}{2}-1} b'_{\frac{n}{2}-2} \cdots b'_0 \end{array}$$

$$\begin{array}{r} 12 \\ \times 13 \\ \hline 36 \\ 12 \\ \hline 156 \\ \hline 10011100 \end{array} \qquad \begin{array}{r} 1100 \\ \times 1101 \\ \hline 1100 \\ 0000 \\ 1100 \\ \hline 1100 \end{array}$$

- Rewrite xy as

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

Divide and conquer

- Split the n bits into two groups of $n/2$

$$\begin{array}{ll} x_1 & x_0 \\ \times & b_{n-1} b_{n-2} \cdots b_{\frac{n}{2}} \quad b_{\frac{n}{2}-1} b_{\frac{n}{2}-2} \cdots b_0 \\ y_1 & y_0 \\ \times & b'_{n-1} b'_{n-2} \cdots b'_{\frac{n}{2}} \quad b'_{\frac{n}{2}-1} b'_{\frac{n}{2}-2} \cdots b'_0 \end{array}$$

$$\begin{array}{r} 12 \\ \times 13 \\ \hline 36 \\ 12 \\ \hline 156 \\ \hline 10011100 \end{array} \qquad \begin{array}{r} 1100 \\ \times 1101 \\ \hline 1100 \\ 0000 \\ 1100 \\ \hline 1100 \end{array}$$

- Rewrite xy as

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

- Regroup as

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

Divide and conquer

- Split the n bits into two groups of $n/2$

$$\begin{array}{rcc} & x_1 & x_0 \\ \times & b_{n-1} b_{n-2} \cdots b_{\frac{n}{2}} & b_{\frac{n}{2}-1} b_{\frac{n}{2}-2} \cdots b_0 \\ \hline & y_1 & y_0 \\ y & b'_{n-1} b'_{n-2} \cdots b'_{\frac{n}{2}} & b'_{\frac{n}{2}-1} b'_{\frac{n}{2}-2} \cdots b'_0 \end{array}$$

$$\begin{array}{rcc} & 12 & 1100 \\ \times & 13 & 1101 \\ \hline & 36 & \\ & 12 & 0000 \\ \hline & 156 & 1100 \\ \hline & & 10011100 \end{array}$$

- Rewrite xy as
 $(x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0)$
- Regroup as
 $x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0$
- Four $n/2$ -bit multiplications

Divide and conquer

- Split the n bits into two groups of $n/2$

$$x \quad b_{n-1} b_{n-2} \cdots b_{\frac{n}{2}} \quad b_{\frac{n}{2}-1} b_{\frac{n}{2}-2} \cdots b_0$$

$x_1 \qquad \qquad \qquad x_0$

$$y \quad b'_{n-1} b'_{n-2} \cdots b'_{\frac{n}{2}} \quad b'_{\frac{n}{2}-1} b'_{\frac{n}{2}-2} \cdots b'_0$$

$y_1 \qquad \qquad \qquad y_0$

- $T(1) = 1, T(n) = 4T(n/2) + n$

- Combining the partial products requires adding $O(n)$ bit numbers

- Rewrite xy as

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

- Regroup as

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

- Four $n/2$ -bit multiplications

Divide and conquer

- Split the n bits into two groups of $n/2$

$$\begin{array}{ccccccc} & x_1 & & & x_0 & & \\ x & b_{n-1} b_{n-2} \cdots b_{\frac{n}{2}} & b_{\frac{n}{2}-1} b_{\frac{n}{2}-2} \cdots b_0 & & & & \\ & y_1 & & & y_0 & & \\ y & b'_{n-1} b'_{n-2} \cdots b'_{\frac{n}{2}} & b'_{\frac{n}{2}-1} b'_{\frac{n}{2}-2} \cdots b'_0 & & & & \end{array}$$

- $T(1) = 1, T(n) = 4T(n/2) + n$
 - Combining the partial products requires adding $O(n)$ bit numbers
- $T(n) = 4T(n/2) + n$

- Rewrite xy as

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

- Regroup as

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

- Four $n/2$ -bit multiplications

Divide and conquer

- Split the n bits into two groups of $n/2$

$$\begin{array}{rccccc} & x_1 & & & x_0 & \\ \times & b_{n-1} b_{n-2} \cdots b_{\frac{n}{2}} & & b_{\frac{n}{2}-1} b_{\frac{n}{2}-2} \cdots b_0 & & \\ & y_1 & & y_0 & & \\ y & b'_{n-1} b'_{n-2} \cdots b'_{\frac{n}{2}} & & b'_{\frac{n}{2}-1} b'_{\frac{n}{2}-2} \cdots b'_0 & & \end{array}$$

- $T(1) = 1, T(n) = 4T(n/2) + n$
 - Combining the partial products requires adding $O(n)$ bit numbers
- $$\begin{aligned} T(n) &= 4T(n/2) + n \\ &= 4(4T(n/4) + n/2) + n \\ &= 4^2 T(n/2^2) + (2+1)n \end{aligned}$$

- Rewrite xy as $(x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0)$
- Regroup as $x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0$
- Four $n/2$ -bit multiplications

Divide and conquer

- Split the n bits into two groups of $n/2$

$$\begin{array}{rccccc} & x_1 & & & x_0 & \\ \times & b_{n-1} b_{n-2} \cdots b_{\frac{n}{2}} & & b_{\frac{n}{2}-1} b_{\frac{n}{2}-2} \cdots b_0 & \\ & y_1 & & y_0 & \\ y & b'_{n-1} b'_{n-2} \cdots b'_{\frac{n}{2}} & & b'_{\frac{n}{2}-1} b'_{\frac{n}{2}-2} \cdots b'_0 & \end{array}$$

- Rewrite xy as $(x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0)$
- Regroup as $x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$
- Four $n/2$ -bit multiplications

- $T(1) = 1, T(n) = 4T(n/2) + n$
 - Combining the partial products requires adding $O(n)$ bit numbers
- $$\begin{aligned} T(n) &= 4T(n/2) + n \\ &= 4(4T(n/4) + n/2) + n \\ &= 4^2 T(n/2^2) + (2+1)n \\ &= 4^2(4T(n/2^3) + n/2^2) \\ &\quad +(2^1 + 2^0)n \\ &= 4^3 T(n/2^3) + (2^2 + 2^1 + 2^0)n \end{aligned}$$

Divide and conquer

- Split the n bits into two groups of $n/2$

$$\begin{array}{rccccc} & x_1 & & & x_0 & \\ \times & b_{n-1} b_{n-2} \cdots b_{\frac{n}{2}} & & b_{\frac{n}{2}-1} b_{\frac{n}{2}-2} \cdots b_0 & \\ & y_1 & & y_0 & \\ y & b'_{n-1} b'_{n-2} \cdots b'_{\frac{n}{2}} & & b'_{\frac{n}{2}-1} b'_{\frac{n}{2}-2} \cdots b'_0 & \end{array}$$

- Rewrite xy as $(x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0)$
- Regroup as $x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0$
- Four $n/2$ -bit multiplications

- $T(1) = 1, T(n) = 4T(n/2) + n$
 - Combining the partial products requires adding $O(n)$ bit numbers
- $$\begin{aligned} T(n) &= 4T(n/2) + n \\ &= 4(4T(n/4) + n/2) + n \\ &= 4^2 T(n/2^2) + (2+1)n \\ &= 4^2(4T(n/2^3) + n/2^2) \\ &\quad +(2^1 + 2^0)n \\ &= 4^3 T(n/2^3) + (2^2 + 2^1 + 2^0)n \\ &= \dots \\ &= 4^{\log n} T(n/2^{\log n}) \\ &\quad +(2^{\log n - 1} + \dots + 2^1 + 2^0)n \end{aligned}$$

Divide and conquer

- Split the n bits into two groups of $n/2$

$$\begin{array}{rccccc} & x_1 & & & x_0 & \\ \times & b_{n-1} b_{n-2} \cdots b_{\frac{n}{2}} & & b_{\frac{n}{2}-1} b_{\frac{n}{2}-2} \cdots b_0 & \\ & y_1 & & y_0 & \\ y & b'_{n-1} b'_{n-2} \cdots b'_{\frac{n}{2}} & & b'_{\frac{n}{2}-1} b'_{\frac{n}{2}-2} \cdots b'_0 & \end{array}$$

- Rewrite xy as $(x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0)$
- Regroup as $x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0$
- Four $n/2$ -bit multiplications

- $T(1) = 1, T(n) = 4T(n/2) + n$
 - Combining the partial products requires adding $O(n)$ bit numbers
- $$\begin{aligned} T(n) &= 4T(n/2) + n \\ &= 4(4T(n/4) + n/2) + n \\ &= 4^2 T(n/2^2) + (2+1)n \\ &= 4^2(4T(n/2^3) + n/2^2) \\ &\quad +(2^1 + 2^0)n \\ &= 4^3 T(n/2^3) + (2^2 + 2^1 + 2^0)n \\ &= \dots \\ &= 4^{\log n} T(n/2^{\log n}) \\ &\quad +(2^{\log n - 1} + \dots + 2^1 + 2^0)n \\ &= O(n^2) \end{aligned}$$

Karatsuba's algorithm

- Rewrite xy as

$$x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$$

- $T(n) = 4T(n/2) + n$ is $O(n^2)$

Karatsuba's algorithm

- Rewrite xy as

$$x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$$

- $T(n) = 4T(n/2) + n$ is $O(n^2)$

- Divide and conquer has not helped!

Karatsuba's algorithm

- Rewrite xy as

$$x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$$

- $T(n) = 4T(n/2) + n$ is $O(n^2)$

- Divide and conquer has not helped!

- $(x_1 - x_0)(y_1 - y_0) =$

$$x_1y_1 - x_1y_0 - x_0y_1 + x_0y_0$$

- $O(n/2)$ bit multiplication

Karatsuba's algorithm

- Rewrite xy as

$$x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$$

- $T(n) = 4T(n/2) + n$ is $O(n^2)$

- Divide and conquer has not helped!

- $(x_1 - x_0)(y_1 - y_0) =$

$$x_1y_1 - x_1y_0 - x_0y_1 + x_0y_0$$

- $O(n/2)$ bit multiplication

- Compute x_1y_1, x_0y_0

- $O(n/2)$ bit multiplications

Karatsuba's algorithm

- Rewrite xy as

$$x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$$

- $T(n) = 4T(n/2) + n$ is $O(n^2)$

- Divide and conquer has not helped!

- $(x_1 - x_0)(y_1 - y_0) =$

$$x_1y_1 - x_1y_0 - x_0y_1 + x_0y_0$$

- $O(n/2)$ bit multiplication

- Compute x_1y_1, x_0y_0

- $O(n/2)$ bit multiplications

- $(x_1y_1 + x_0y_0) - (x_1 - x_0)(y_1 - y_0)$

leaves $x_1y_0 + x_0y_1$

- 3 $O(n/2)$ bit multiplications

Karatsuba's algorithm

- Rewrite xy as

$$x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$$

- $T(n) = 4T(n/2) + n$ is $O(n^2)$

- Divide and conquer has not helped!

- $(x_1 - x_0)(y_1 - y_0) =$

$$x_1y_1 - x_1y_0 - x_0y_1 + x_0y_0$$

- $O(n/2)$ bit multiplication

- Compute x_1y_1, x_0y_0

- $O(n/2)$ bit multiplications

- $(x_1y_1 + x_0y_0) - (x_1 - x_0)(y_1 - y_0)$
leaves $x_1y_0 + x_0y_1$

- $3 O(n/2)$ bit multiplications

The Algorithm

```
Fast-Multiply(x, y, n)
```

```
    if n = 1  
        return x · y
```

```
    else  
        m = n/2  
        (x1, x0) = (x/2m, x mod 2m) Bit shifting  
        (y1, y0) = (y/2m, y mod 2m) Bit shifting  
        (a, b) = (x1 - x0, y1 - y0)
```

```
        p = Fast-Multiply(x1, y1, m)
```

```
        q = Fast-Multiply(x0, y0, m)
```

```
        r = Fast-Multiply(a, b, m)
```

```
        return p · 2n + (p + q - r) · 2n/2 + q
```

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$
- $T(n) = 3T(n/2) + n$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$
- $$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \end{aligned}$$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$
- $$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \\ &= 3^2(3T(n/2^3) + n/2^2) \\ &\quad + ((3/2)^1 + 1)n \\ &= 3^3 T(n/2^3) \\ &\quad + ((3/2)^2 + (3/2)^1 + 1)n \end{aligned}$$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$
- $$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \\ &= 3^2(3T(n/2^3) + n/2^2) \\ &\quad + ((3/2)^1 + 1)n \\ &= 3^3 T(n/2^3) \\ &\quad + ((3/2)^2 + (3/2)^1 + 1)n \\ &= \dots \\ &= 3^{\log n} T(n/2^{\log_2 n}) \\ &\quad + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n \end{aligned}$$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$
- $$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \\ &= 3^2(3T(n/2^3) + n/2^2) \\ &\quad + ((3/2)^1 + 1)n \\ &= 3^3 T(n/2^3) \\ &\quad + ((3/2)^2 + (3/2)^1 + 1)n \\ &= \dots \\ &= 3^{\log n} T(n/2^{\log_2 n}) \\ &\quad + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n \\ &= 3^{\log n} \\ &\quad + [((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n \end{aligned}$$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$
- $a^{\log n} = n^{\log a}$

$$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \\ &= 3^2(3T(n/2^3) + n/2^2) \\ &\quad + ((3/2)^1 + 1)n \\ &= 3^3 T(n/2^3) \\ &\quad + ((3/2)^2 + (3/2)^1 + 1)n \\ &= \dots \\ &= 3^{\log n} T(n/2^{\log_2 n}) \\ &\quad + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n \\ &= 3^{\log n} \\ &\quad + [((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n \end{aligned}$$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$

- $a^{\log n} = n^{\log a}$

- $$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \\ &= 3^2(3T(n/2^3) + n/2^2) \\ &\quad + ((3/2)^1 + 1)n \\ &= 3^3 T(n/2^3) \\ &\quad + ((3/2)^2 + (3/2)^1 + 1)n \\ &= \dots \\ &= 3^{\log n} T(n/2^{\log_2 n}) \\ &\quad + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n \\ &= 3^{\log n} \\ &\quad + [((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n \end{aligned}$$

- $3^{\log n} = n^{\log 3}$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$

- $$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \\ &= 3^2(3T(n/2^3) + n/2^2) \\ &\quad + ((3/2)^1 + 1)n \\ &= 3^3 T(n/2^3) \\ &\quad + ((3/2)^2 + (3/2)^1 + 1)n \\ &= \dots \\ &= 3^{\log n} T(n/2^{\log_2 n}) \\ &\quad + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n \\ &= 3^{\log n} \\ &\quad + [((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n \end{aligned}$$

- $a^{\log n} = n^{\log a}$

- $3^{\log n} = n^{\log 3}$

- $n \cdot (3/2)^{\log n}$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$

- $$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \\ &= 3^2(3T(n/2^3) + n/2^2) \\ &\quad + ((3/2)^1 + 1)n \\ &= 3^3 T(n/2^3) \\ &\quad + ((3/2)^2 + (3/2)^1 + 1)n \\ &= \dots \\ &= 3^{\log n} T(n/2^{\log_2 n}) \\ &\quad + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n \\ &= 3^{\log n} \\ &\quad + [((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n \end{aligned}$$

- $a^{\log n} = n^{\log a}$

- $3^{\log n} = n^{\log 3}$

- $n \cdot (3/2)^{\log n} = n \cdot n^{\log(3/2)}$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$

- $$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \\ &= 3^2(3T(n/2^3) + n/2^2) \\ &\quad + ((3/2)^1 + 1)n \\ &= 3^3 T(n/2^3) \\ &\quad + ((3/2)^2 + (3/2)^1 + 1)n \\ &= \dots \\ &= 3^{\log n} T(n/2^{\log_2 n}) \\ &\quad + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n \\ &= 3^{\log n} \\ &\quad + [((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n \end{aligned}$$

- $a^{\log n} = n^{\log a}$

- $3^{\log n} = n^{\log 3}$

- $$\begin{aligned} n \cdot (3/2)^{\log n} &= n \cdot n^{\log(3/2)} \\ &= n \cdot n^{\log 3 - \log 2} \end{aligned}$$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$

- $$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \\ &= 3^2(3T(n/2^3) + n/2^2) \\ &\quad + ((3/2)^1 + 1)n \\ &= 3^3 T(n/2^3) \\ &\quad + ((3/2)^2 + (3/2)^1 + 1)n \\ &= \dots \\ &= 3^{\log n} T(n/2^{\log_2 n}) \\ &\quad + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n \\ &= 3^{\log n} \\ &\quad + [((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n \end{aligned}$$

- $a^{\log n} = n^{\log a}$

- $3^{\log n} = n^{\log 3}$

- $$\begin{aligned} n \cdot (3/2)^{\log n} &= n \cdot n^{\log(3/2)} \\ &= n \cdot n^{\log 3 - \log 2} \\ &= n^1 \cdot n^{\log 3 - 1} \end{aligned}$$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$

- $$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3T(n/4) + n/2) + n \\ &= 3^2 T(n/2^2) + (3/2 + 1)n \\ &= 3^2(3T(n/2^3) + n/2^2) \\ &\quad + ((3/2)^1 + 1)n \\ &= 3^3 T(n/2^3) \\ &\quad + ((3/2)^2 + (3/2)^1 + 1)n \\ &= \dots \\ &= 3^{\log n} T(n/2^{\log_2 n}) \\ &\quad + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n \\ &= 3^{\log n} \\ &\quad + [((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n \end{aligned}$$

- $a^{\log n} = n^{\log a}$

- $3^{\log n} = n^{\log 3}$

- $$\begin{aligned} n \cdot (3/2)^{\log n} &= n \cdot n^{\log(3/2)} \\ &= n \cdot n^{\log 3 - \log 2} \\ &= n^1 \cdot n^{\log 3 - 1} \\ &= n^{1 + \log 3 - 1} \end{aligned}$$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$

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

$$= 3(3T(n/4) + n/2) + n$$

$$= 3^2 T(n/2^2) + (3/2 + 1)n$$

$$= 3^2(3T(n/2^3) + n/2^2) + ((3/2)^1 + 1)n$$

$$= 3^3 T(n/2^3) + ((3/2)^2 + (3/2)^1 + 1)n$$

$$= \dots$$

$$= 3^{\log n} T(n/2^{\log_2 n}) + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n$$

$$= 3^{\log n} + [((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n$$

- $a^{\log n} = n^{\log a}$

- $3^{\log n} = n^{\log 3}$

- $n \cdot (3/2)^{\log n} = n \cdot n^{\log(3/2)}$
 $= n \cdot n^{\log 3 - \log 2}$

$$= n^1 \cdot n^{\log 3 - 1}$$

$$= n^{1 + \log 3 - 1}$$

$$= n^{\log 3}$$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$

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

$$= 3(3T(n/4) + n/2) + n$$

$$= 3^2 T(n/2^2) + (3/2 + 1)n$$

$$= 3^2(3T(n/2^3) + n/2^2) + ((3/2)^1 + 1)n$$

$$= 3^3 T(n/2^3) + ((3/2)^2 + (3/2)^1 + 1)n$$

$$= \dots$$

$$= 3^{\log n} T(n/2^{\log_2 n}) + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n$$

$$= 3^{\log n} + [((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n$$

- $a^{\log n} = n^{\log a}$

- $3^{\log n} = n^{\log 3}$

- $n \cdot (3/2)^{\log n} = n \cdot n^{\log(3/2)}$
 $= n \cdot n^{\log 3 - \log 2}$

$$= n^1 \cdot n^{\log 3 - 1}$$

$$= n^{1 + \log 3 - 1}$$

$$= n^{\log 3}$$

- $\log 3 \approx 1.59$

Karatsuba's algorithm — Analysis

- $T(1) = 1, T(n) = 3T(n/2) + n$

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

$$= 3(3T(n/4) + n/2) + n$$

$$= 3^2 T(n/2^2) + (3/2 + 1)n$$

$$= 3^2(3T(n/2^3) + n/2^2) + ((3/2)^1 + 1)n$$

$$= 3^3 T(n/2^3) + ((3/2)^2 + (3/2)^1 + 1)n$$

$$= \dots$$

$$= 3^{\log n} T(n/2^{\log_2 n}) + ((3/2)^{\log n - 1} + \dots + (3/2)^1 + 1)n$$

$$= 3^{\log n} + [((3/2)^{\log n - 1} - 1)/((3/2) - 1)]n$$

- $a^{\log n} = n^{\log a}$

- $3^{\log n} = n^{\log 3}$

- $n \cdot (3/2)^{\log n} = n \cdot n^{\log(3/2)}$

$$= n \cdot n^{\log 3 - \log 2}$$

$$= n^1 \cdot n^{\log 3 - 1}$$

$$= n^{1+\log 3 - 1}$$

$$= n^{\log 3}$$

- $\log 3 \approx 1.59$

- Divide and conquer reduces the complexity of integer multiplication from $O(n^2)$ to $O(n^{1.59})$

Historical note

- In the 1950's, Andrei Kolmogorov, one of the giants of 20th century mathematics, publicly conjectured that multiplication could not be done in subquadratic time

Historical note

- In the 1950's, Andrei Kolmogorov, one of the giants of 20th century mathematics, publicly conjectured that multiplication could not be done in subquadratic time
- Kolmogorov mentioned this conjecture at a seminar in Moscow University in 1960

Historical note

- In the 1950's, Andrei Kolmogorov, one of the giants of 20th century mathematics, publicly conjectured that multiplication could not be done in subquadratic time
- Kolmogorov mentioned this conjecture at a seminar in Moscow University in 1960
- Anatolii Karatsuba, a 23 year old student, came back 2 weeks later to Kolmogorov with this divide and conquer algorithm!

Historical note

- In the 1950's, Andrei Kolmogorov, one of the giants of 20th century mathematics, publicly conjectured that multiplication could not be done in subquadratic time
- Kolmogorov mentioned this conjecture at a seminar in Moscow University in 1960
- Anatolii Karatsuba, a 23 year old student, came back 2 weeks later to Kolmogorov with this divide and conquer algorithm!
- Karatsuba's original proposal was slightly different
 - Instead of $r = (x_1 - x_0)(y_1 - y_0)$, he used $r = (x_1 + x_0)(y_1 + y_0)$
 - Then, $x_0y_1 + x_1y_0 = r - (x_1y_1 + x_0y_0)$
 - Difficulty is that $x_1 + x_0$, $y_1 + y_0$ could have $n + 1$ bits, complicates the analysis

Historical note

- In the 1950's, Andrei Kolmogorov, one of the giants of 20th century mathematics, publicly conjectured that multiplication could not be done in subquadratic time
- Kolmogorov mentioned this conjecture at a seminar in Moscow University in 1960
- Anatolii Karatsuba, a 23 year old student, came back 2 weeks later to Kolmogorov with this divide and conquer algorithm!
- Karatsuba's original proposal was slightly different
 - Instead of $r = (x_1 - x_0)(y_1 - y_0)$, he used $r = (x_1 + x_0)(y_1 + y_0)$
 - Then, $x_0y_1 + x_1y_0 = r - (x_1y_1 + x_0y_0)$
 - Difficulty is that $x_1 + x_0$, $y_1 + y_0$ could have $n + 1$ bits, complicates the analysis
- Using $r = (x_1 - x_0)(y_1 - y_0)$ to simplify the analysis is due to Donald Knuth

Historical note

- In the 1950's, Andrei Kolmogorov, one of the giants of 20th century mathematics, publicly conjectured that multiplication could not be done in subquadratic time
- Kolmogorov mentioned this conjecture at a seminar in Moscow University in 1960
- Anatolii Karatsuba, a 23 year old student, came back 2 weeks later to Kolmogorov with this divide and conquer algorithm!
- Karatsuba's original proposal was slightly different
 - Instead of $r = (x_1 - x_0)(y_1 - y_0)$, he used $r = (x_1 + x_0)(y_1 + y_0)$
 - Then, $x_0y_1 + x_1y_0 = r - (x_1y_1 + x_0y_0)$
 - Difficulty is that $x_1 + x_0$, $y_1 + y_0$ could have $n+1$ bits, complicates the analysis
- Using $r = (x_1 - x_0)(y_1 - y_0)$ to simplify the analysis is due to Donald Knuth
- Karatsuba's algorithm can be used in any base, not just for binary multiplication

Divide and Conquer: Recursion Trees

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 8

Solving recurrences

- Divide and conquer involves breaking up a problem into disjoint subproblems and combining the solutions efficiently

Solving recurrences

- Divide and conquer involves breaking up a problem into disjoint subproblems and combining the solutions efficiently
- Complexity $T(n)$ is expressed as a recurrence

Solving recurrences

- Divide and conquer involves breaking up a problem into disjoint subproblems and combining the solutions efficiently
- Complexity $T(n)$ is expressed as a recurrence
- For searching and sorting, we solved simple recurrences by repeated substitution
 - Binary search: $T(n) = T(n/2) + 1$, $T(n)$ is $O(\log n)$
 - Merge sort: $T(n) = 2T(n/2) + n$, $T(n)$ is $O(n \log n)$

Solving recurrences

- Divide and conquer involves breaking up a problem into disjoint subproblems and combining the solutions efficiently
- Complexity $T(n)$ is expressed as a recurrence
- For searching and sorting, we solved simple recurrences by repeated substitution
 - Binary search: $T(n) = T(n/2) + 1$, $T(n)$ is $O(\log n)$
 - Merge sort: $T(n) = 2T(n/2) + n$, $T(n)$ is $O(n \log n)$
- For integer multiplication, the analysis became more complicated
 - Naive divide and conquer: $T(n) = 4T(n/2) + n$, $T(n)$ is $O(n^2)$
 - Karatsuba's algorithm: $T(n) = 3T(n/2) + n$, $T(n)$ is $O(n^{\log 3})$

Solving recurrences

- Divide and conquer involves breaking up a problem into disjoint subproblems and combining the solutions efficiently
- Complexity $T(n)$ is expressed as a recurrence
- For searching and sorting, we solved simple recurrences by repeated substitution
 - Binary search: $T(n) = T(n/2) + 1$, $T(n)$ is $O(\log n)$
 - Merge sort: $T(n) = 2T(n/2) + n$, $T(n)$ is $O(n \log n)$
- For integer multiplication, the analysis became more complicated
 - Naive divide and conquer: $T(n) = 4T(n/2) + n$, $T(n)$ is $O(n^2)$
 - Karatsuba's algorithm: $T(n) = 3T(n/2) + n$, $T(n)$ is $O(n^{\log 3})$
- Is there a uniform way to compute the asymptotic expression for $T(n)$?

Recursion trees

- **Recursion tree** Rooted tree with one node for each recursive subproblem

Recursion trees

- **Recursion tree** Rooted tree with one node for each recursive subproblem
- **Value** of each node is time spent on that subproblem **excluding** recursive calls

Recursion trees

- **Recursion tree** Rooted tree with one node for each recursive subproblem
- **Value** of each node is time spent on that subproblem **excluding** recursive calls
- Concretely, on an input of size n

Recursion trees

- **Recursion tree** Rooted tree with one node for each recursive subproblem
- **Value** of each node is time spent on that subproblem **excluding** recursive calls
- Concretely, on an input of size n
 - $f(n)$ is the time spent on non-recursive work

Recursion trees

- **Recursion tree** Rooted tree with one node for each recursive subproblem
- **Value** of each node is time spent on that subproblem **excluding** recursive calls
- Concretely, on an input of size n
 - $f(n)$ is the time spent on non-recursive work
 - r is the number of recursive calls

Recursion trees

- **Recursion tree** Rooted tree with one node for each recursive subproblem
- **Value** of each node is time spent on that subproblem **excluding** recursive calls
- Concretely, on an input of size n
 - $f(n)$ is the time spent on non-recursive work
 - r is the number of recursive calls
 - Each recursive call works on a subproblem of size n/c

Recursion trees

- **Recursion tree** Rooted tree with one node for each recursive subproblem
- **Value** of each node is time spent on that subproblem **excluding** recursive calls
- Concretely, on an input of size n
 - $f(n)$ is the time spent on non-recursive work
 - r is the number of recursive calls
 - Each recursive call works on a subproblem of size n/c
- Resulting recurrence: $T(n) = rT(n/c) + f(n)$

Recursion trees

- **Recursion tree** Rooted tree with one node for each recursive subproblem
- **Value** of each node is time spent on that subproblem **excluding** recursive calls
- Concretely, on an input of size n
 - $f(n)$ is the time spent on non-recursive work
 - r is the number of recursive calls
 - Each recursive call works on a subproblem of size n/c
- Resulting recurrence: $T(n) = rT(n/c) + f(n)$
- Root of recursion tree for $T(n)$ has value $f(n)$

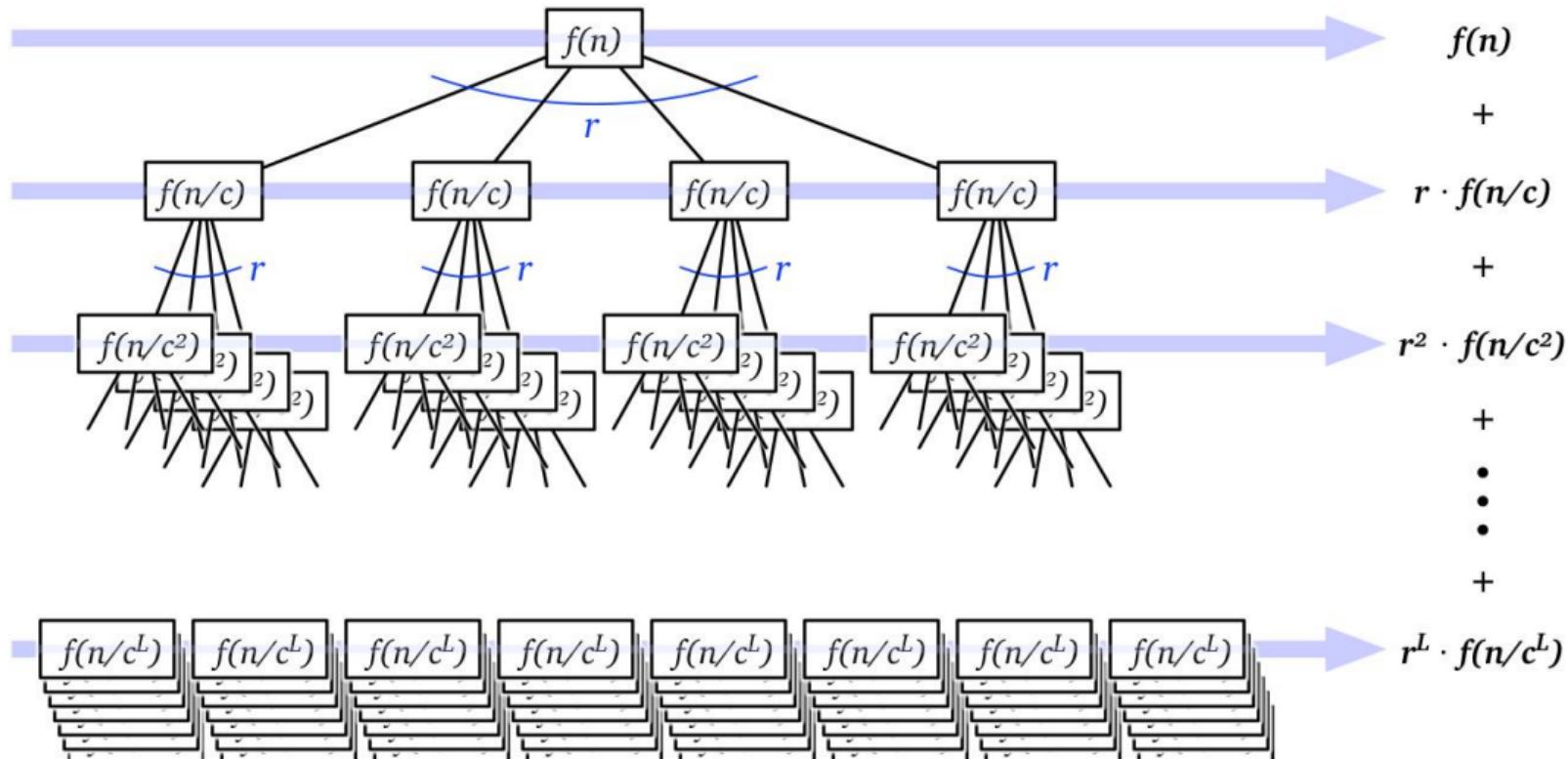
Recursion trees

- **Recursion tree** Rooted tree with one node for each recursive subproblem
- **Value** of each node is time spent on that subproblem **excluding** recursive calls
- Concretely, on an input of size n
 - $f(n)$ is the time spent on non-recursive work
 - r is the number of recursive calls
 - Each recursive call works on a subproblem of size n/c
- Resulting recurrence: $T(n) = rT(n/c) + f(n)$
- Root of recursion tree for $T(n)$ has value $f(n)$
- Root has r children, each (recursively) the root of a tree for $T(n/c)$

Recursion trees

- **Recursion tree** Rooted tree with one node for each recursive subproblem
- **Value** of each node is time spent on that subproblem **excluding** recursive calls
- Concretely, on an input of size n
 - $f(n)$ is the time spent on non-recursive work
 - r is the number of recursive calls
 - Each recursive call works on a subproblem of size n/c
- Resulting recurrence: $T(n) = rT(n/c) + f(n)$
- Root of recursion tree for $T(n)$ has value $f(n)$
- Root has r children, each (recursively) the root of a tree for $T(n/c)$
- Each node at level d has value $f(n/c^d)$
 - Assume, for simplicity, that n was a power of c

Recursion tree for $T(n) = rT(n/c) + f(n)$



Recursion trees

- Leaves correspond to the base case $T(1)$
 - Safe to assume $T(1) = 1$, asymptotic complexity ignores constants

Recursion trees

- Leaves correspond to the base case $T(1)$
 - Safe to assume $T(1) = 1$, asymptotic complexity ignores constants
- Level i has r^i nodes, each with value $f(n/c^i)$

Recursion trees

- Leaves correspond to the base case $T(1)$
 - Safe to assume $T(1) = 1$, asymptotic complexity ignores constants
- Level i has r^i nodes, each with value $f(n/c^i)$
- Tree has L levels, $L = \log_c n$

Recursion trees

- Leaves correspond to the base case $T(1)$
 - Safe to assume $T(1) = 1$, asymptotic complexity ignores constants
- Level i has r^i nodes, each with value $f(n/c^i)$
- Tree has L levels, $L = \log_c n$
- Total cost is $T(n) = \sum_{i=0}^L r^i \cdot f(n/c^i)$

Recursion trees

- Leaves correspond to the base case $T(1)$
 - Safe to assume $T(1) = 1$, asymptotic complexity ignores constants
- Level i has r^i nodes, each with value $f(n/c^i)$
- Tree has L levels, $L = \log_c n$
- Total cost is $T(n) = \sum_{i=0}^L r^i \cdot f(n/c^i)$
- Number of leaves is r^L
 - Last term in the level by level sum is $r^L \cdot f(1) = r^{\log_c n} \cdot 1 = n^{\log_c r}$
 - Recall that $a^{\log_b c} = c^{\log_b a}$

Recursion trees

- Tree has $\log_c n$ levels, last level has cost is $n^{\log_c r}$

- Total cost is $T(n) = \sum_{i=0}^L r^i \cdot f(n/c^i)$

Recursion trees

- Tree has $\log_c n$ levels, last level has cost is $n^{\log_c r}$
- Total cost is $T(n) = \sum_{i=0}^L r^i \cdot f(n/c^i)$
- Think of the total cost as a series. Three common cases

Recursion trees

- Tree has $\log_c n$ levels, last level has cost is $n^{\log_c r}$
- Total cost is $T(n) = \sum_{i=0}^L r^i \cdot f(n/c^i)$
- Think of the total cost as a series. Three common cases
- **Decreasing** Each term is a constant factor smaller than previous term
 - Root dominates the sum, $T(n) = O(f(n))$

Recursion trees

- Tree has $\log_c n$ levels, last level has cost is $n^{\log_c r}$
- Total cost is $T(n) = \sum_{i=0}^L r^i \cdot f(n/c^i)$
- Think of the total cost as a series. Three common cases
- **Decreasing** Each term is a constant factor smaller than previous term
 - Root dominates the sum, $T(n) = O(f(n))$
- **Equal** All terms in the series are equal
 - $T(n) = O(f(n) \cdot L) = O(f(n) \log n)$ — $\log_c n$ is asymptotically same as $\log n$

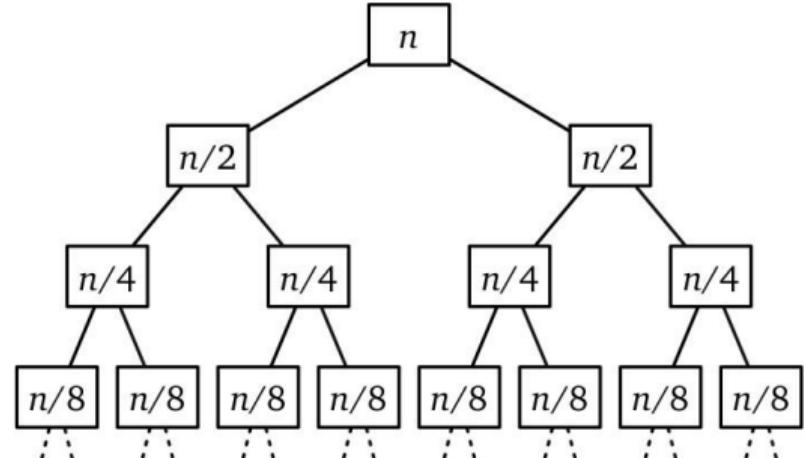
Recursion trees

- Tree has $\log_c n$ levels, last level has cost is $n^{\log_c r}$
- Total cost is $T(n) = \sum_{i=0}^L r^i \cdot f(n/c^i)$
- Think of the total cost as a series. Three common cases
- **Decreasing** Each term is a constant factor smaller than previous term
 - Root dominates the sum, $T(n) = O(f(n))$
- **Equal** All terms in the series are equal
 - $T(n) = O(f(n) \cdot L) = O(f(n) \log n)$ — $\log_c n$ is asymptotically same as $\log n$
- **Increasing** Series grows exponentially, each term a constant factor larger than previous term
 - Leaves dominate the sum, $T(n) = O(n^{\log_c r})$

Examples

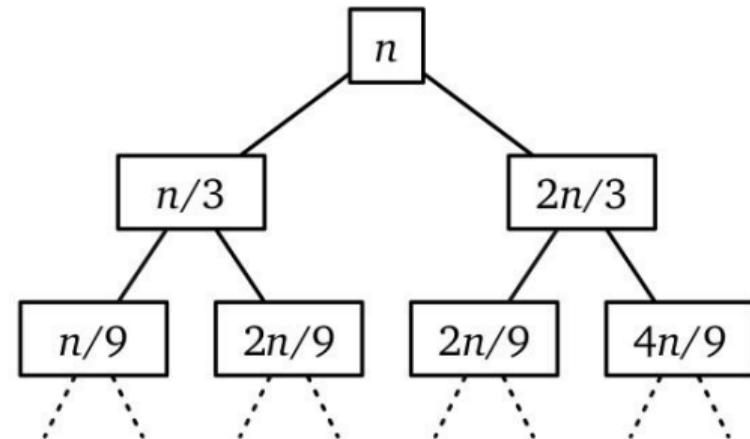
- Merge sort

- $T(n) = 2T(n/2) + n$
- Series is equal, $T(n)$ is $O(n \log n)$



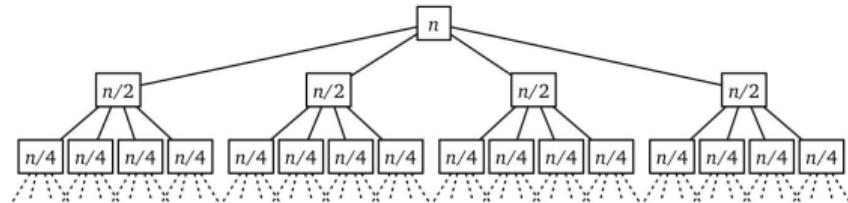
Examples

- Merge sort
 - $T(n) = 2T(n/2) + n$
 - Series is equal, $T(n)$ is $O(n \log n)$
- Quick sort with pivot always in the middle third of values
 - $T(n) = T(n/3) + T(2n/3) + n$
 - Unequal partitions, allow “holes”
 - Depth is $\log_{\frac{3}{2}} n = O(\log n)$
 - Series is equal, $T(n)$ is $O(n \log n)$



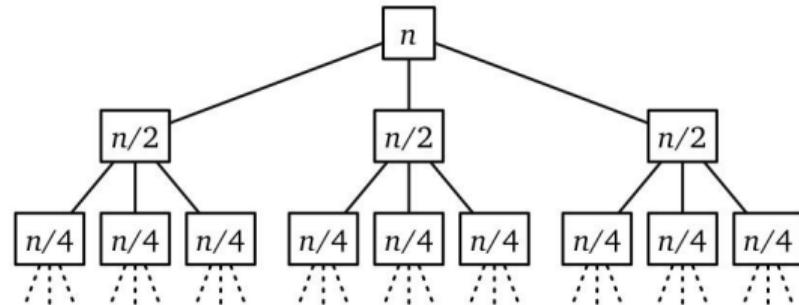
Examples

- Merge sort
 - $T(n) = 2T(n/2) + n$
 - Series is equal, $T(n)$ is $O(n \log n)$
- Quick sort with pivot always in the middle third of values
 - $T(n) = T(n/3) + T(2n/3) + n$
 - Unequal partitions, allow “holes”
 - Depth is $\log_{\frac{3}{2}} n = O(\log n)$
 - Series is equal, $T(n)$ is $O(n \log n)$
- Naive integer multiplication, exponential, $T(n) = n^{\log_2 4} = n^2$



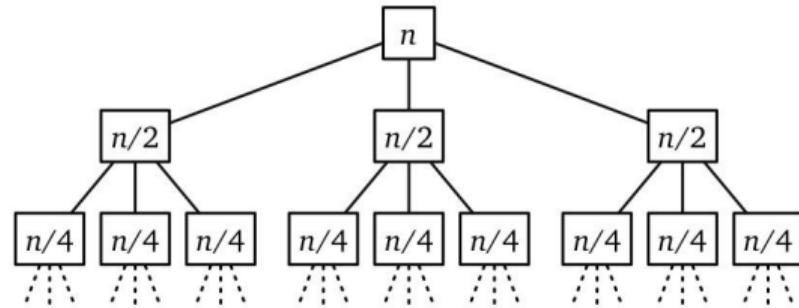
Examples

- Merge sort
 - $T(n) = 2T(n/2) + n$
 - Series is equal, $T(n)$ is $O(n \log n)$
- Quick sort with pivot always in the middle third of values
 - $T(n) = T(n/3) + T(2n/3) + n$
 - Unequal partitions, allow “holes”
 - Depth is $\log_{\frac{3}{2}} n = O(\log n)$
 - Series is equal, $T(n)$ is $O(n \log n)$
- Naive integer multiplication, exponential, $T(n) = n^{\log_2 4} = n^2$
- Karatsuba, exponential, $T(n) = n^{\log_2 3}$



Examples

- Merge sort
 - $T(n) = 2T(n/2) + n$
 - Series is equal, $T(n)$ is $O(n \log n)$
- Quick sort with pivot always in the middle third of values
 - $T(n) = T(n/3) + T(2n/3) + n$
 - Unequal partitions, allow “holes”
 - Depth is $\log_{\frac{3}{2}} n = O(\log n)$
 - Series is equal, $T(n)$ is $O(n \log n)$
- Naive integer multiplication, exponential, $T(n) = n^{\log_2 4} = n^2$
- Karatsuba, exponential, $T(n) = n^{\log_2 3}$



■ Acknowledgment

Illustrations from
Algorithms by Jeff Erickson,
<https://jeffe.cs.illinois.edu/teaching/algorithms/>

Divide and Conquer: Quick Select

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 8

Selection

- Find the k^{th} largest value in a sequence of length n

Selection

- Find the k^{th} largest value in a sequence of length n
- Sort in descending order and look at position $k — O(n \log n)$

Selection

- Find the k^{th} largest value in a sequence of length n
- Sort in descending order and look at position k — $O(n \log n)$
- Can we do better?
 - $k = 1$ — maximum, $O(n)$
 - $k = n$ — minimum, $O(n)$

Selection

- Find the k^{th} largest value in a sequence of length n
- Sort in descending order and look at position k — $O(n \log n)$
- Can we do better?
 - $k = 1$ — maximum, $O(n)$
 - $k = n$ — minimum, $O(n)$
- For any fixed k , k passes, $O(kn)$

Selection

- Find the k^{th} largest value in a sequence of length n
- Sort in descending order and look at position k — $O(n \log n)$
- Can we do better?
 - $k = 1$ — maximum, $O(n)$
 - $k = n$ — minimum, $O(n)$
- For any fixed k , k passes, $O(kn)$
- Median — $k = n/2$
 - If we can find median in $O(n)$, quicksort becomes $O(n \log n)$

Divide and conquer

- Recall partitioning for quicksort
 - Pivot partitions sequence as `lower` and `upper`

Divide and conquer

- Recall partitioning for quicksort
 - Pivot partitions sequence as `lower` and `upper`
- Let `m = len(lower)`. 3 cases:
 - `k <= m` — answer lies in `lower`
 - `k == m+1` — answer is `pivot`
 - `k > m+1` — answer lies in `upper`

Divide and conquer

- Recall partitioning for quicksort
 - Pivot partitions sequence as `lower` and `upper`
- Let `m = len(lower)`. 3 cases:
 - `k <= m` — answer lies in `lower`
 - `k == m+1` — answer is `pivot`
 - `k > m+1` — answer lies in `upper`
- Recursive strategy
 - Case 1: `select(lower,k)`
 - Case 2: `return(pivot)`
 - Case 3: `select(upper,k-(m+1))`

Divide and conquer

- Recall partitioning for quicksort
 - Pivot partitions sequence as `lower` and `upper`
- Let `m = len(lower)`. 3 cases:
 - `k <= m` — answer lies in `lower`
 - `k == m+1` — answer is `pivot`
 - `k > m+1` — answer lies in `upper`
- Recursive strategy
 - Case 1: `select(lower,k)`
 - Case 2: `return(pivot)`
 - Case 3: `select(upper,k-(m+1))`

```
def quickselect(L,l,r,k): # k-th largest in L[l:r]
    if (k < 1) or (k > r-1):
        return(None)

    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper + 1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            (lower,upper) = (lower+1,upper+1)
    (L[1],L[lower-1]) = (L[lower-1],L[1]) # Move pivot
    lower = lower - 1

    # Recursive calls
    lowerlen = lower - 1
    if k <= lowerlen:
        return(quickselect(L,l,lower,k))
    elif k == (lowerlen + 1):
        return(L[lower])
    else:
        return(quickselect(L,lower+1,r,k-(lowerlen+1)))
```

Analysis

- Recurrence is similar to quicksort

```
def quickselect(L,l,r,k): # k-th largest in L[l:r]
    if (k < 1) or (k > r-1):
        return(None)

    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper + 1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            (lower,upper) = (lower+1,upper+1)
    (L[1],L[lower-1]) = (L[lower-1],L[1]) # Move pivot
    lower = lower - 1

    # Recursive calls
    lowerlen = lower - 1
    if k <= lowerlen:
        return(quickselect(L,l,lower,k))
    elif k == (lowerlen + 1):
        return(L[lower])
    else:
        return(quickselect(L,lower+1,r,k-(lowerlen+1)))
```

Analysis

- Recurrence is similar to quicksort

- $T(1) = 1$

$$T(n) = \max(T(m), T(n - (m+1))) + n,$$

where $m = \text{len(lower)}$

```
def quickselect(L,l,r,k): # k-th largest in L[l:r]
    if (k < 1) or (k > r-1):
        return(None)

    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper + 1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            (lower,upper) = (lower+1,upper+1)
    (L[1],L[lower-1]) = (L[lower-1],L[1]) # Move pivot
    lower = lower - 1

    # Recursive calls
    lowerlen = lower - 1
    if k <= lowerlen:
        return(quickselect(L,l,lower,k))
    elif k == (lowerlen + 1):
        return(L[lower])
    else:
        return(quickselect(L,lower+1,r,k-(lowerlen+1)))
```

Analysis

- Recurrence is similar to quicksort
- $T(1) = 1$
- $T(n) = \max(T(m), T(n - (m+1))) + n$, where $m = \text{len(lower)}$
- Worst case: m is always 0 or $n-1$
 - $T(n) = T(n - 1) + n$
 - $T(n)$ is $O(n^2)$

```
def quickselect(L,l,r,k): # k-th largest in L[l:r]
    if (k < 1) or (k > r-1):
        return(None)

    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper + 1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            (lower,upper) = (lower+1,upper+1)
    (L[1],L[lower-1]) = (L[lower-1],L[1]) # Move pivot
    lower = lower - 1

    # Recursive calls
    lowerlen = lower - 1
    if k <= lowerlen:
        return(quickselect(L,l,lower,k))
    elif k == (lowerlen + 1):
        return(L[lower])
    else:
        return(quickselect(L,lower+1,r,k-(lowerlen+1)))
```

Analysis

- Recurrence is similar to quicksort
- $T(1) = 1$
 $T(n) = \max(T(m), T(n - (m+1))) + n$,
where $m = \text{len(lower)}$
- Worst case: m is always 0 or $n-1$
 - $T(n) = T(n-1) + n$
 - $T(n)$ is $O(n^2)$
- Recall: if pivot is within a fixed fraction, quicksort is $O(n \log n)$
 - E.g., pivot in middle third of values
 - $T(n) = T(n/3) + T(2n/3) + n$

```
def quickselect(L,l,r,k): # k-th largest in L[l:r]
    if (k < 1) or (k > r-1):
        return(None)

    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper + 1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            (lower,upper) = (lower+1,upper+1)
    (L[1],L[lower-1]) = (L[lower-1],L[1]) # Move pivot
    lower = lower - 1

    # Recursive calls
    lowerlen = lower - 1
    if k <= lowerlen:
        return(quickselect(L,l,lower,k))
    elif k == (lowerlen + 1):
        return(L[lower])
    else:
        return(quickselect(L,lower+1,r,k-(lowerlen+1)))
```

Analysis

- Recurrence is similar to quicksort
- $T(1) = 1$
 $T(n) = \max(T(m), T(n - (m+1))) + n$,
where $m = \text{len(lower)}$
- Worst case: m is always 0 or $n-1$
 - $T(n) = T(n-1) + n$
 - $T(n)$ is $O(n^2)$
- Recall: if pivot is within a fixed fraction, quicksort is $O(n \log n)$
 - E.g., pivot in middle third of values
 - $T(n) = T(n/3) + T(2n/3) + n$
- Can we find a good pivot quickly?

```
def quickselect(L,l,r,k): # k-th largest in L[l:r]
    if (k < 1) or (k > r-1):
        return(None)

    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper + 1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            (lower,upper) = (lower+1,upper+1)
    (L[1],L[lower-1]) = (L[lower-1],L[1]) # Move pivot
    lower = lower - 1

    # Recursive calls
    lowerlen = lower - 1
    if k <= lowerlen:
        return(quickselect(L,l,lower,k))
    elif k == (lowerlen + 1):
        return(L[lower])
    else:
        return(quickselect(L,lower+1,r,k-(lowerlen+1)))
```

Median of medians

- Divide L into blocks of 5

Median of medians

- Divide L into blocks of 5
- Find the median of each block (brute force)

Median of medians

- Divide L into blocks of 5
- Find the median of each block (brute force)
- Let M be the list of block medians

Median of medians

- Divide L into blocks of 5
- Find the median of each block (brute force)
- Let M be the list of block medians
- Recursively apply the process to M

Median of medians

- Divide L into blocks of 5
- Find the median of each block (brute force)
- Let M be the list of block medians
- Recursively apply the process to M

```
def MoM(L): # Median of medians  
  
    if len(L) <= 5:  
        L.sort()  
        return(L[len(L)//2])  
  
    # Construct list of block medians  
    M = []  
  
    for i in range(0,len(L),5):  
        X = L[i:i+5]  
        X.sort()  
        M.append(X[len(X)//2])  
  
    return(MoM(M))
```

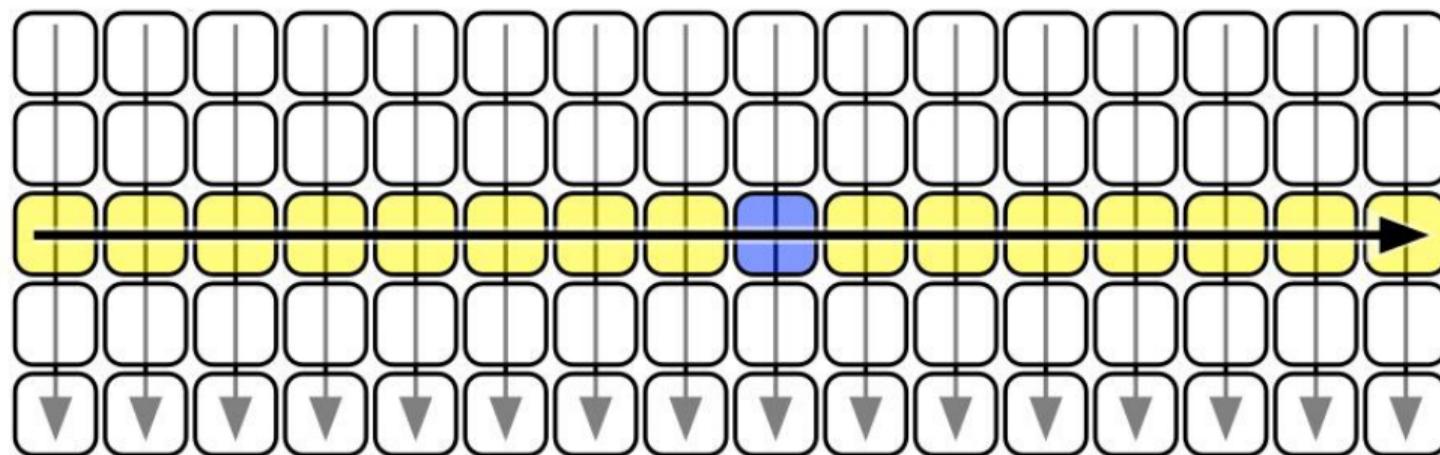
Median of medians

- Divide L into blocks of 5
- Find the median of each block (brute force)
- Let M be the list of block medians
- Recursively apply the process to M
- What can we guarantee about $\text{MoM}(L)$?

```
def MoM(L): # Median of medians  
  
    if len(L) <= 5:  
        L.sort()  
        return(L[len(L)//2])  
  
    # Construct list of block medians  
    M = []  
  
    for i in range(0,len(L),5):  
        X = L[i:i+5]  
        X.sort()  
        M.append(X[len(X)//2])  
  
    return(MoM(M))
```

Median of medians

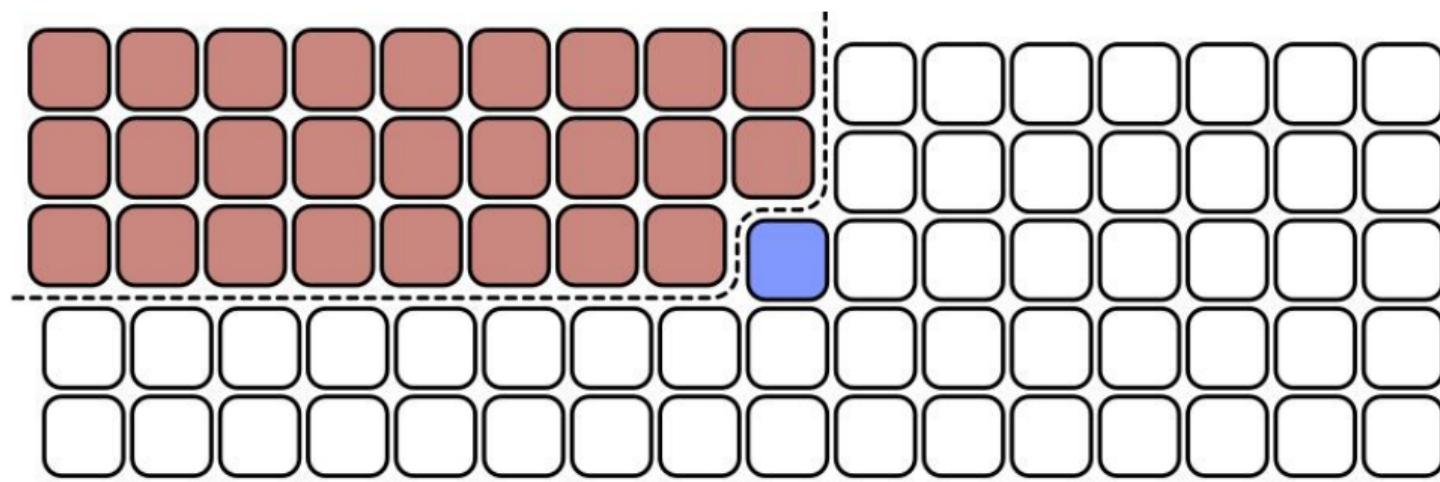
- We can visualize the blocks as follows



- Each block of 5 is arranged in ascending order, top to bottom
- Block medians are the middle row

Median of medians

- We can visualize the blocks as follows
- Each block of 5 is arranged in ascending order, top to bottom
- Block medians are the middle row
- The median of block medians lies between $3\lfloor \log(L) \rfloor / 10$ and $7\lfloor \log(L) \rfloor / 10$



Analysis

- Use median of block medians to locate pivot for `quickselect`

Analysis

- Use median of block medians to locate pivot for `quickselect`

```
def fastselect(L,l,r,k): # k-th largest in L[l:r]
    if (k < 1) or (k > r-1):
        return(None)

    # Find MoM pivot and move to L[1]
    pivot = MoM(L[l:r])
    pivotpos = min([i for i in range(l,r) if L[i] == pivot])
    (L[1],L[pivotpos]) = (L[pivotpos],L[1])

    # Partition as before
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        ...
        ...

    # Recursive calls
    lowerlen = lower - 1
    if k <= lowerlen:
        return(fastselect(L,1,lower,k))
    elif k == (lowerlen + 1):
        return(L[lower])
    else:
        return(fastselect(L,lower+1,r,k-(lowerlen+1)))
```

Analysis

- Use median of block medians to locate pivot for `quickselect`
- MoM is $O(n)$

$$T(1) = 1$$

$$T(n) = T(n/5) + n,$$

```
def fastselect(L,l,r,k): # k-th largest in L[l:r]
    if (k < 1) or (k > r-1):
        return(None)

    # Find MoM pivot and move to L[1]
    pivot = MoM(L[l:r])
    pivotpos = min([i for i in range(l,r) if L[i] == pivot])
    (L[1],L[pivotpos]) = (L[pivotpos],L[1])

    # Partition as before
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        ...
        ...

    # Recursive calls
    lowerlen = lower - 1
    if k <= lowerlen:
        return(fastselect(L,1,lower,k))
    elif k == (lowerlen + 1):
        return(L[lower])
    else:
        return(fastselect(L,lower+1,r,k-(lowerlen+1)))
```

Analysis

- Use median of block medians to locate pivot for `quickselect`

- MoM is $O(n)$

$$T(1) = 1$$

$$T(n) = T(n/5) + n,$$

- Recurrence for `fastselect` is now

$$T(1) = 1$$

$$T(n) = \max(T(3m/10), T(7m/10) + n,$$

where $m = \text{len(lower)}$

```
def fastselect(L,l,r,k): # k-th largest in L[l:r]
    if (k < 1) or (k > r-1):
        return(None)

    # Find MoM pivot and move to L[1]
    pivot = MoM(L[l:r])
    pivotpos = min([i for i in range(l,r) if L[i] == pivot])
    (L[1],L[pivotpos]) = (L[pivotpos],L[1])

    # Partition as before
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        ...
        ...

    # Recursive calls
    lowerlen = lower - 1
    if k <= lowerlen:
        return(fastselect(L,1,lower,k))
    elif k == (lowerlen + 1):
        return(L[lower])
    else:
        return(fastselect(L,lower+1,r,k-(lowerlen+1)))
```

Analysis

- Use median of block medians to locate pivot for `quickselect`

- MoM is $O(n)$

$$T(1) = 1$$

$$T(n) = T(n/5) + n,$$

- Recurrence for `fastselect` is now

$$T(1) = 1$$

$$T(n) = \max(T(3m/10), T(7m/10) + n,$$

where $m = \text{len(lower)}$

- $T(n)$ is $O(n)$

```
def fastselect(L,l,r,k): # k-th largest in L[l:r]
    if (k < 1) or (k > r-1):
        return(None)

    # Find MoM pivot and move to L[1]
    pivot = MoM(L[l:r])
    pivotpos = min([i for i in range(l,r) if L[i] == pivot])
    (L[1],L[pivotpos]) = (L[pivotpos],L[1])

    # Partition as before
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        ...
        ...

    # Recursive calls
    lowerlen = lower - 1
    if k <= lowerlen:
        return(fastselect(L,1,lower,k))
    elif k == (lowerlen + 1):
        return(L[lower])
    else:
        return(fastselect(L,lower+1,r,k-(lowerlen+1)))
```

Analysis

- Use median of block medians to locate pivot for `quickselect`

- MoM is $O(n)$

$$T(1) = 1$$

$$T(n) = T(n/5) + n,$$

- Recurrence for `fastselect` is now

$$T(1) = 1$$

$$T(n) = \max(T(3m/10), T(7m/10) + n,$$

where $m = \text{len(lower)}$

- $T(n)$ is $O(n)$

- Can also use MoM to make quicksort $O(n \log n)$

```
def fastselect(L,l,r,k): # k-th largest in L[l:r]
    if (k < 1) or (k > r-1):
        return(None)

    # Find MoM pivot and move to L[1]
    pivot = MoM(L[l:r])
    pivotpos = min([i for i in range(l,r) if L[i] == pivot])
    (L[1],L[pivotpos]) = (L[pivotpos],L[1])

    # Partition as before
    (pivot,lower,upper) = (L[1],l+1,l+1)
    for i in range(l+1,r):
        ...
        ...

    # Recursive calls
    lowerlen = lower - 1
    if k <= lowerlen:
        return(fastselect(L,1,lower,k))
    elif k == (lowerlen + 1):
        return(L[lower])
    else:
        return(fastselect(L,lower+1,r,k-(lowerlen+1)))
```

Summary

- Median of block medians helps find a good pivot in $O(n)$

Summary

- Median of block medians helps find a good pivot in $O(n)$
- Selection becomes $O(n)$, quicksort becomes $O(n \log n)$

Summary

- Median of block medians helps find a good pivot in $O(n)$
- Selection becomes $O(n)$, quicksort becomes $O(n \log n)$
- Notice that `fastselect` with `k = len(L)/2` finds median in time $O(n)$

Summary

- Median of block medians helps find a good pivot in $O(n)$
- Selection becomes $O(n)$, quicksort becomes $O(n \log n)$
- Notice that `fastselect` with `k = len(L)/2` finds median in time $O(n)$

Historical note

Summary

- Median of block medians helps find a good pivot in $O(n)$
- Selection becomes $O(n)$, quicksort becomes $O(n \log n)$
- Notice that `fastselect` with `k = len(L)/2` finds median in time $O(n)$

Historical note

- C.A.R. Hoare described `quickselect` in the same paper that introduced `quicksort`, 1962

Summary

- Median of block medians helps find a good pivot in $O(n)$
- Selection becomes $O(n)$, quicksort becomes $O(n \log n)$
- Notice that `fastselect` with `k = len(L)/2` finds median in time $O(n)$

Historical note

- C.A.R. Hoare described `quickselect` in the same paper that introduced `quicksort`, 1962
- The median of medians algorithm is due to Manuel Blum, Robert Floyd, Vaughn Pratt, Ron Rivest and Robert Tarjan, 1973

Acknowledgment

Illustrations from *Algorithms* by Jeff Erickson, <https://jeffe.cs.illinois.edu/teaching/algorithms/>

Dynamic Programming

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 9

Inductive definitions

■ Factorial

- $\text{fact}(0) = 1$
- $\text{fact}(n) = n \times \text{fact}(n - 1)$

Inductive definitions

■ Factorial

- $\text{fact}(0) = 1$
- $\text{fact}(n) = n \times \text{fact}(n - 1)$

■ Insertion sort

- $\text{isort}([]) = []$
- $\text{isort}([x_0, x_1, \dots, x_n]) =$
 $\text{insert}(\text{isort}([x_0, x_1, \dots, x_{n-1}]), x_n)$

Inductive definitions . . . recursive programs

■ Factorial

- $\text{fact}(0) = 1$
- $\text{fact}(n) = n \times \text{fact}(n - 1)$

```
def fact(n):  
    if n <= 0:  
        return(1)  
    else:  
        return(n * fact(n-1))
```

■ Insertion sort

- $\text{isort}([]) = []$
- $\text{isort}([x_0, x_1, \dots, x_n]) = \text{insert}(\text{isort}([x_0, x_1, \dots, x_{n-1}]), x_n)$

```
def isort(l):  
    if l == []:  
        return(l)  
    else:  
        return(insert(isort(l[:-1]),l[-1]))
```

Optimal substructure property

- Solution to original problem can be derived by combining solutions to subproblems

- Factorial

- $\text{fact}(0) = 1$
- $\text{fact}(n) = n \times \text{fact}(n - 1)$

- Insertion sort

- $\text{isort}([]) = []$
- $\text{isort}([x_0, x_1, \dots, x_n]) = \text{insert}(\text{isort}([x_0, x_2, \dots, x_{n-1}]), x_n)$

Optimal substructure property

- Solution to original problem can be derived by combining solutions to subproblems
- $\text{fact}(n-1)$ is a subproblem of $\text{fact}(n)$
 - So are $\text{fact}(n-2)$, $\text{fact}(n-3)$, ..., $\text{fact}(0)$

■ Factorial

- $\text{fact}(0) = 1$
- $\text{fact}(n) = n \times \text{fact}(n - 1)$

■ Insertion sort

- $\text{isort}([]) = []$
- $\text{isort}([x_0, x_1, \dots, x_n]) = \text{insert}(\text{isort}([x_0, x_2, \dots, x_{n-1}]), x_n)$

Optimal substructure property

- Solution to original problem can be derived by combining solutions to subproblems
- $\text{fact}(n-1)$ is a subproblem of $\text{fact}(n)$
 - So are $\text{fact}(n-2)$, $\text{fact}(n-3)$, ..., $\text{fact}(0)$
- $\text{isort}([x_0, x_1, \dots, x_{n-1}])$ is a subproblem of $\text{isort}([x_0, x_1, \dots, x_n])$
 - So is $\text{isort}([x_i, \dots, x_j])$ for any $0 \leq i < j \leq n$

Factorial

- $\text{fact}(0) = 1$
- $\text{fact}(n) = n \times \text{fact}(n - 1)$

Insertion sort

- $\text{isort}([]) = []$
- $\text{isort}([x_0, x_1, \dots, x_n]) = \text{insert}(\text{isort}([x_0, x_1, \dots, x_{n-1}]), x_n)$

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slot for instructor i starts at $s(i)$ and finishes at $f(i)$
- Slots may overlap, so not all bookings can be honoured
- Choose a subset of bookings to maximize the number of teachers who get to use the room

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slot for instructor i starts at $s(i)$ and finishes at $f(i)$
- Slots may overlap, so not all bookings can be honoured
- Choose a subset of bookings to maximize the number of teachers who get to use the room

Subproblems

- Each subset of bookings is a subproblem

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slot for instructor i starts at $s(i)$ and finishes at $f(i)$
- Slots may overlap, so not all bookings can be honoured
- Choose a subset of bookings to maximize the number of teachers who get to use the room

Subproblems

- Each subset of bookings is a subproblem

Generic greedy strategy

- Pick one request from those in contention
- Eliminate bookings in conflict with this choice
- Solve the resulting subproblem

Subproblems . . .

- Each subset of bookings is a subproblem

Generic greedy strategy

- Pick one request from those in contention
- Eliminate bookings in conflict with this choice
- Solve the resulting subproblem

Subproblems . . .

- Each subset of bookings is a subproblem
- Given N bookings, 2^N subproblems

Generic greedy strategy

- Pick one request from those in contention
- Eliminate bookings in conflict with this choice
- Solve the resulting subproblem

Subproblems . . .

- Each subset of bookings is a subproblem
- Given N bookings, 2^N subproblems
- Greedy strategy looks at only a small fraction of subproblems

Generic greedy strategy

- Pick one request from those in contention
- Eliminate bookings in conflict with this choice
- Solve the resulting subproblem

Subproblems . . .

- Each subset of bookings is a subproblem
- Given N bookings, 2^N subproblems
- Greedy strategy looks at only a small fraction of subproblems
 - Each choice rules out a large number of subproblems

Generic greedy strategy

- Pick one request from those in contention
- Eliminate bookings in conflict with this choice
- Solve the resulting subproblem

Subproblems . . .

- Each subset of bookings is a subproblem
- Given N bookings, 2^N subproblems
- Greedy strategy looks at only a small fraction of subproblems
 - Each choice rules out a large number of subproblems
 - Greedy strategy needs a proof of optimality

Generic greedy strategy

- Pick one request from those in contention
- Eliminate bookings in conflict with this choice
- Solve the resulting subproblem

Weighted Interval scheduling

- Same scenario as before but each request comes with a **weight**
 - For instance, the room rent for using the resource

Weighted Interval scheduling

- Same scenario as before but each request comes with a **weight**
 - For instance, the room rent for using the resource
- Revised goal: maximize the total weight of the bookings that are selected
 - Not the same as maximizing the number of bookings selected

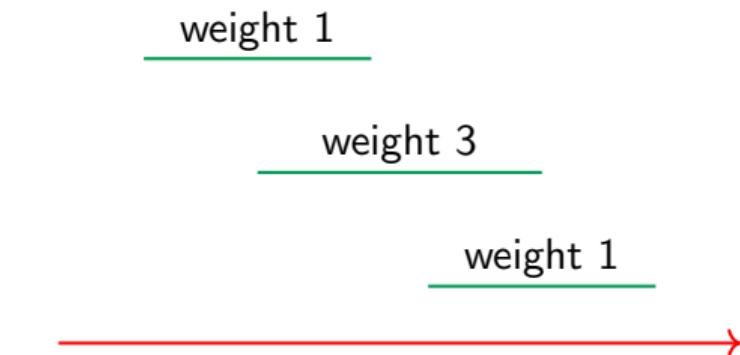
Weighted Interval scheduling

- Same scenario as before but each request comes with a **weight**
 - For instance, the room rent for using the resource
- Revised goal: maximize the total weight of the bookings that are selected
 - Not the same as maximizing the number of bookings selected
- Greedy strategy for unweighted case
 - Select request with earliest finish time

Weighted Interval scheduling

- Same scenario as before but each request comes with a **weight**
 - For instance, the room rent for using the resource
- Revised goal: maximize the total weight of the bookings that are selected
 - Not the same as maximizing the number of bookings selected
- Greedy strategy for unweighted case
 - Select request with earliest finish time

- Not a valid strategy any more



Weighted Interval scheduling

- Same scenario as before but each request comes with a **weight**
 - For instance, the room rent for using the resource
- Revised goal: maximize the total weight of the bookings that are selected
 - Not the same as maximizing the number of bookings selected
- Greedy strategy for unweighted case
 - Select request with earliest finish time

- Not a valid strategy any more

weight 1

weight 3

weight 1



- Search for another greedy strategy that works . . .

Weighted Interval scheduling

- Same scenario as before but each request comes with a **weight**
 - For instance, the room rent for using the resource
- Revised goal: maximize the total weight of the bookings that are selected
 - Not the same as maximizing the number of bookings selected
- Greedy strategy for unweighted case
 - Select request with earliest finish time

- Not a valid strategy any more

weight 1

weight 3

weight 1



- Search for another greedy strategy that works ...
- ... or look for an inductive solution that is “obviously” correct

Weighted Interval scheduling

- Order the bookings by starting time,

b_1, b_2, \dots, b_n

Weighted Interval scheduling

- Order the bookings by starting time,
 b_1, b_2, \dots, b_n
- Begin with b_1

Weighted Interval scheduling

- Order the bookings by starting time,

b_1, b_2, \dots, b_n

- Begin with b_1

- Either b_1 is part of the optimal solution, or it is not

Weighted Interval scheduling

- Order the bookings by starting time,

b_1, b_2, \dots, b_n

- Begin with b_1

- Either b_1 is part of the optimal solution, or it is not
 - Include $b_1 \Rightarrow$ eliminate conflicting requests in b_2, \dots, b_n and solve resulting subproblem

Weighted Interval scheduling

- Order the bookings by starting time,

b_1, b_2, \dots, b_n

- Begin with b_1

- Either b_1 is part of the optimal solution, or it is not
 - Include $b_1 \Rightarrow$ eliminate conflicting requests in b_2, \dots, b_n and solve resulting subproblem
 - Exclude $b_1 \Rightarrow$ solve subproblem b_2, \dots, b_n

Weighted Interval scheduling

- Order the bookings by starting time,

b_1, b_2, \dots, b_n

- Begin with b_1

- Either b_1 is part of the optimal solution, or it is not
- Include $b_1 \Rightarrow$ eliminate conflicting requests in b_2, \dots, b_n and solve resulting subproblem
- Exclude $b_1 \Rightarrow$ solve subproblem b_2, \dots, b_n
- Evaluate both options, choose the maximum

Weighted Interval scheduling

- Order the bookings by starting time,
 b_1, b_2, \dots, b_n
- Begin with b_1
 - Either b_1 is part of the optimal solution, or it is not
 - Include $b_1 \Rightarrow$ eliminate conflicting requests in b_2, \dots, b_n and solve resulting subproblem
 - Exclude $b_1 \Rightarrow$ solve subproblem b_2, \dots, b_n
 - Evaluate both options, choose the maximum
- Inductive solution considers all options

Weighted Interval scheduling

- Order the bookings by starting time,
 b_1, b_2, \dots, b_n
- Begin with b_1
 - Either b_1 is part of the optimal solution, or it is not
 - Include $b_1 \Rightarrow$ eliminate conflicting requests in b_2, \dots, b_n and solve resulting subproblem
 - Exclude $b_1 \Rightarrow$ solve subproblem b_2, \dots, b_n
 - Evaluate both options, choose the maximum
- Inductive solution considers all options
 - For each b_j , optimal solution either has b_j or does not

Weighted Interval scheduling

- Order the bookings by starting time,
 b_1, b_2, \dots, b_n
- Begin with b_1
 - Either b_1 is part of the optimal solution, or it is not
 - Include $b_1 \Rightarrow$ eliminate conflicting requests in b_2, \dots, b_n and solve resulting subproblem
 - Exclude $b_1 \Rightarrow$ solve subproblem b_2, \dots, b_n
 - Evaluate both options, choose the maximum
- Inductive solution considers all options
 - For each b_j , optimal solution either has b_j or does not
 - For b_1 , we have explicitly checked both cases

Weighted Interval scheduling

- Order the bookings by starting time,
 b_1, b_2, \dots, b_n
- Begin with b_1
 - Either b_1 is part of the optimal solution, or it is not
 - Include $b_1 \Rightarrow$ eliminate conflicting requests in b_2, \dots, b_n and solve resulting subproblem
 - Exclude $b_1 \Rightarrow$ solve subproblem b_2, \dots, b_n
 - Evaluate both options, choose the maximum
- Inductive solution considers all options
 - For each b_j , optimal solution either has b_j or does not
 - For b_1 , we have explicitly checked both cases
 - If b_2 is not in conflict with b_1 , it will be considered in both subproblems, with and without b_1

Weighted Interval scheduling

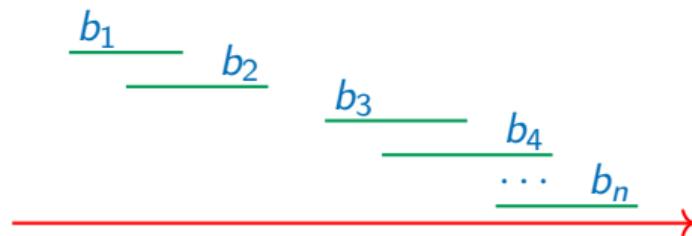
- Order the bookings by starting time,
 b_1, b_2, \dots, b_n
- Begin with b_1
 - Either b_1 is part of the optimal solution, or it is not
 - Include $b_1 \Rightarrow$ eliminate conflicting requests in b_2, \dots, b_n and solve resulting subproblem
 - Exclude $b_1 \Rightarrow$ solve subproblem b_2, \dots, b_n
 - Evaluate both options, choose the maximum
- Inductive solution considers all options
 - For each b_j , optimal solution either has b_j or does not
 - For b_1 , we have explicitly checked both cases
 - If b_2 is not in conflict with b_1 , it will be considered in both subproblems, with and without b_1
 - If b_2 is in conflict with b_1 , it will be considered in the subproblem where b_1 is excluded

Weighted Interval scheduling

- Order the bookings by starting time,
 b_1, b_2, \dots, b_n
- Begin with b_1
 - Either b_1 is part of the optimal solution, or it is not
 - Include $b_1 \Rightarrow$ eliminate conflicting requests in b_2, \dots, b_n and solve resulting subproblem
 - Exclude $b_1 \Rightarrow$ solve subproblem b_2, \dots, b_n
 - Evaluate both options, choose the maximum
- Inductive solution considers all options
 - For each b_j , optimal solution either has b_j or does not
 - For b_1 , we have explicitly checked both cases
 - If b_2 is not in conflict with b_1 , it will be considered in both subproblems, with and without b_1
 - If b_2 is in conflict with b_1 , it will be considered in the subproblem where b_1 is excluded
 - ...

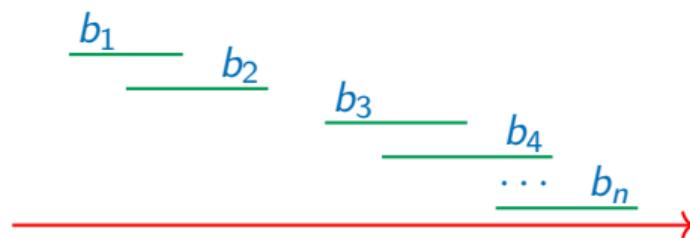
The challenge

- b_1 is in conflict with b_2 , but both are compatible with b_3, b_4, \dots, b_n



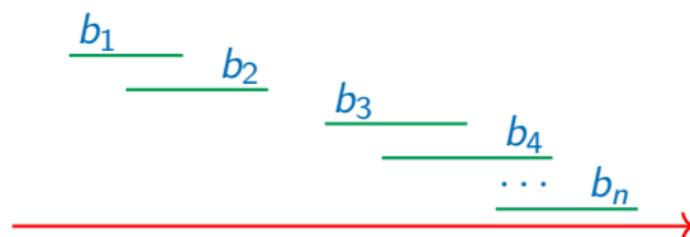
The challenge

- b_1 is in conflict with b_2 , but both are compatible with b_3, b_4, \dots, b_n
 - Choose $b_1 \Rightarrow$ subproblem b_3, \dots, b_n



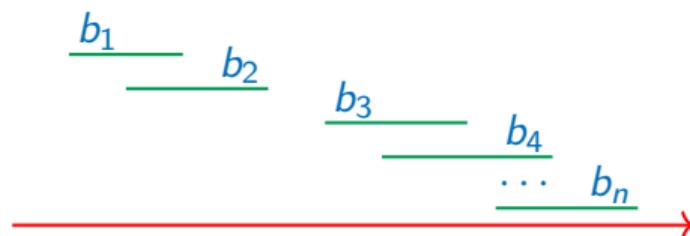
The challenge

- b_1 is in conflict with b_2 , but both are compatible with b_3, b_4, \dots, b_n
 - Choose $b_1 \Rightarrow$ subproblem b_3, \dots, b_n
 - Exclude $b_1 \Rightarrow$ subproblem b_2, \dots, b_n



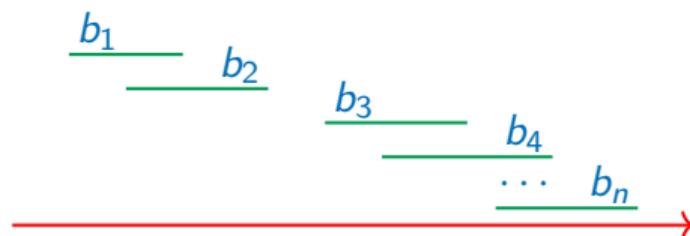
The challenge

- b_1 is in conflict with b_2 , but both are compatible with b_3, b_4, \dots, b_n
 - Choose $b_1 \Rightarrow$ subproblem b_3, \dots, b_n
 - Exclude $b_1 \Rightarrow$ subproblem b_2, \dots, b_n
 - Next stage:
 - Choose/exclude b_2



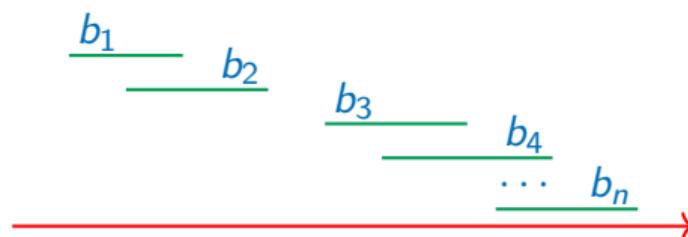
The challenge

- b_1 is in conflict with b_2 , but both are compatible with b_3, b_4, \dots, b_n
 - Choose $b_1 \Rightarrow$ subproblem b_3, \dots, b_n
 - Exclude $b_1 \Rightarrow$ subproblem b_2, \dots, b_n
 - Next stage:
 - Choose/exclude b_2
 - Both choices result in b_3, \dots, b_n , same subproblem



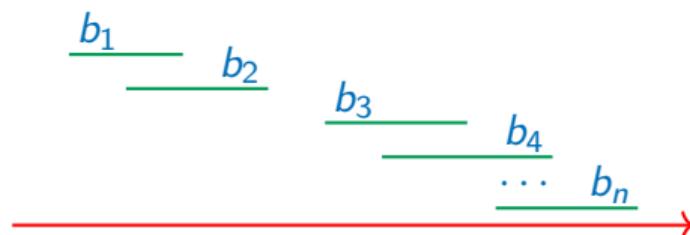
The challenge

- b_1 is in conflict with b_2 , but both are compatible with b_3, b_4, \dots, b_n
 - Choose $b_1 \Rightarrow$ subproblem b_3, \dots, b_n
 - Exclude $b_1 \Rightarrow$ subproblem b_2, \dots, b_n
 - Next stage:
 - Choose/exclude b_2
 - Both choices result in b_3, \dots, b_n , same subproblem
- Inductive solution generates same subproblem at different stages



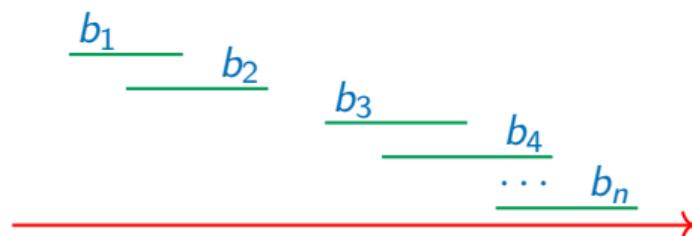
The challenge

- b_1 is in conflict with b_2 , but both are compatible with b_3, b_4, \dots, b_n
 - Choose $b_1 \Rightarrow$ subproblem b_3, \dots, b_n
 - Exclude $b_1 \Rightarrow$ subproblem b_2, \dots, b_n
 - Next stage:
 - Choose/exclude b_2
 - Both choices result in b_3, \dots, b_n , same subproblem
- Inductive solution generates same subproblem at different stages
- Naive recursive implementation evaluates each instance of subproblem from scratch



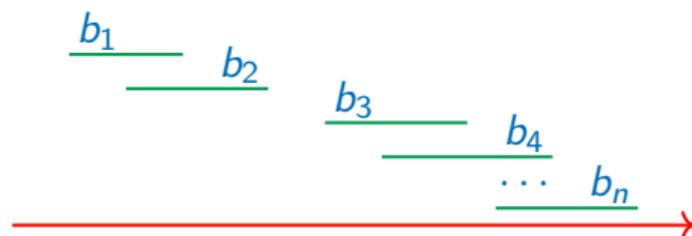
The challenge

- b_1 is in conflict with b_2 , but both are compatible with b_3, b_4, \dots, b_n
 - Choose $b_1 \Rightarrow$ subproblem b_3, \dots, b_n
 - Exclude $b_1 \Rightarrow$ subproblem b_2, \dots, b_n
 - Next stage:
 - Choose/exclude b_2
 - Both choices result in b_3, \dots, b_n , same subproblem
- Inductive solution generates same subproblem at different stages
- Naive recursive implementation evaluates each instance of subproblem from scratch
- Can we avoid this wasteful recomputation?



The challenge

- b_1 is in conflict with b_2 , but both are compatible with b_3, b_4, \dots, b_n
 - Choose $b_1 \Rightarrow$ subproblem b_3, \dots, b_n
 - Exclude $b_1 \Rightarrow$ subproblem b_2, \dots, b_n
 - Next stage:
 - Choose/exclude b_2
 - Both choices result in b_3, \dots, b_n , same subproblem



- Inductive solution generates same subproblem at different stages
- Naive recursive implementation evaluates each instance of subproblem from scratch
- Can we avoid this wasteful recomputation?
- Memoization and dynamic programming

Memoization

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 9

Inductive definitions, recursive programs, subproblems

■ Factorial

- $\text{fact}(0) = 1$
- $\text{fact}(n) = n \times \text{fact}(n - 1)$

■ Insertion sort

- $\text{isort}([]) = []$ p
- $\text{isort}([x_0, x_1, \dots, x_n]) =$
 $\text{insert}(\text{isort}([x_0, x_2, \dots, x_{n-1}]), x_n)$

Inductive definitions, recursive programs, subproblems

■ Factorial

- $\text{fact}(0) = 1$
- $\text{fact}(n) = n \times \text{fact}(n - 1)$

■ Insertion sort

- $\text{isort}([]) = []$ p
- $\text{isort}([x_0, x_1, \dots, x_n]) =$
 $\text{insert}(\text{isort}([x_0, x_2, \dots, x_{n-1}]), x_n)$

```
def fact(n):  
    if n <= 0:  
        return(1)  
    else:  
        return(n * fact(n-1))
```

Inductive definitions, recursive programs, subproblems

■ Factorial

- $\text{fact}(0) = 1$
- $\text{fact}(n) = n \times \text{fact}(n - 1)$

■ Insertion sort

- $\text{isort}([]) = []$ p
- $\text{isort}([x_0, x_1, \dots, x_n]) = \text{insert}(\text{isort}([x_0, x_2, \dots, x_{n-1}]), x_n)$

```
def fact(n):  
    if n <= 0:  
        return(1)  
    else:  
        return(n * fact(n-1))
```

■ $\text{fact}(n-1)$ is a subproblem of $\text{fact}(n)$

- So are $\text{fact}(n-2)$, $\text{fact}(n-3)$, ..., $\text{fact}(0)$

■ $\text{isort}([x_0, x_1, \dots, x_{n-1}])$ is a subproblem of $\text{isort}([x_0, x_2, \dots, x_n])$

- So is $\text{isort}([x_i, \dots, x_j])$ for any $0 \leq i < j \leq n$

■ Solution to original problem can be derived by combining solutions to subproblems

Evaluating subproblems

■ Fibonacci numbers

- $\text{fib}(0) = 0$
- $\text{fib}(1) = 1$
- $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

Evaluating subproblems

■ Fibonacci numbers

- $fib(0) = 0$
- $fib(1) = 1$
- $fib(n) = fib(n-1) + fib(n-2)$

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) + fib(n-2)  
    return(value)
```

Evaluating subproblems

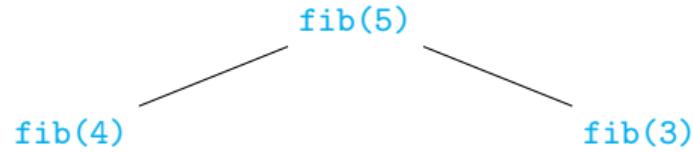
Evaluating `fib(5)`

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
               fib(n-2)  
    return(value)
```

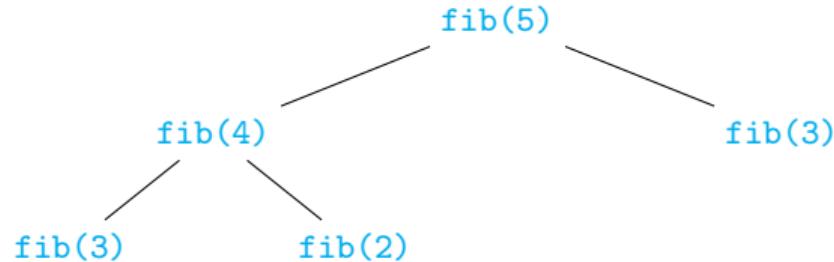
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
               fib(n-2)  
    return(value)
```

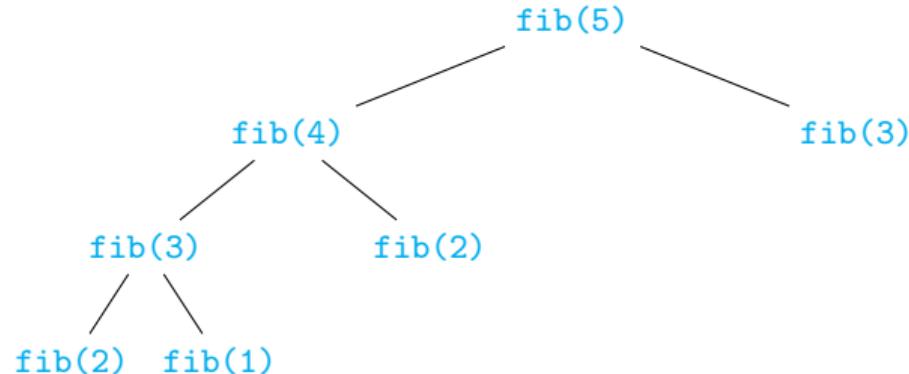
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
               fib(n-2)  
    return(value)
```

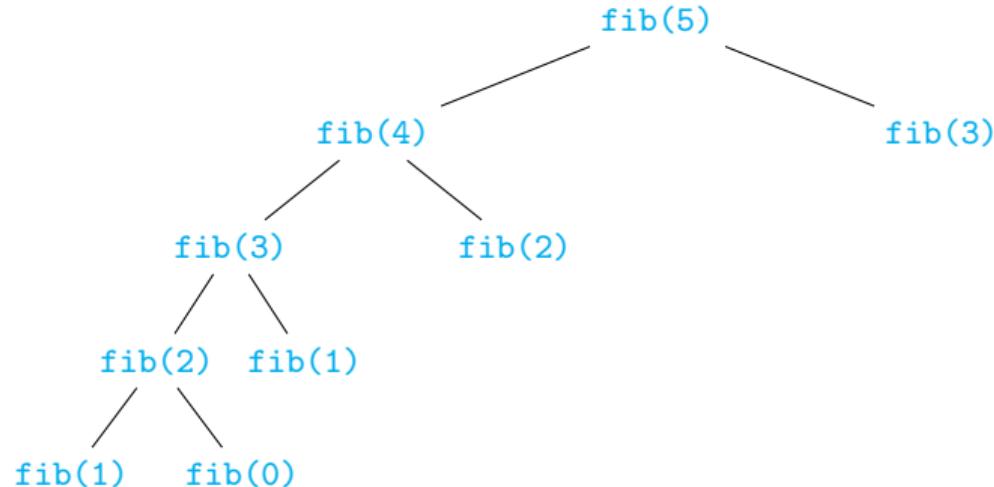
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
               fib(n-2)  
    return(value)
```

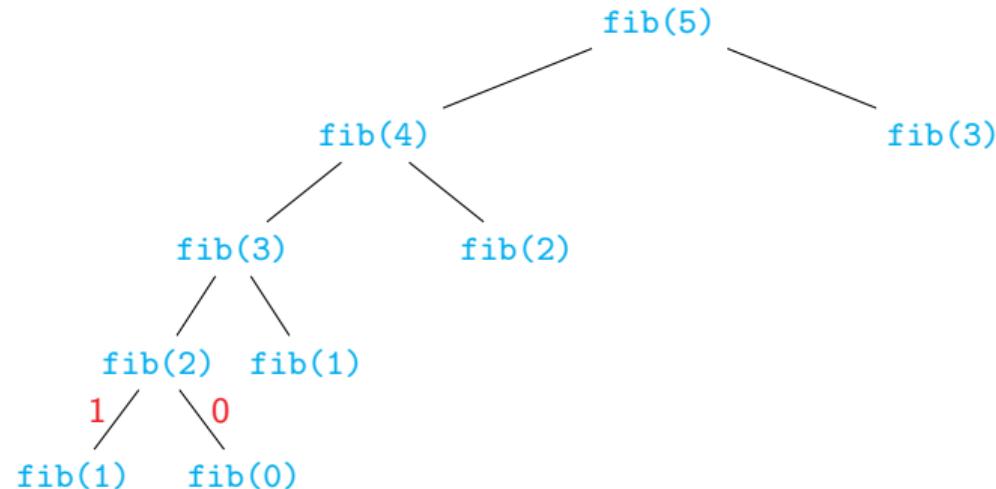
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
               fib(n-2)  
    return(value)
```

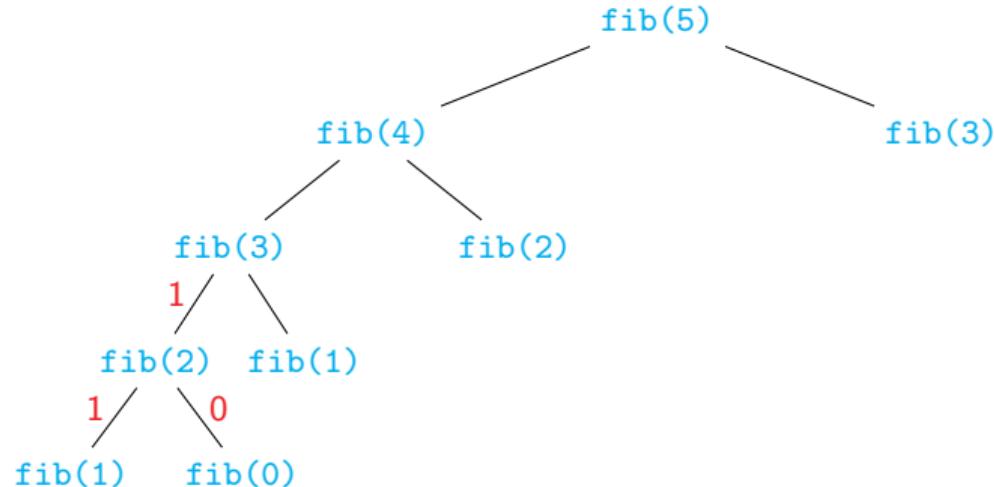
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

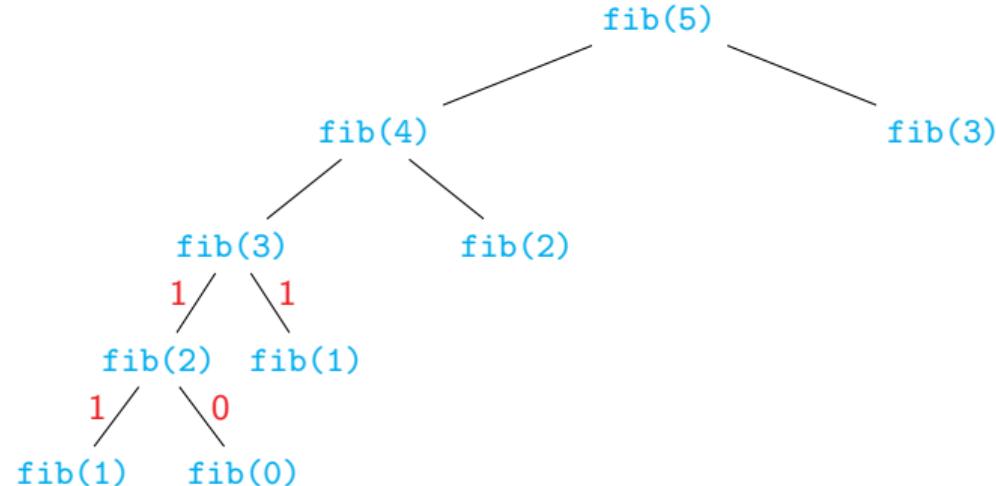
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

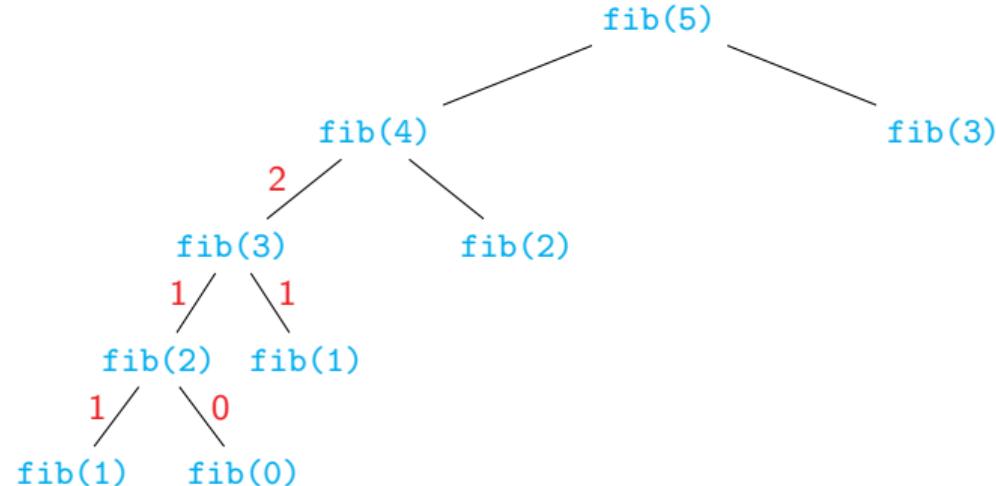
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

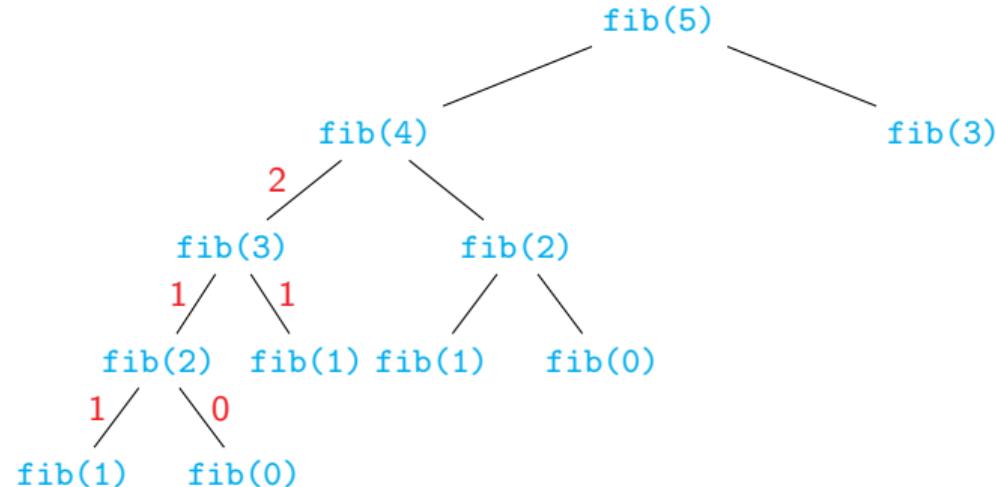
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

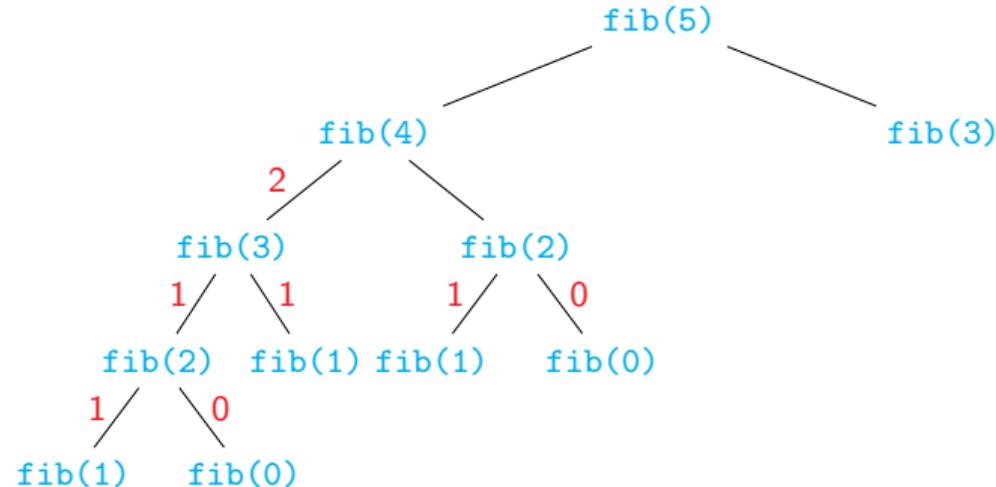
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

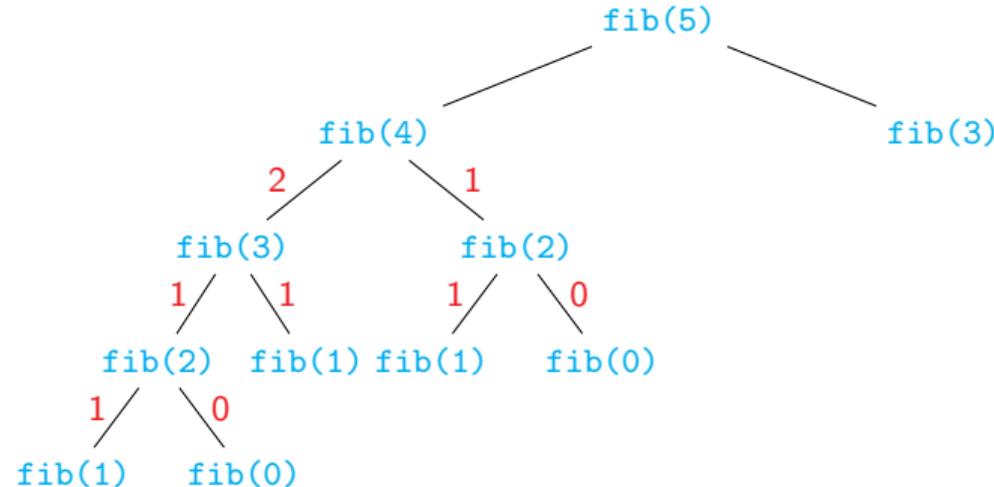
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

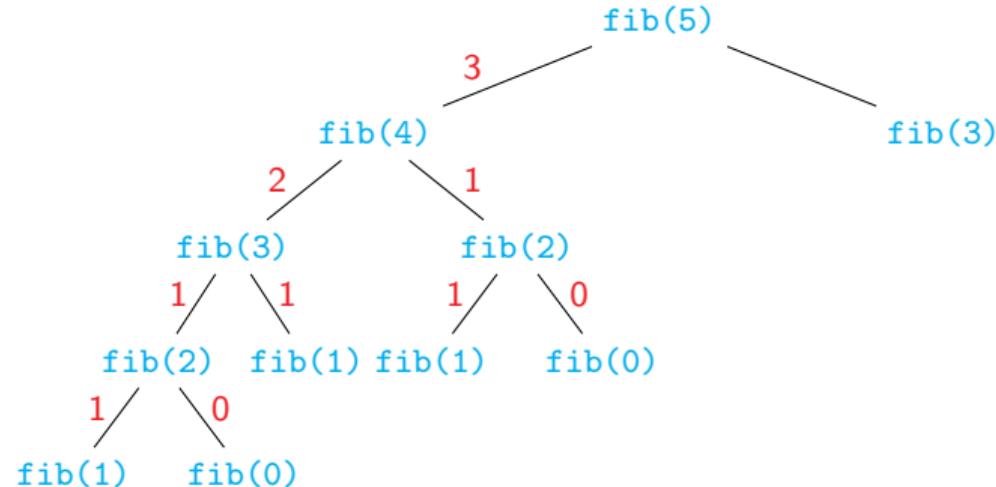
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

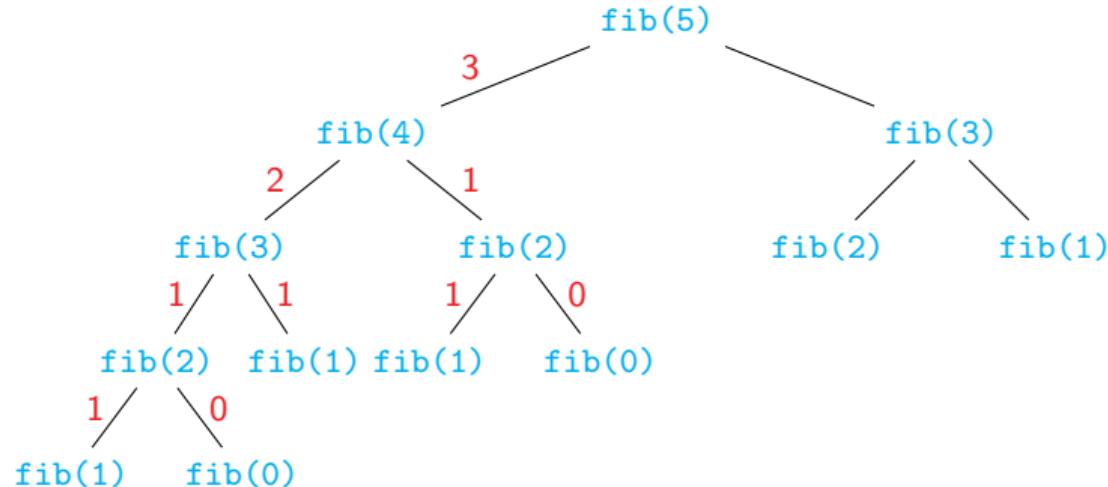
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
               fib(n-2)  
    return(value)
```

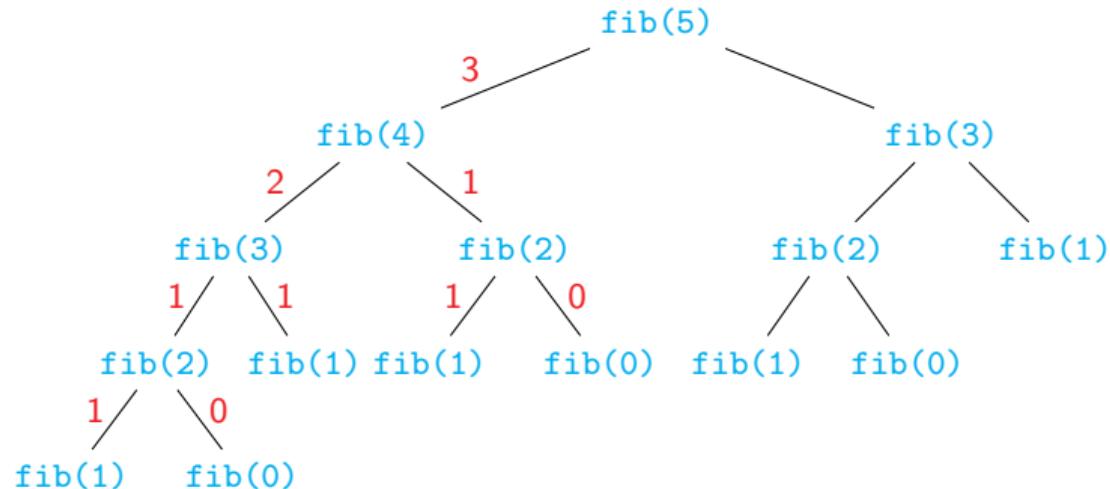
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
               fib(n-2)  
    return(value)
```

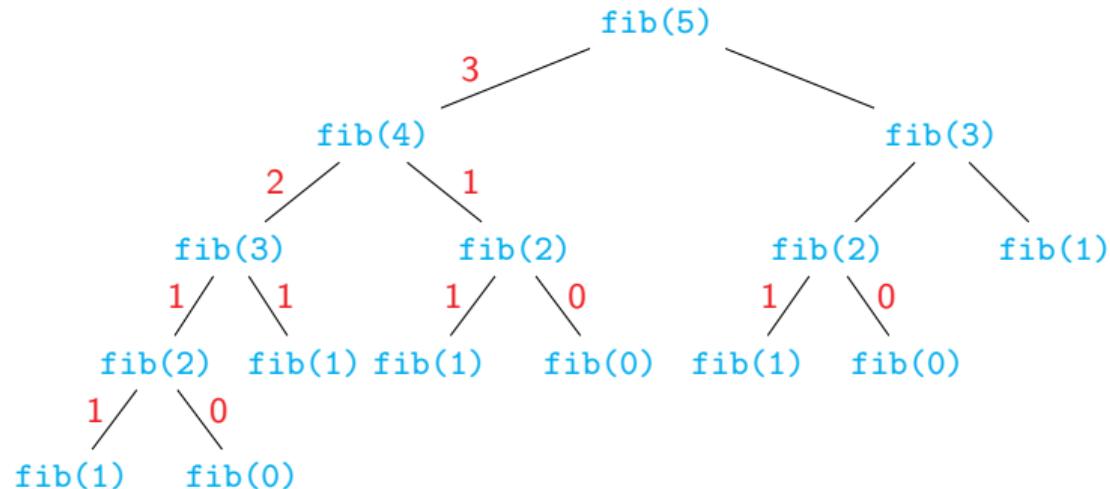
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

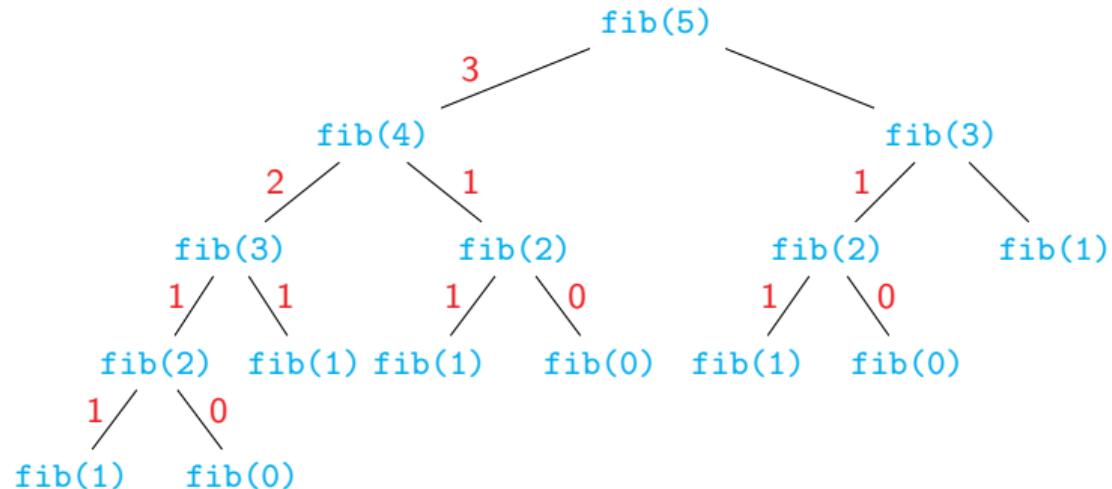
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

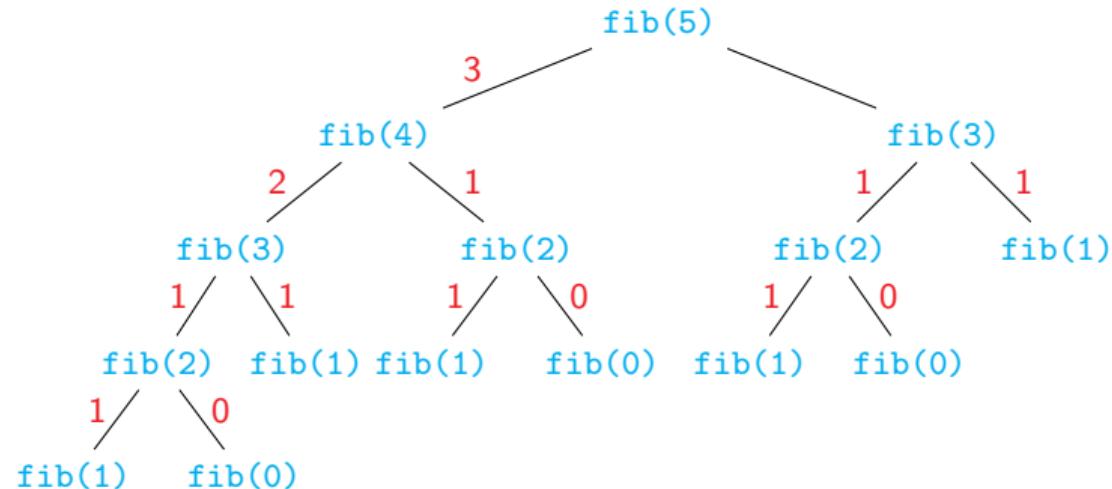
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

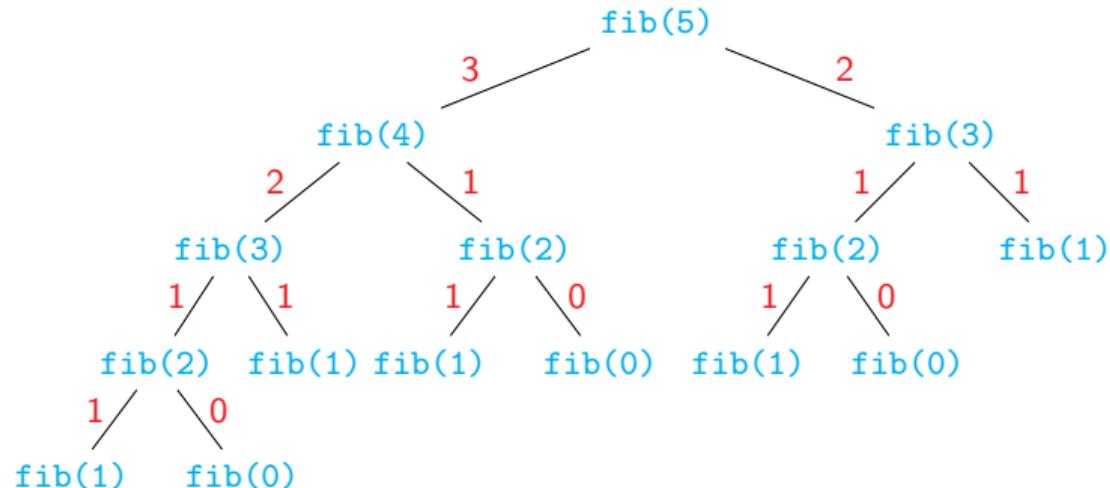
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

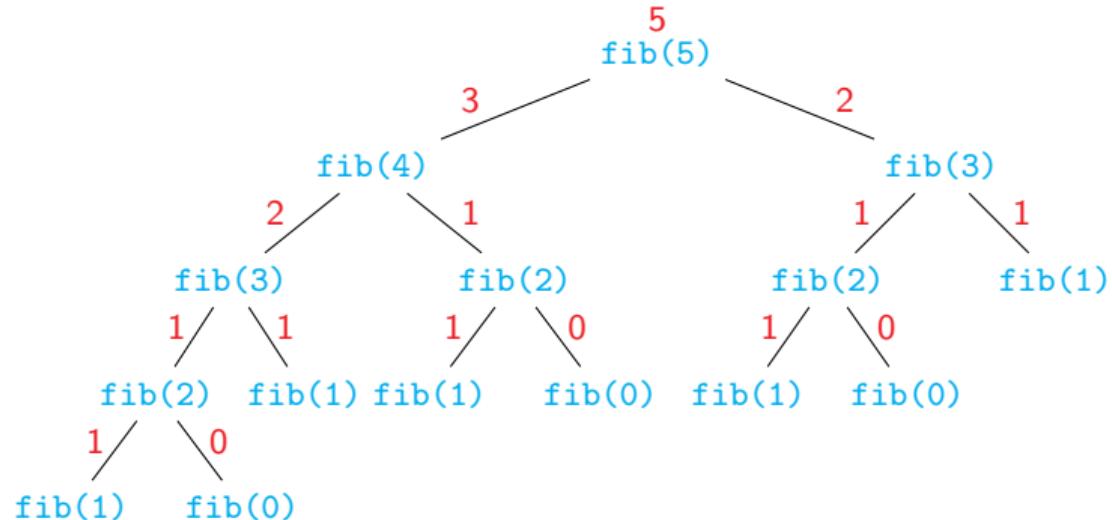
Evaluating $\text{fib}(5)$



Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
               fib(n-2)  
    return(value)
```

Evaluating $\text{fib}(5)$

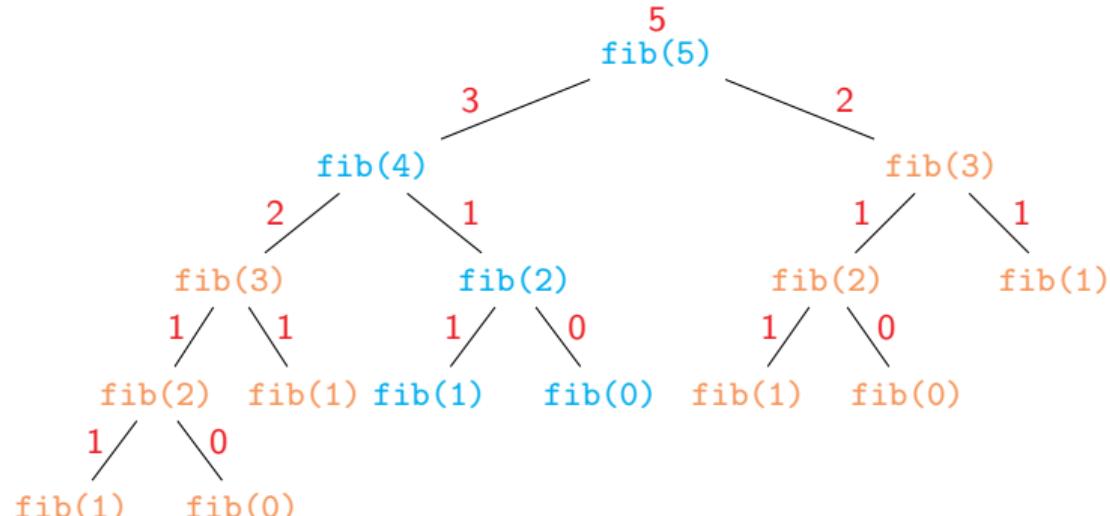


Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

- Wasteful recomputation
- Computation tree grows exponentially

Evaluating $\text{fib}(5)$

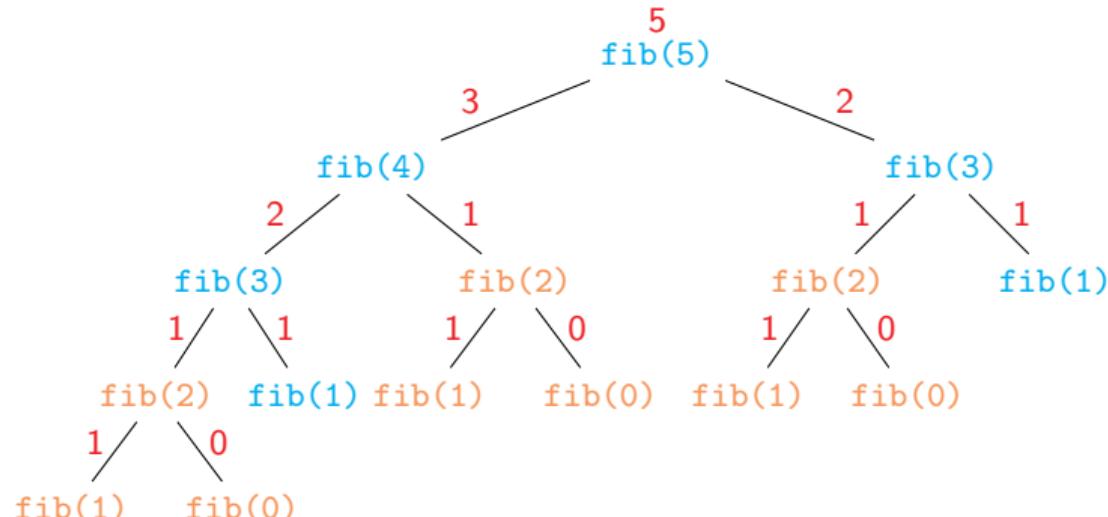


Evaluating subproblems

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) +  
                fib(n-2)  
    return(value)
```

- Wasteful recomputation
- Computation tree grows exponentially

Evaluating $\text{fib}(5)$



Evaluating subproblems

- Build a table of values already computed
 - Memory table

Evaluating subproblems

- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before

Evaluating subproblems

- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table

Evaluating subproblems

- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call

Evaluating subproblems

- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear

Evaluating subproblems

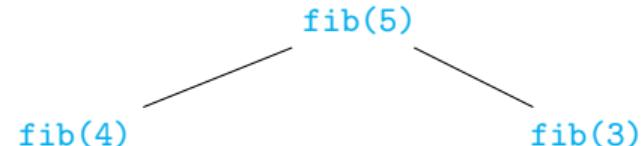
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear

`fib(5)`

<code>k</code>						
<code>fib(k)</code>						

Evaluating subproblems

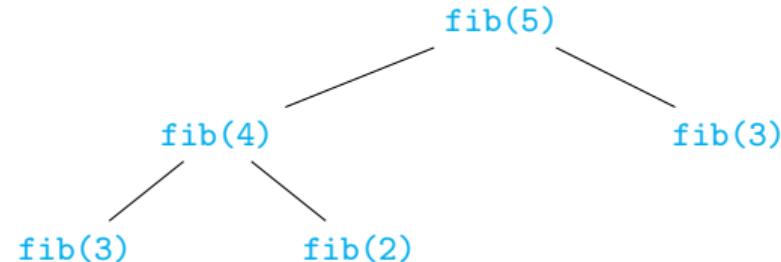
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k						
$\text{fib}(k)$						

Evaluating subproblems

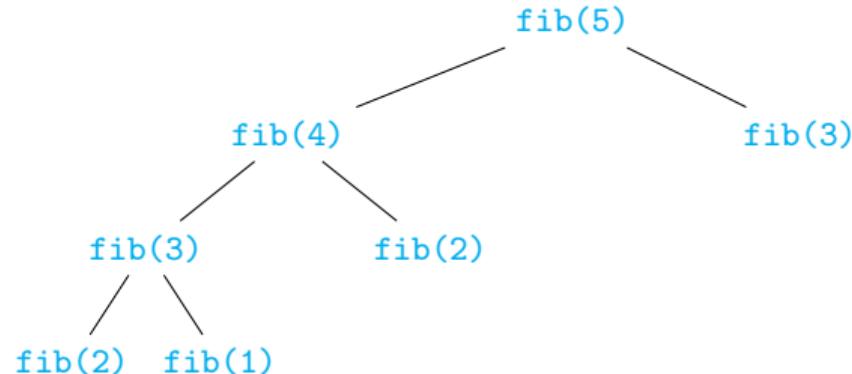
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k						
$\text{fib}(k)$						

Evaluating subproblems

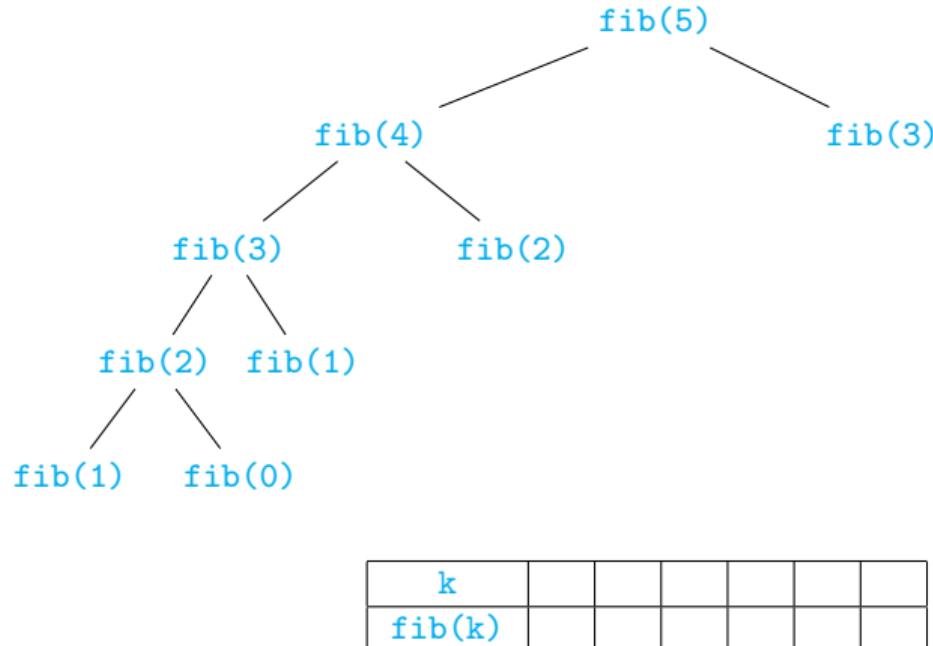
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k						
$\text{fib}(k)$						

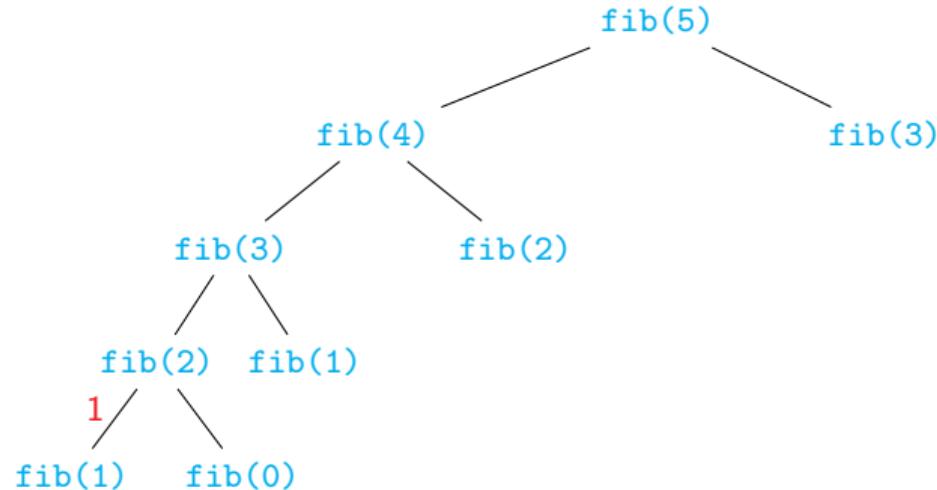
Evaluating subproblems

- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



Evaluating subproblems

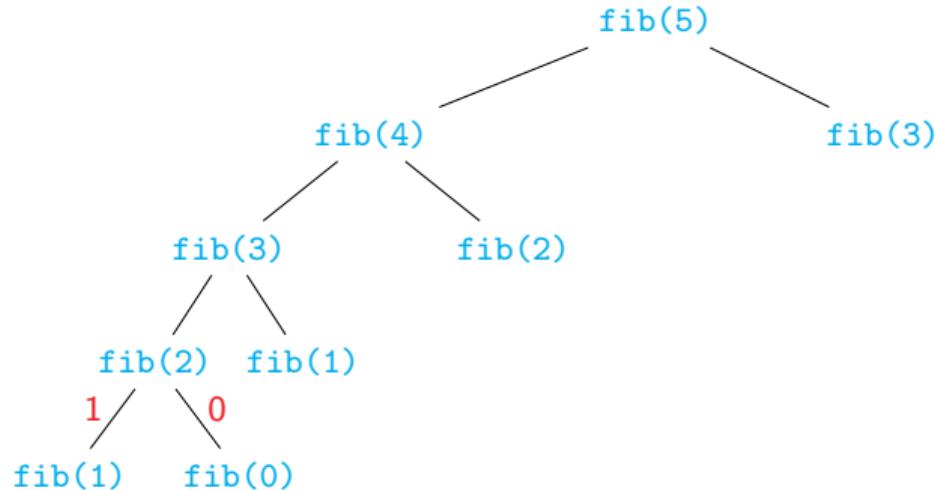
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1					
$\text{fib}(k)$	1					

Evaluating subproblems

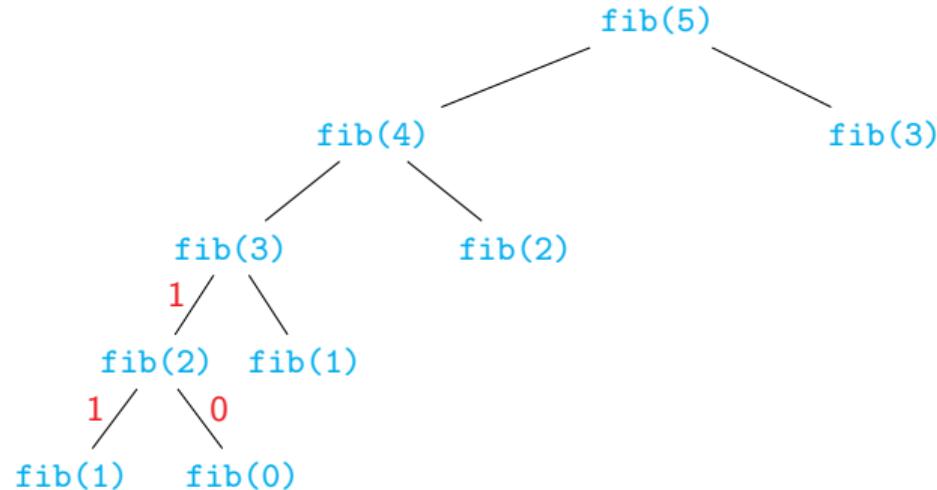
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0				
$\text{fib}(k)$	1	0				

Evaluating subproblems

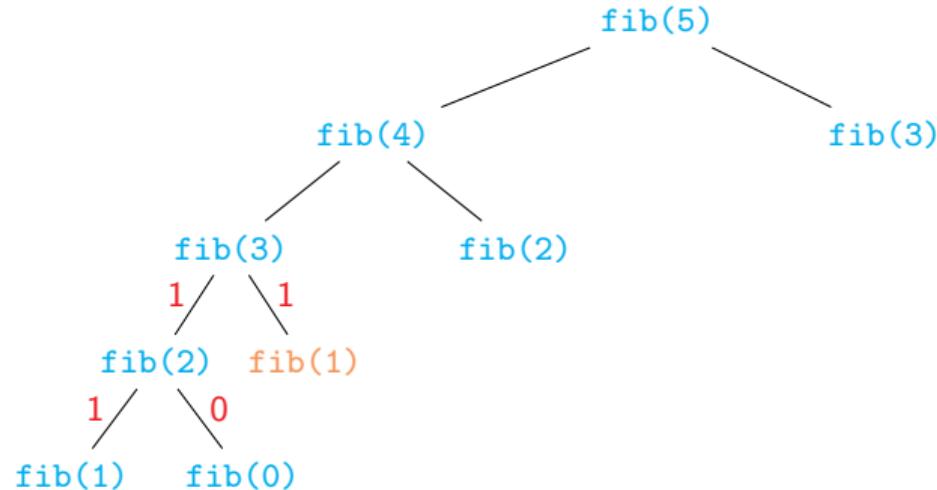
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2			
$\text{fib}(k)$	1	0	1			

Evaluating subproblems

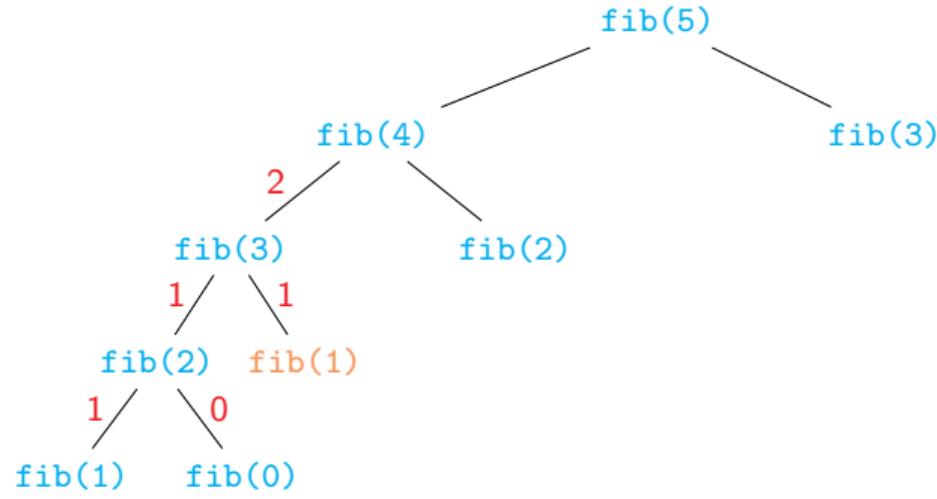
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2			
$\text{fib}(k)$	1	0	1			

Evaluating subproblems

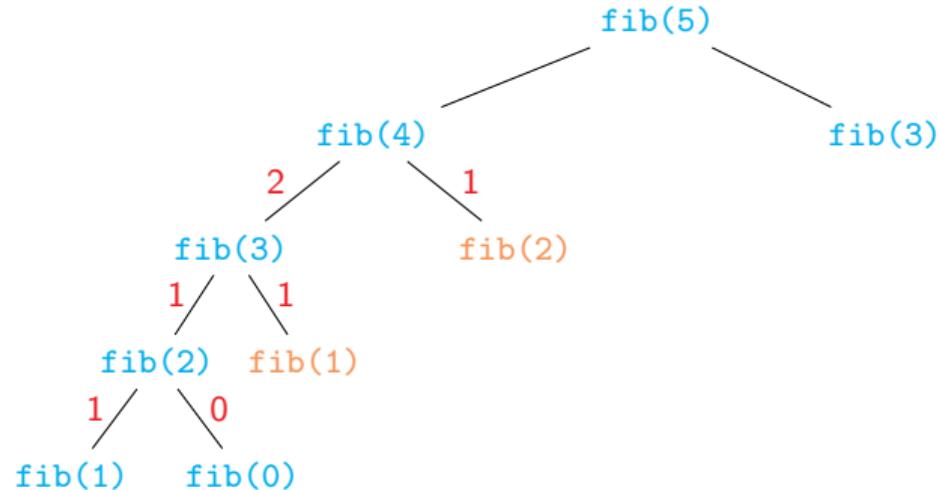
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2	3		
$\text{fib}(k)$	1	0	1	2		

Evaluating subproblems

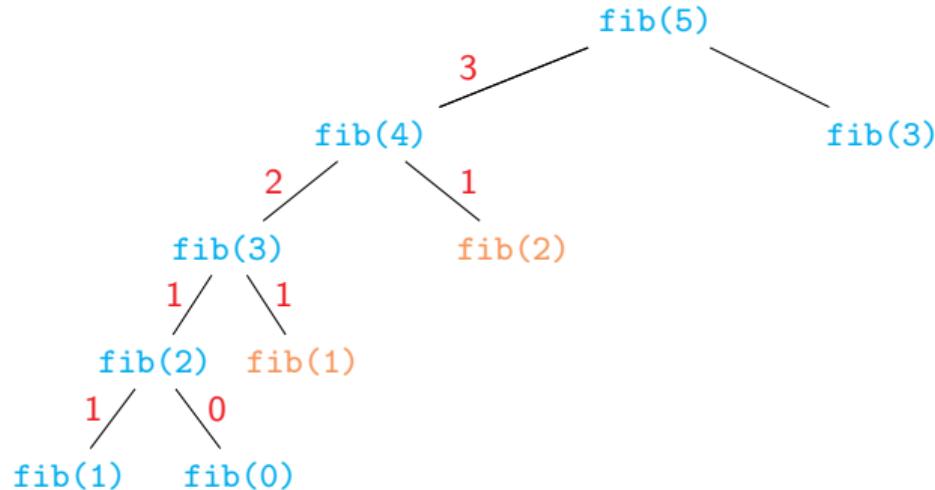
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2	3	
$\text{fib}(k)$	1	0	1	2	

Evaluating subproblems

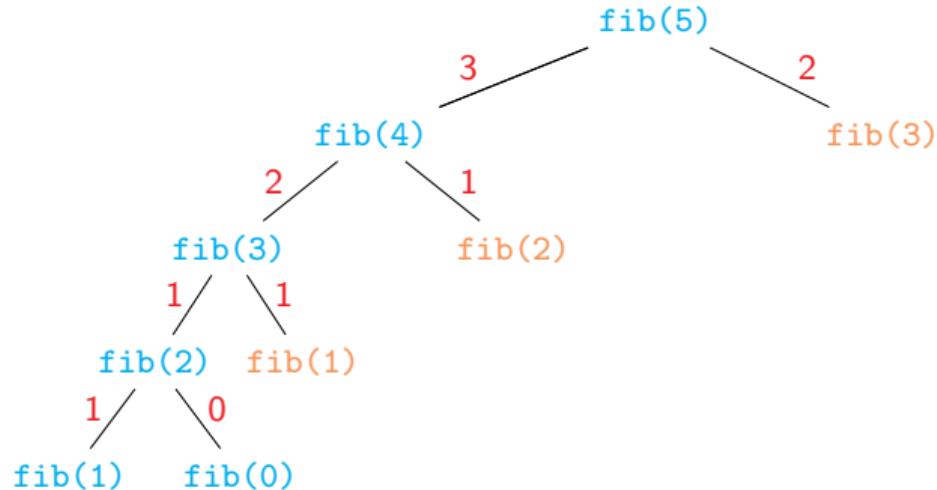
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2	3	4	
$\text{fib}(k)$	1	0	1	2	3	

Evaluating subproblems

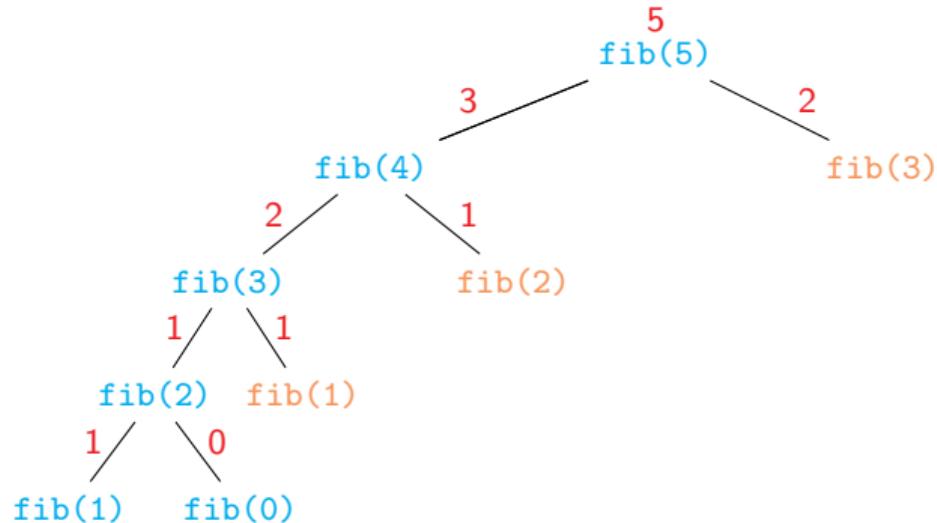
- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2	3	4	
$\text{fib}(k)$	1	0	1	2	3	

Evaluating subproblems

- Build a table of values already computed
 - Memory table
- Memoization
 - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2	3	4	5
$\text{fib}(k)$	1	0	1	2	3	5

Memoizing recursive implementations

```
def fib(n):  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) + fib(n-2)  
    return(value)
```

Memoizing recursive implementations

```
def fib(n):
    if n in fibtable.keys():
        return(fibtable[n])
    if n <= 1:
        value = n
    else:
        value = fib(n-1) + fib(n-2)
    fibtable[n] = value
    return(value)
```

Memoizing recursive implementations

```
def fib(n):  
    if n in fibtable.keys():  
        return(fibtable[n])  
  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) + fib(n-2)  
  
    fibtable[n] = value  
  
    return(value)
```

In general

```
def f(x,y,z):  
    if (x,y,z) in ftable.keys():  
        return(ftable[(x,y,z)])  
  
    recursively compute value  
    from subproblems  
  
    ftable[(x,y,z)] = value  
  
    return(value)
```

Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies form a dag

Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies form a dag
- Solve subproblems in topological order
 - Never need to make a recursive call

Dynamic programming

- Anticipate the structure of subproblems

- Derive from inductive definition
 - Dependencies form a dag

Evaluating `fib(5)`

`fib(5)`

- Solve subproblems in topological order

- Never need to make a recursive call

`fib(4)`

`fib(3)`

`fib(2)`

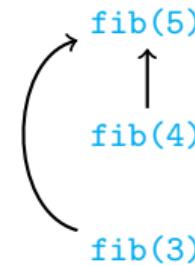
`fib(1)`

`fib(0)`

Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies form a dag
- Solve subproblems in topological order
 - Never need to make a recursive call

Evaluating $\text{fib}(5)$



$\text{fib}(2)$

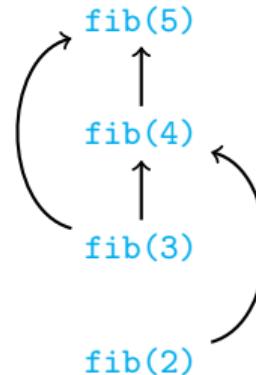
$\text{fib}(1)$

$\text{fib}(0)$

Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies form a dag
- Solve subproblems in topological order
 - Never need to make a recursive call

Evaluating $\text{fib}(5)$



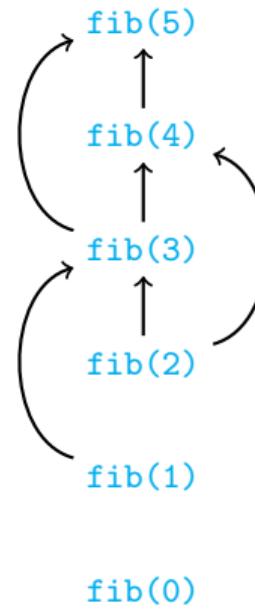
$\text{fib}(1)$

$\text{fib}(0)$

Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies form a dag
- Solve subproblems in topological order
 - Never need to make a recursive call

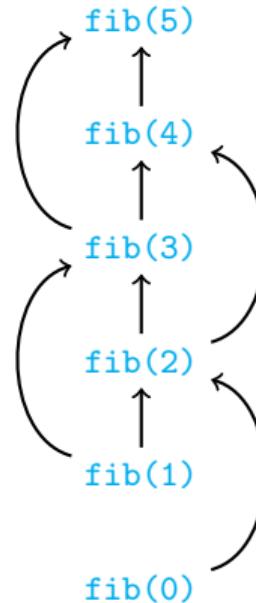
Evaluating $\text{fib}(5)$



Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies form a dag
- Solve subproblems in topological order
 - Never need to make a recursive call

Evaluating $\text{fib}(5)$



Summary

Memoization

- Store subproblem values in a table
- Look up the table before making a recursive call

Summary

Memoization

- Store subproblem values in a table
- Look up the table before making a recursive call

Dynamic programming

- Solve subproblems in topological order of dependency
 - Dependencies must form a dag
- Iterative evaluation of subproblems, no recursion

Grid Paths

Madhavan Mukund

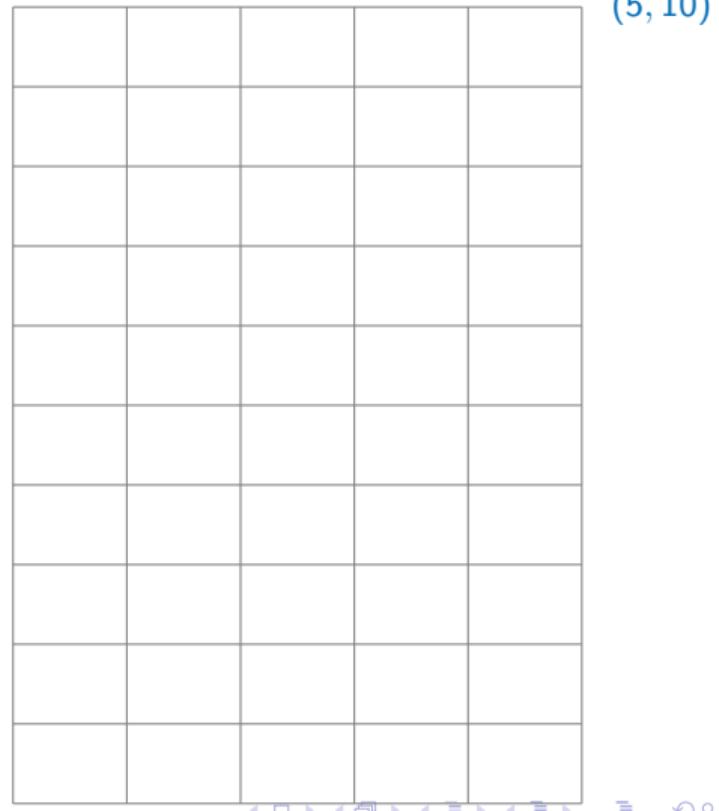
<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 9

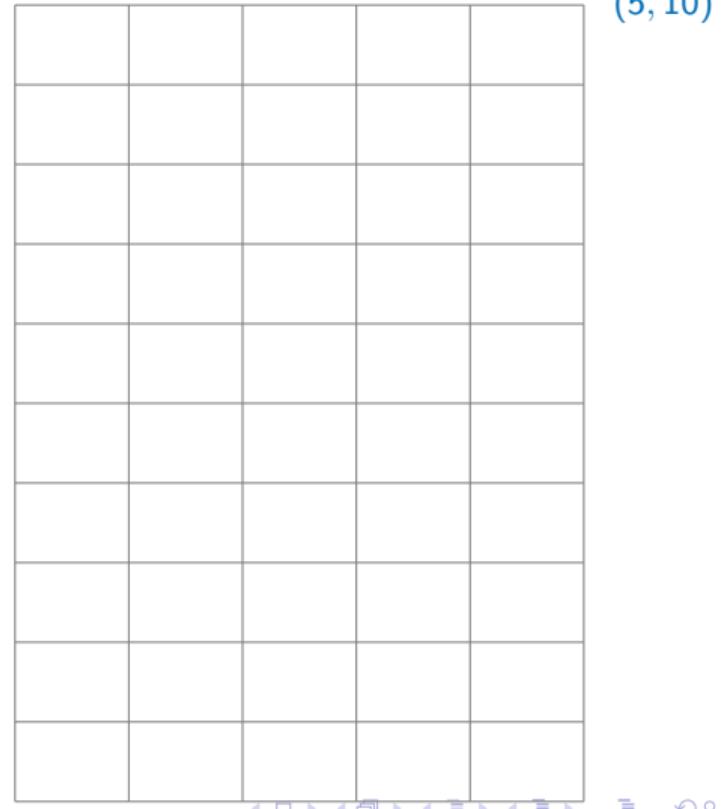
Grid paths

- Rectangular grid of one-way roads



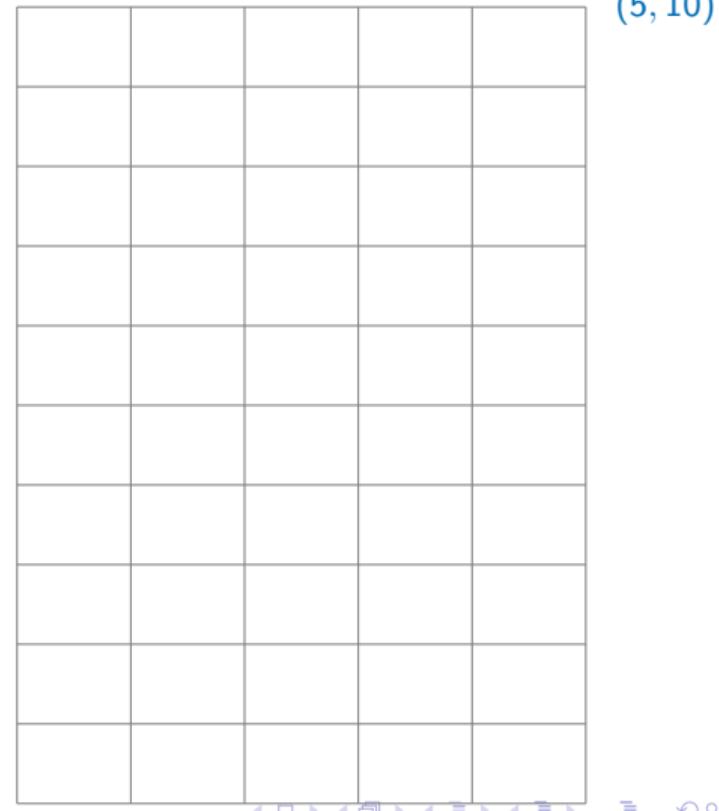
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right



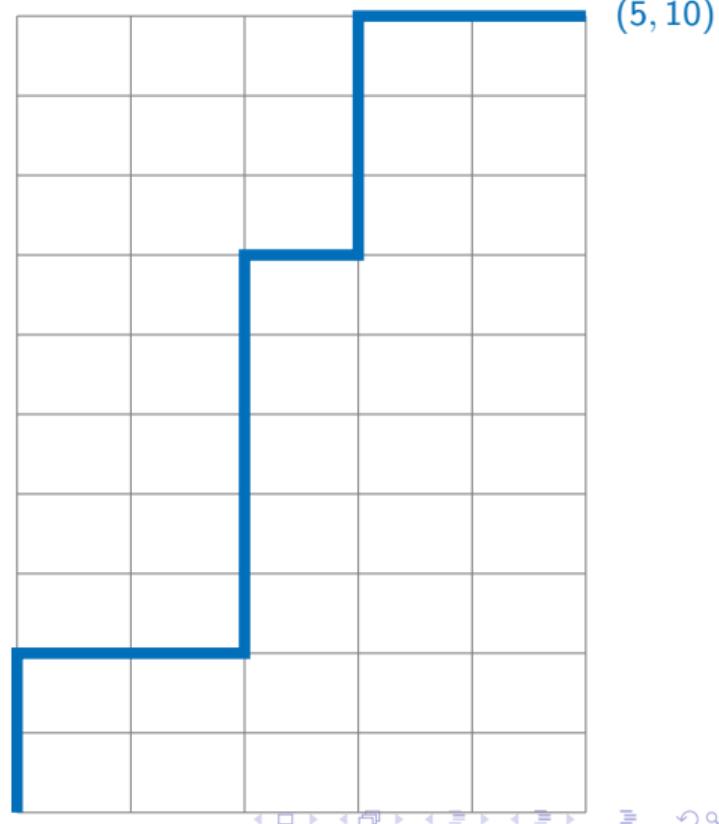
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n) ?



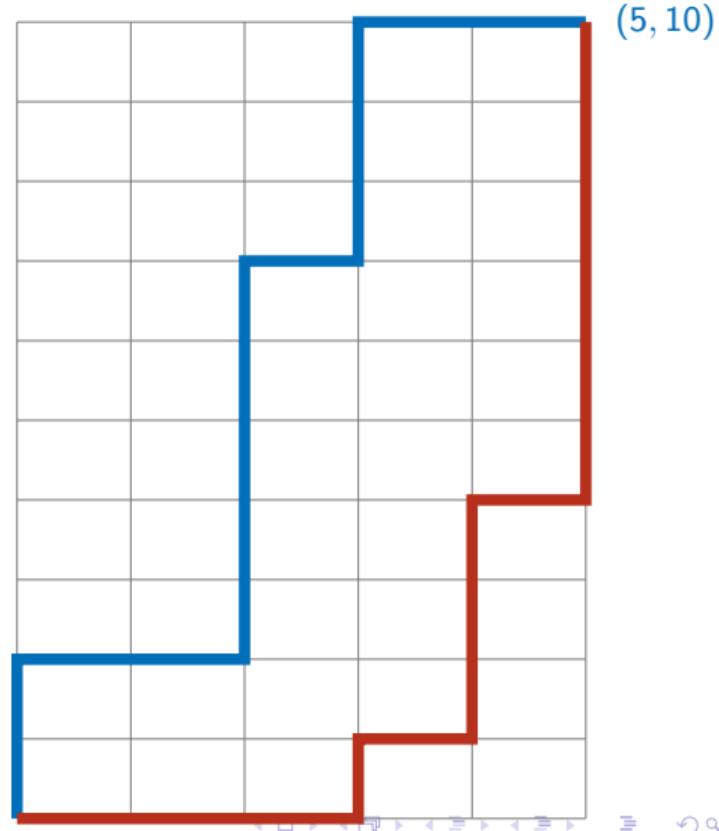
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n) ?



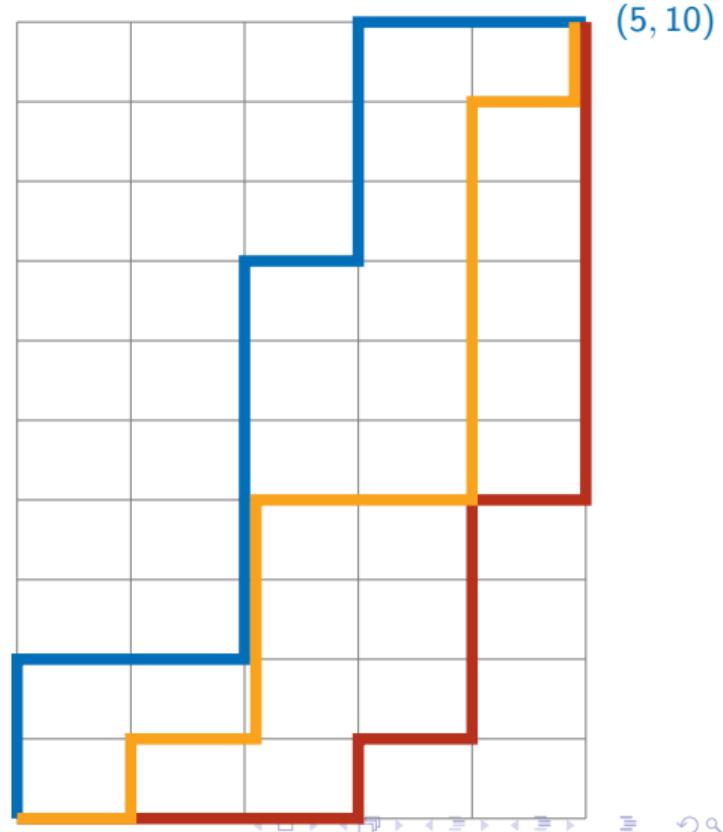
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n) ?



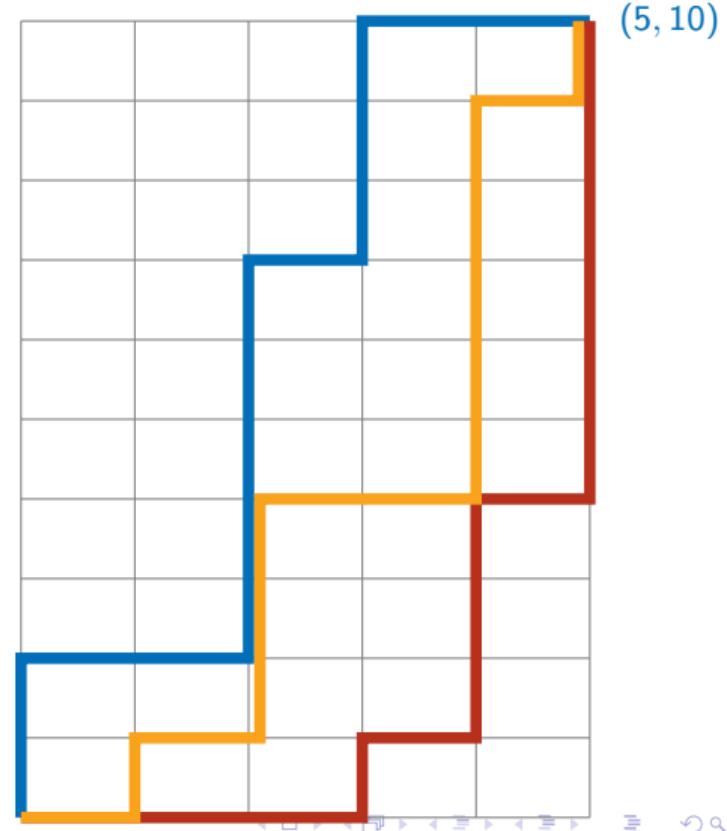
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n) ?



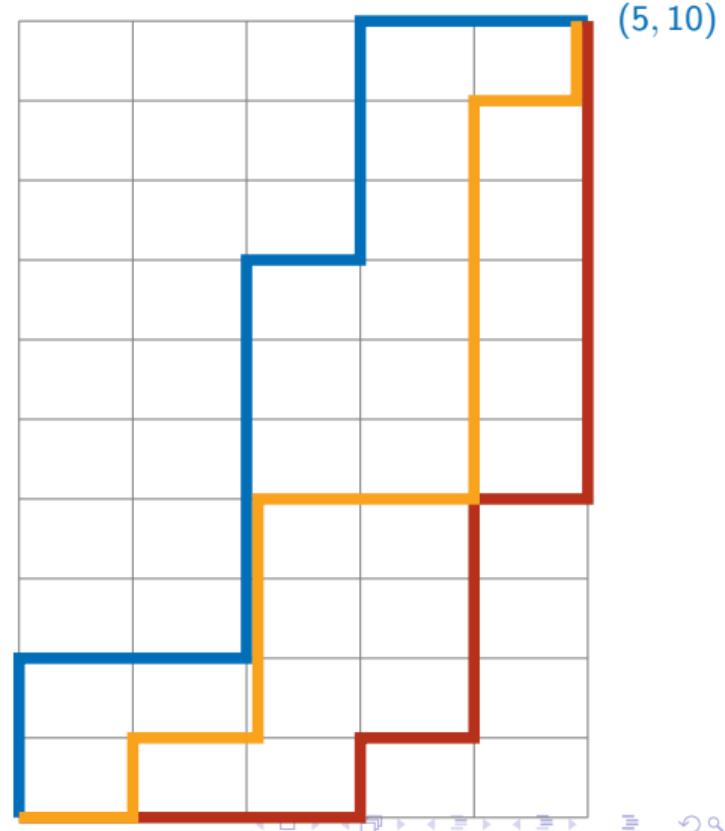
Combinatorial solution

- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
 - In general $m+n$ segments from $(0, 0)$ to (m, n)



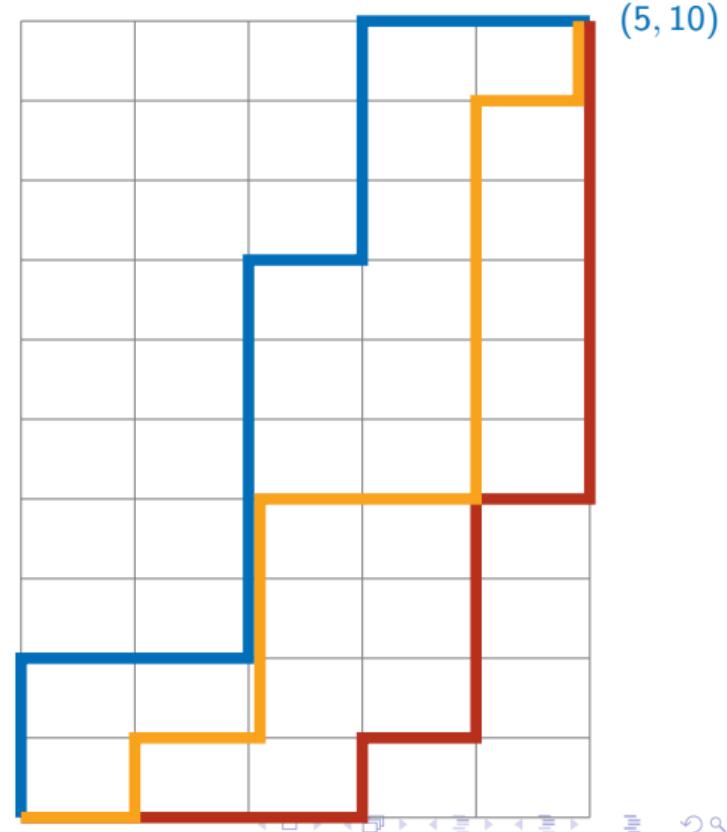
Combinatorial solution

- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
 - In general $m+n$ segments from $(0, 0)$ to (m, n)
- Out of 15, exactly 5 are right moves, 10 are up moves



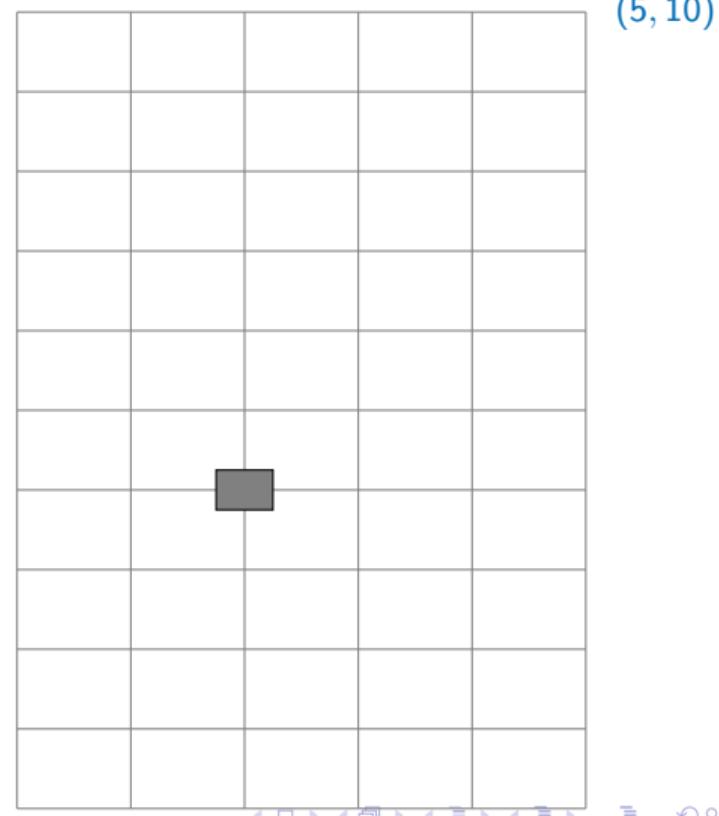
Combinatorial solution

- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
 - In general $m+n$ segments from $(0, 0)$ to (m, n)
- Out of 15, exactly 5 are right moves, 10 are up moves
- Fix the positions of the 5 right moves among the 15 positions overall
 - $\binom{15}{5} = \frac{15!}{10! \cdot 5!} = 3003$
 - Same as $\binom{15}{10}$ — fix the 10 up moves



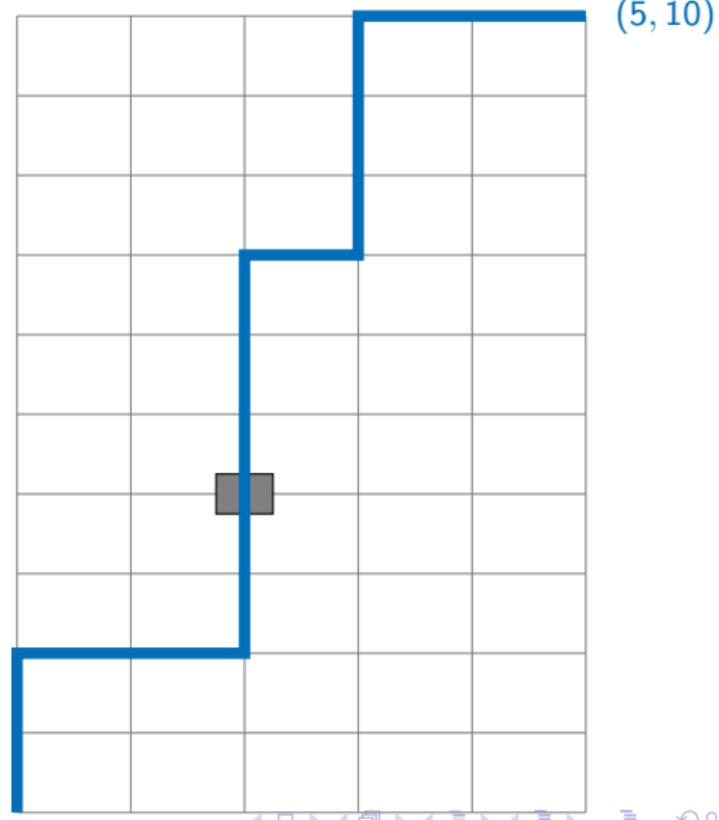
Holes

- What if an intersection is blocked?
 - For instance, $(2, 4)$



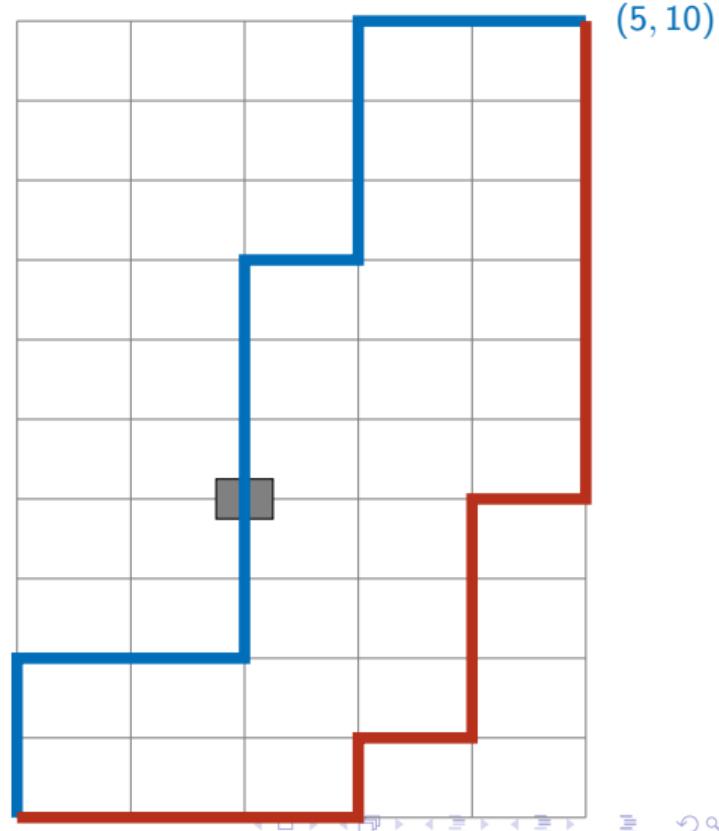
Holes

- What if an intersection is blocked?
 - For instance, $(2, 4)$
- Need to discard paths passing through $(2, 4)$
 - Two of our earlier examples are invalid paths



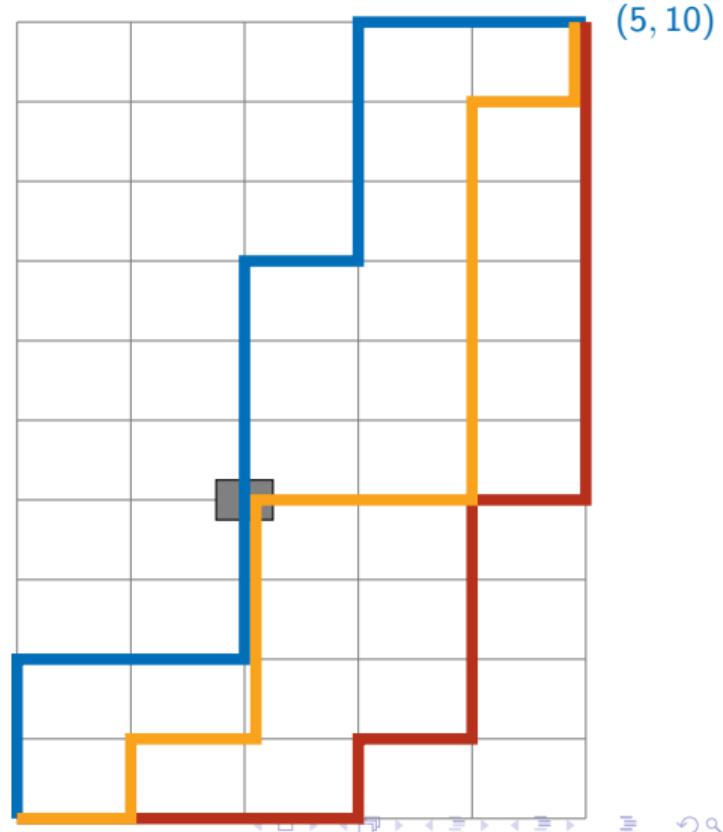
Holes

- What if an intersection is blocked?
 - For instance, $(2, 4)$
- Need to discard paths passing through $(2, 4)$
 - Two of our earlier examples are invalid paths



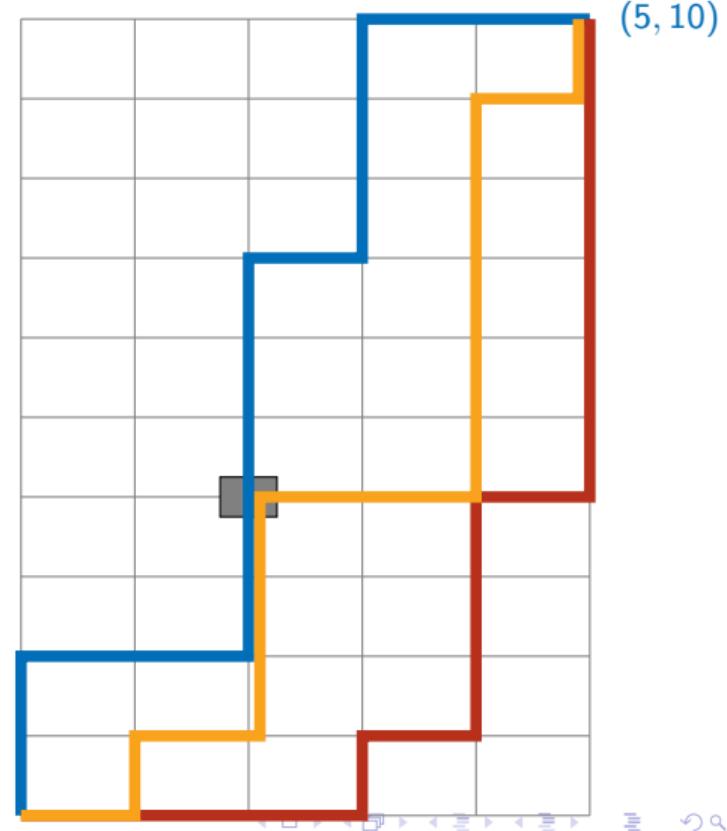
Holes

- What if an intersection is blocked?
 - For instance, $(2, 4)$
- Need to discard paths passing through $(2, 4)$
 - Two of our earlier examples are invalid paths



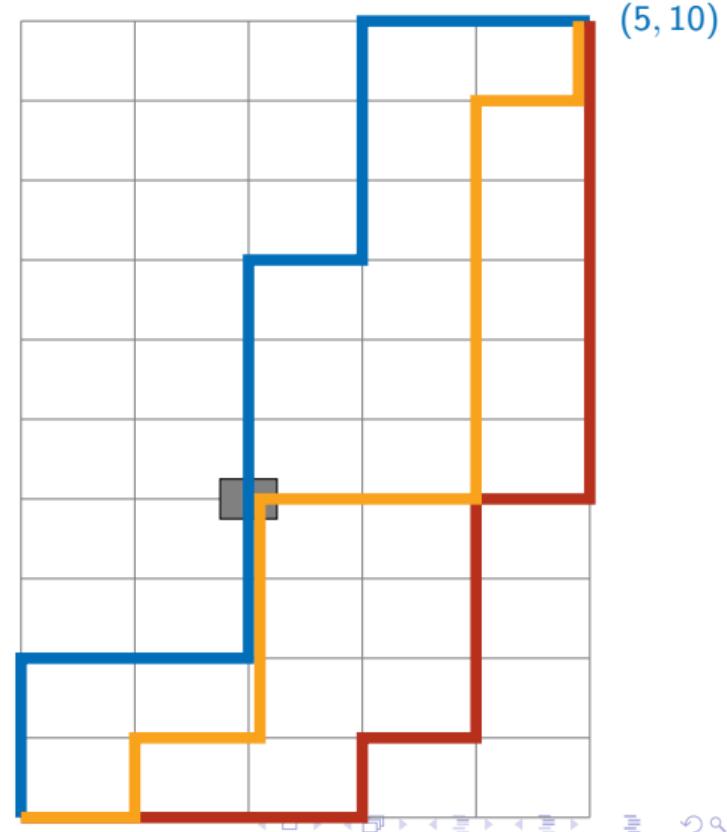
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$



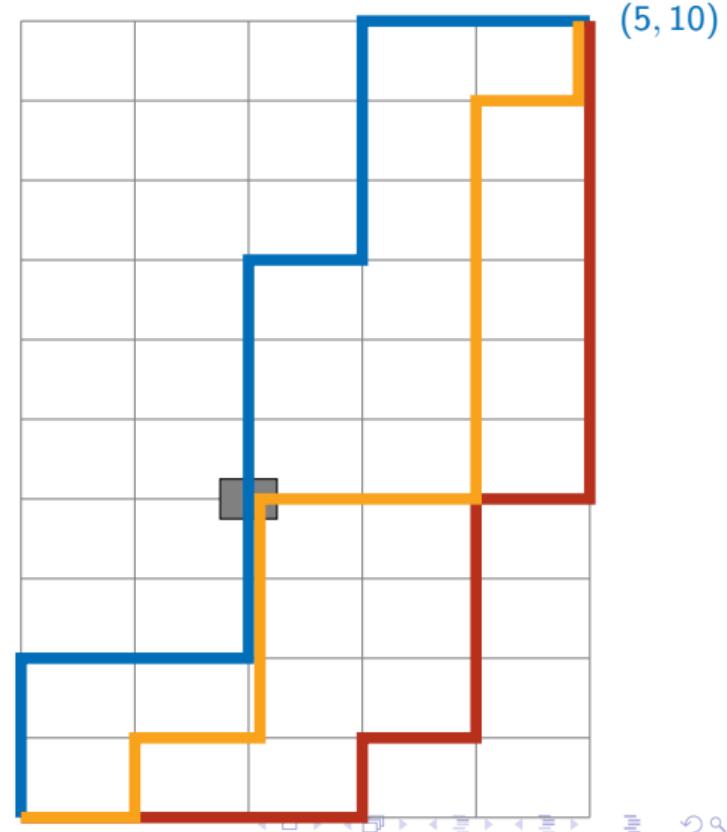
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$
- Every path via $(2, 4)$ combines a path from $(0, 0)$ to $(2, 4)$ with a path from $(2, 4)$ to $(5, 10)$
 - Count these separately
 - $\binom{2+4}{2} = 15$ paths $(0, 0)$ to $(2, 4)$
 - $\binom{3+6}{3} = 84$ paths $(2, 4)$ to $(5, 10)$



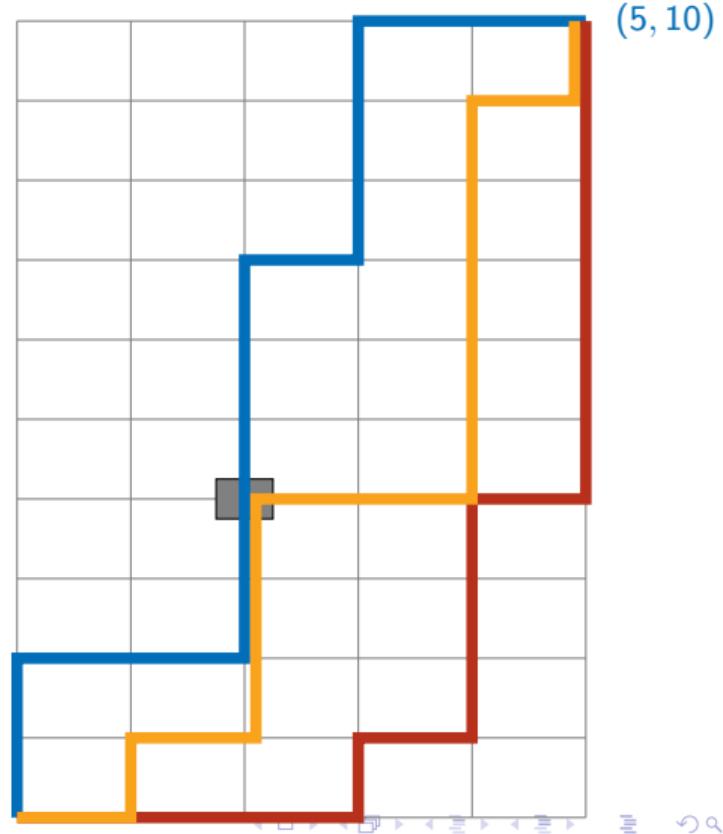
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$
- Every path via $(2, 4)$ combines a path from $(0, 0)$ to $(2, 4)$ with a path from $(2, 4)$ to $(5, 10)$
 - Count these separately
 - $\binom{2+4}{2} = 15$ paths $(0, 0)$ to $(2, 4)$
 - $\binom{3+6}{3} = 84$ paths $(2, 4)$ to $(5, 10)$
- $15 \times 84 = 1260$ paths via $(2, 4)$



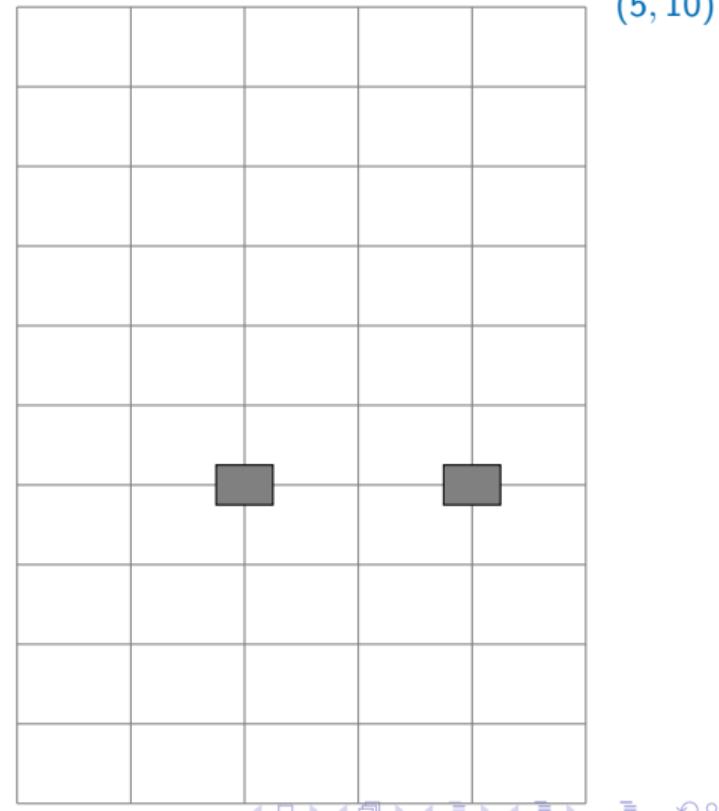
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$
- Every path via $(2, 4)$ combines a path from $(0, 0)$ to $(2, 4)$ with a path from $(2, 4)$ to $(5, 10)$
 - Count these separately
 - $\binom{2+4}{2} = 15$ paths $(0, 0)$ to $(2, 4)$
 - $\binom{3+6}{3} = 84$ paths $(2, 4)$ to $(5, 10)$
- $15 \times 84 = 1260$ paths via $(2, 4)$
- $3003 - 1260 = 1743$ valid paths avoiding $(2, 4)$



More holes

- What if two intersections are blocked?

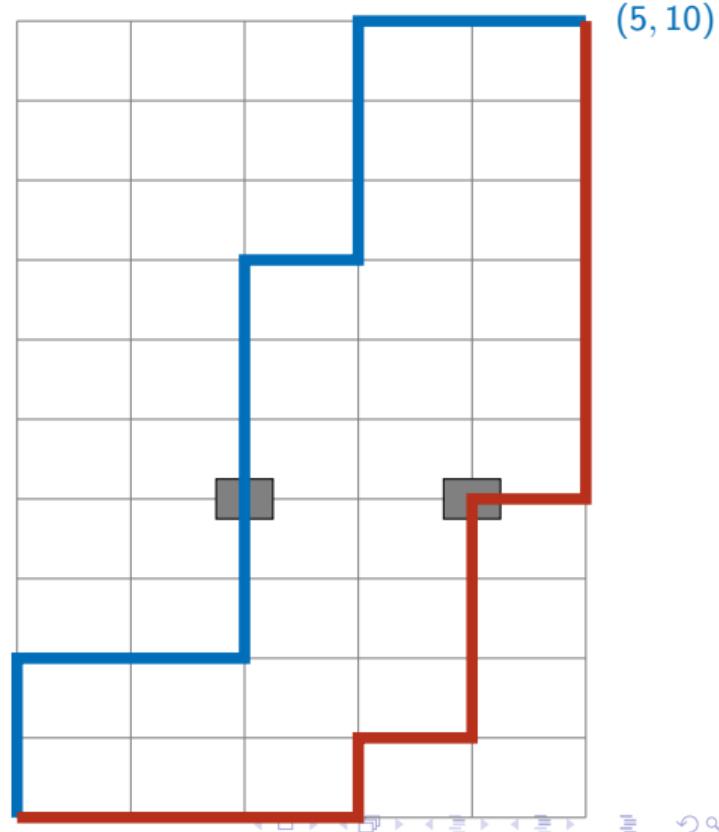


$(5, 10)$

$(0, 0)$

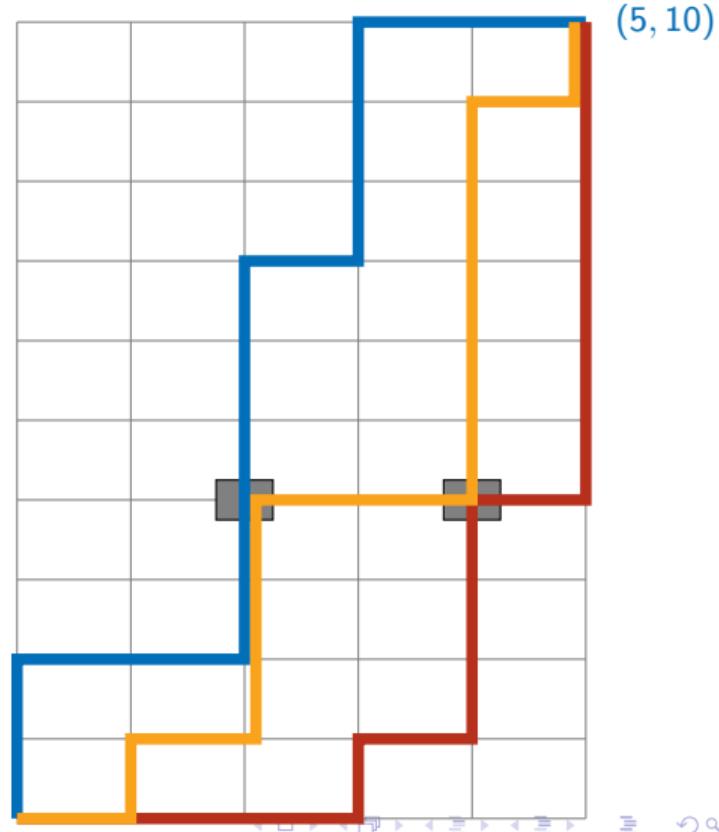
More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$



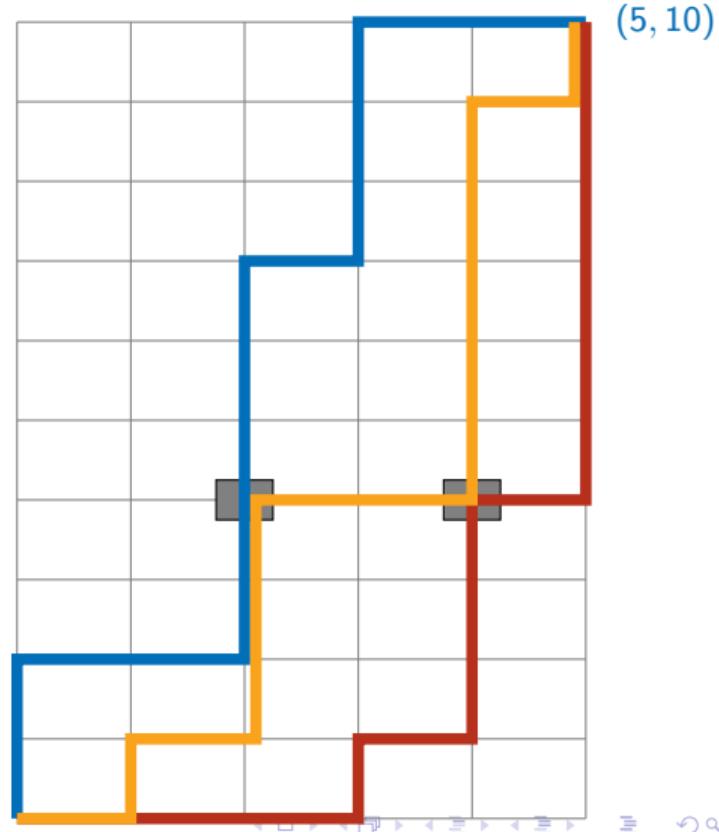
More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$
 - Some paths are counted twice



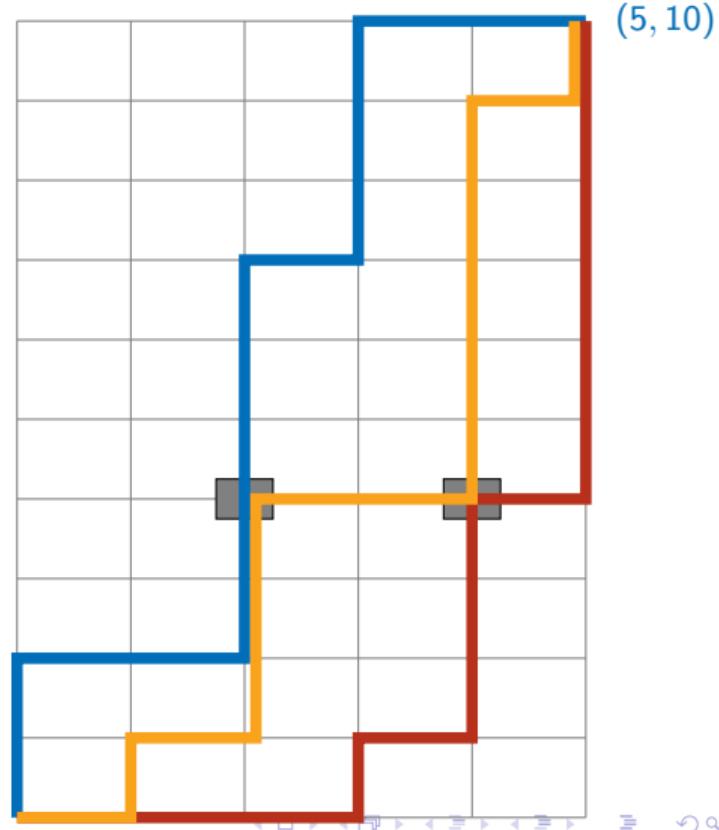
More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$
 - Some paths are counted twice
- Add back the paths that pass through both holes



More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$
 - Some paths are counted twice
- Add back the paths that pass through both holes
- Inclusion-exclusion — counting is messy



Inductive formulation

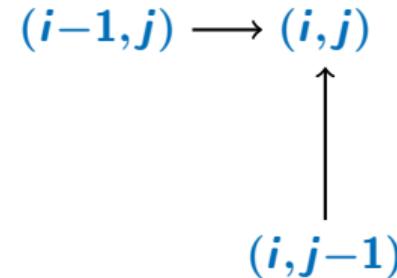
- How can a path reach (i, j)

Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$

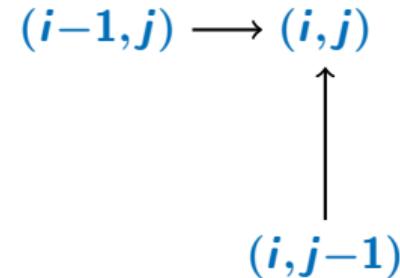
Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$



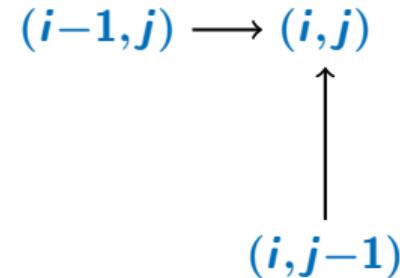
Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
- Each path to these neighbours extends to a unique path to (i, j)



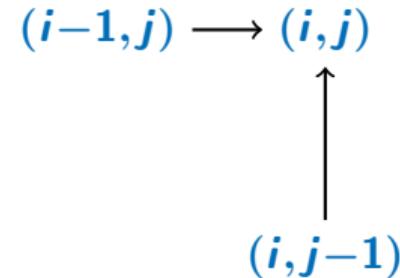
Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
 - $P(i, j) = P(i - 1, j) + P(i, j - 1)$



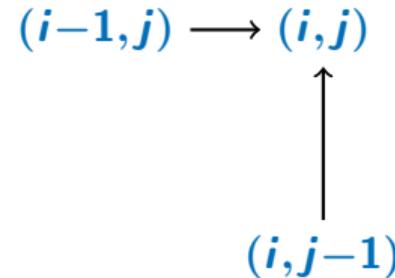
Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
 - $P(i, j) = P(i - 1, j) + P(i, j - 1)$
 - $P(0, 0) = 1$ — base case



Inductive formulation

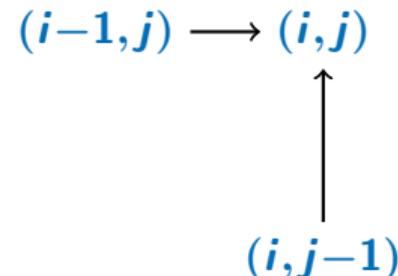
- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
 - $P(i, j) = P(i - 1, j) + P(i, j - 1)$
 - $P(0, 0) = 1$ — base case
 - $P(i, 0) = P(i - 1, 0)$ — bottom row



$(i, j-1)$

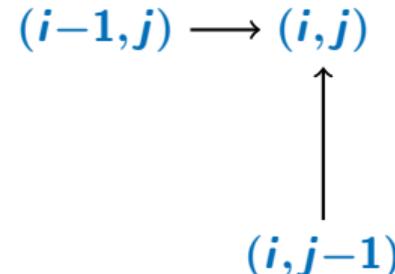
Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
 - $P(i, j) = P(i - 1, j) + P(i, j - 1)$
 - $P(0, 0) = 1$ — base case
 - $P(i, 0) = P(i - 1, 0)$ — bottom row
 - $P(0, j) = P(0, j - 1)$ — left column



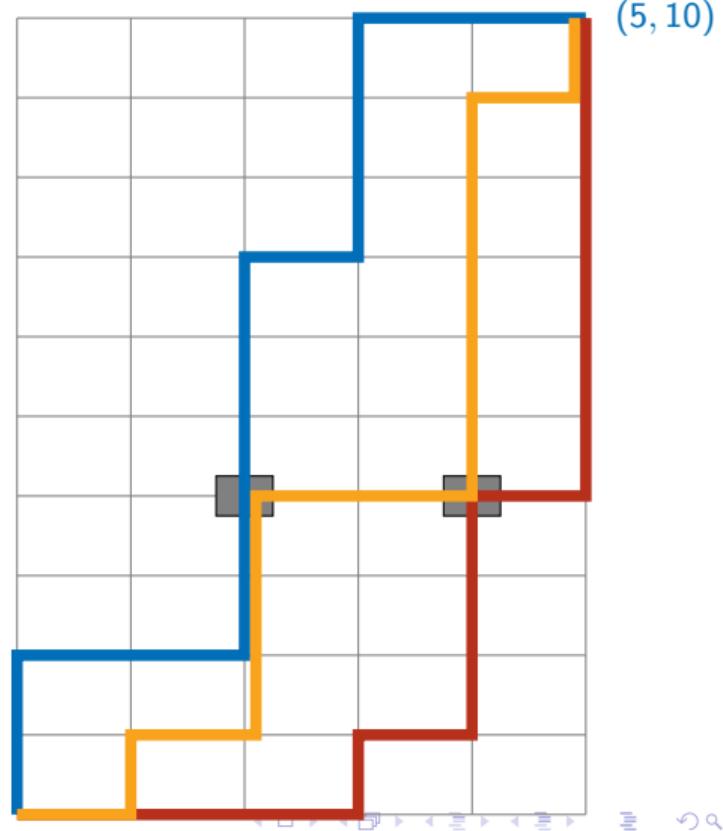
Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
 - $P(i, j) = P(i - 1, j) + P(i, j - 1)$
 - $P(0, 0) = 1$ — base case
 - $P(i, 0) = P(i - 1, 0)$ — bottom row
 - $P(0, j) = P(0, j - 1)$ — left column
- $P(i, j) = 0$ if there is a hole at (i, j)



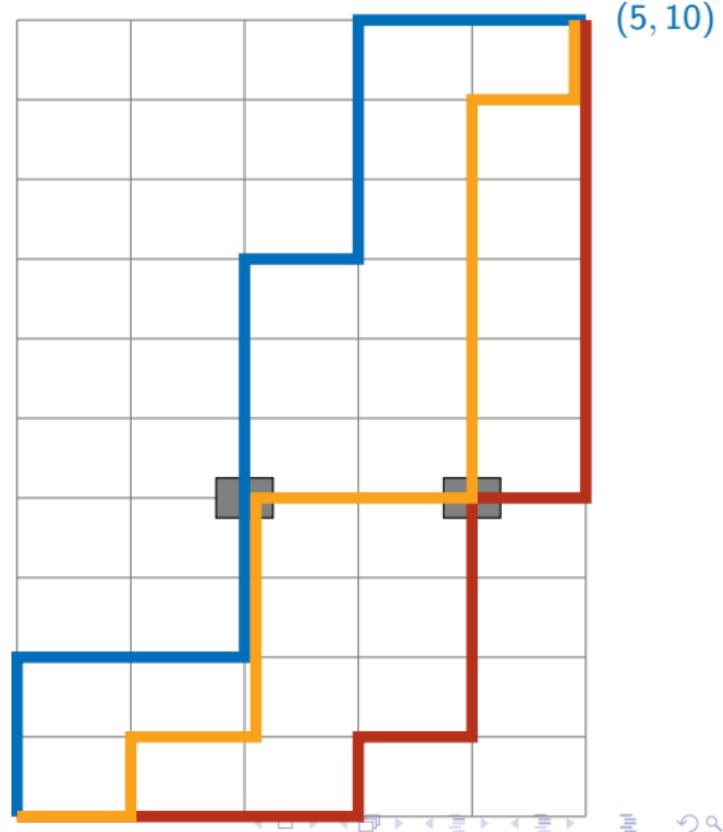
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly



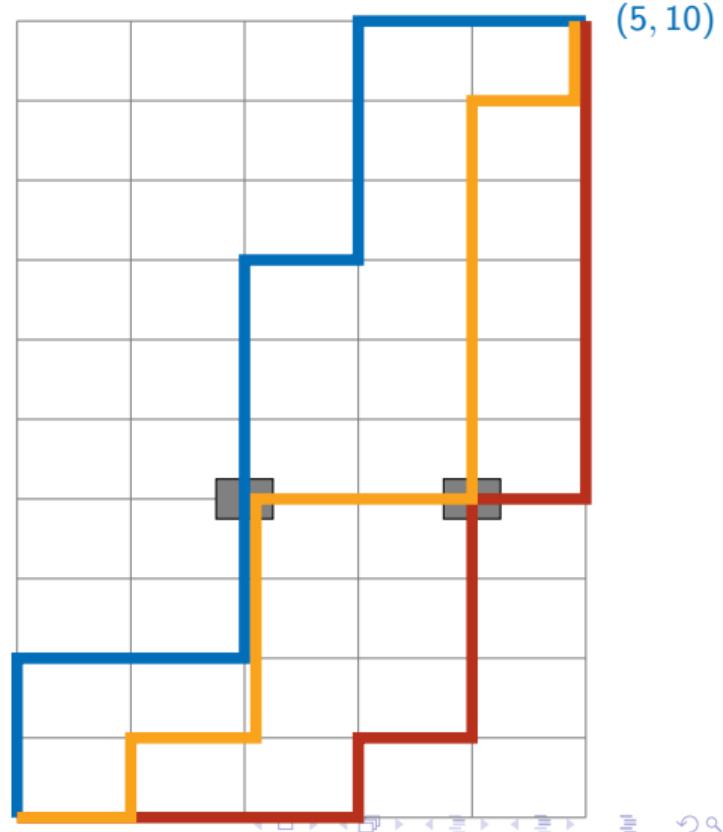
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
 - $P(5, 10)$ requires $P(4, 10), P(5, 9)$



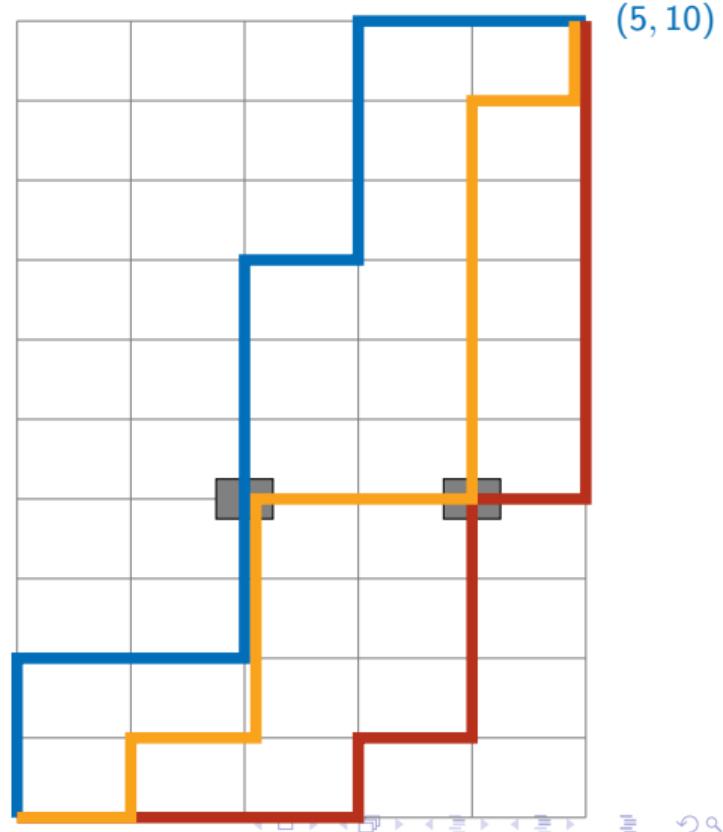
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
 - $P(5, 10)$ requires $P(4, 10), P(5, 9)$
 - Both $P(4, 10), P(5, 9)$ require $P(4, 9)$



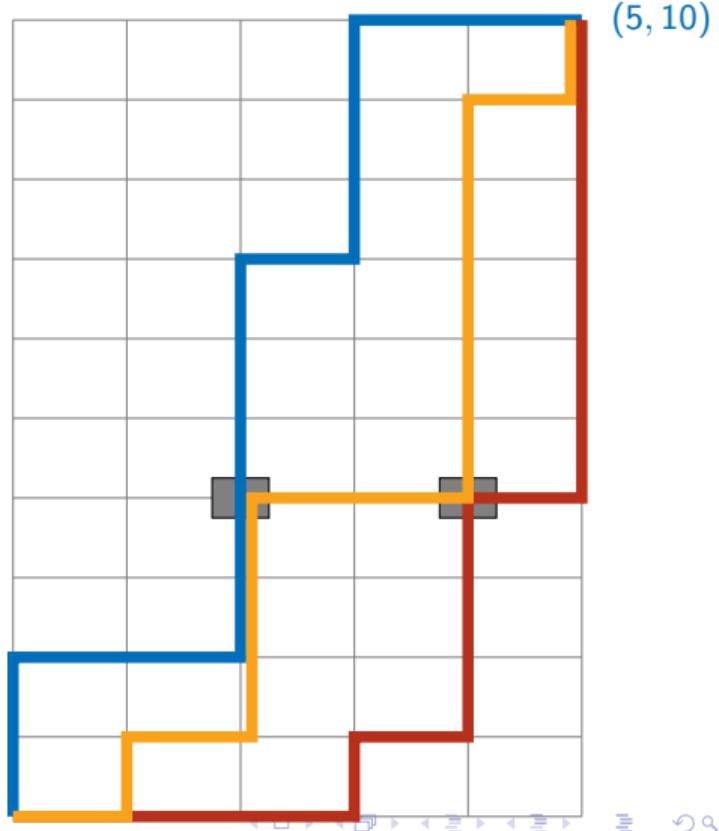
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
 - $P(5, 10)$ requires $P(4, 10), P(5, 9)$
 - Both $P(4, 10), P(5, 9)$ require $P(4, 9)$
- Use memoization . . .



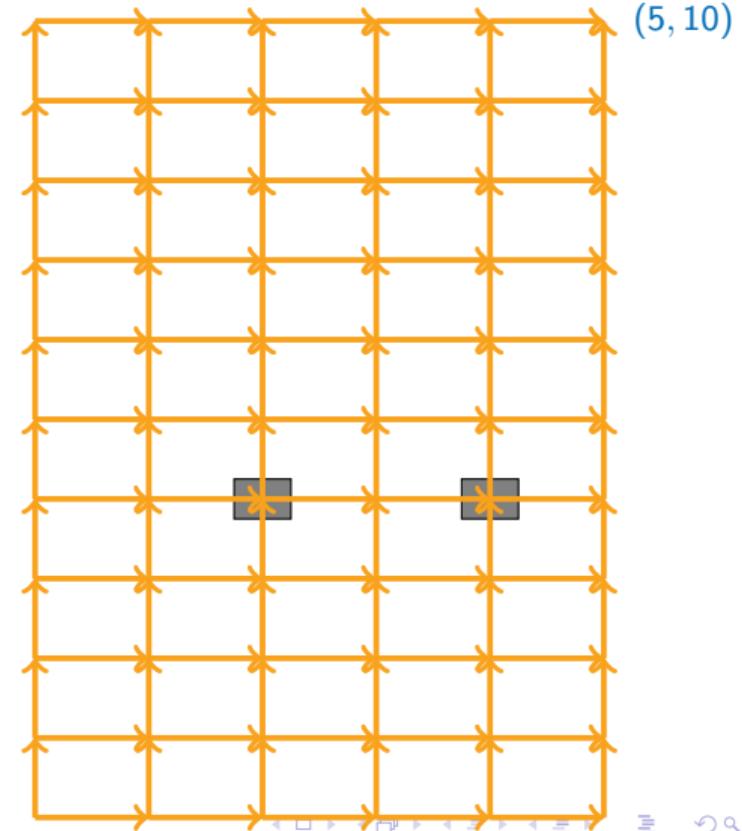
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
 - $P(5, 10)$ requires $P(4, 10), P(5, 9)$
 - Both $P(4, 10), P(5, 9)$ require $P(4, 9)$
- Use memoization ...
- ... or find a suitable order to compute the subproblems



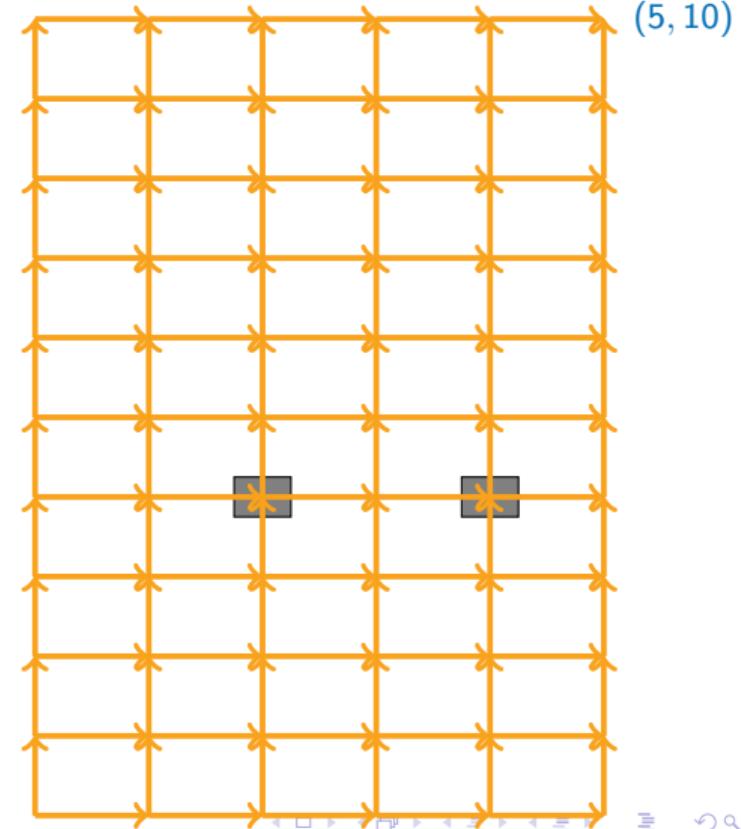
Dynamic programming

- Identify DAG structure



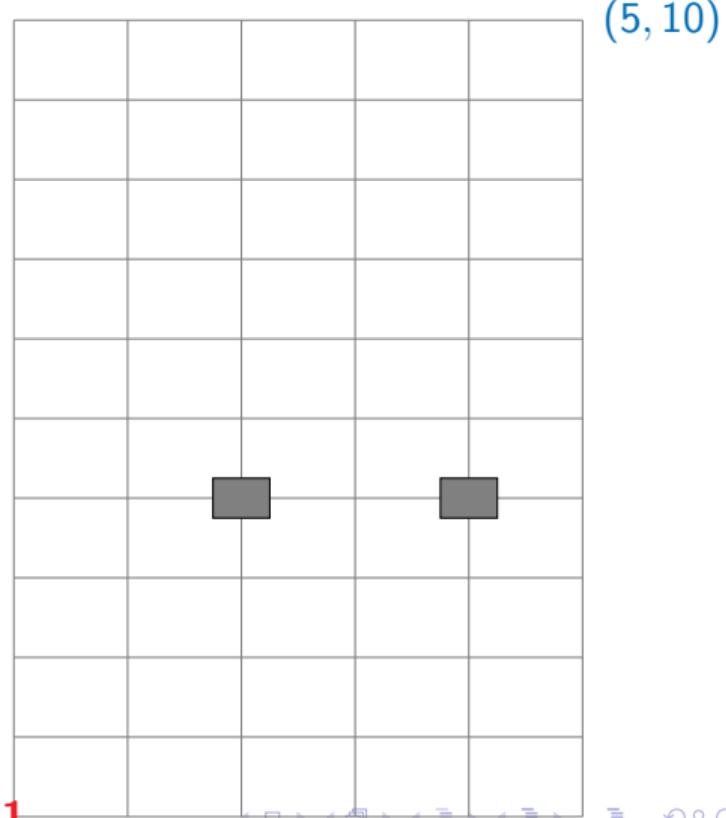
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies



Dynamic programming

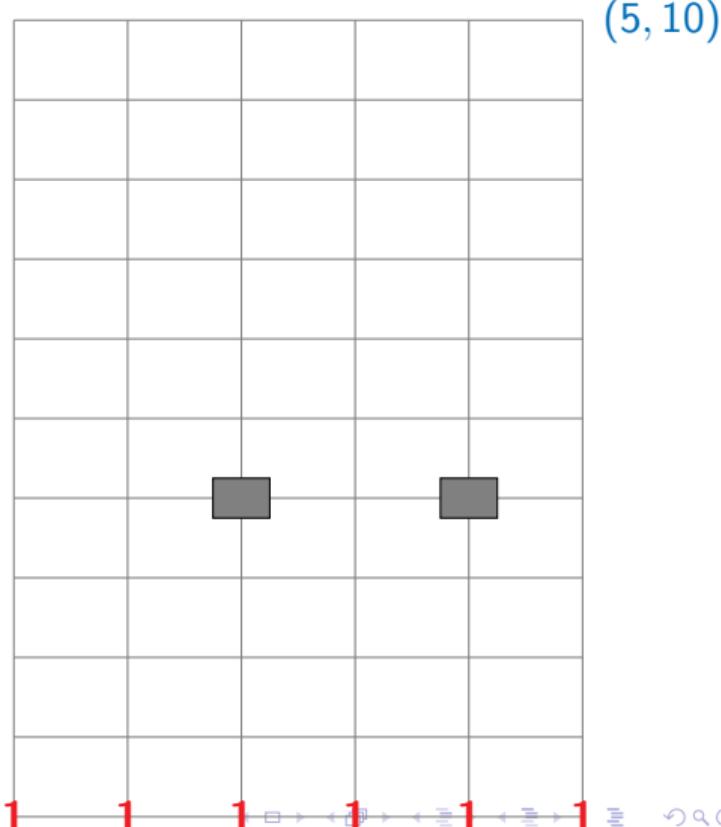
- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$



$(0, 0)$ 1

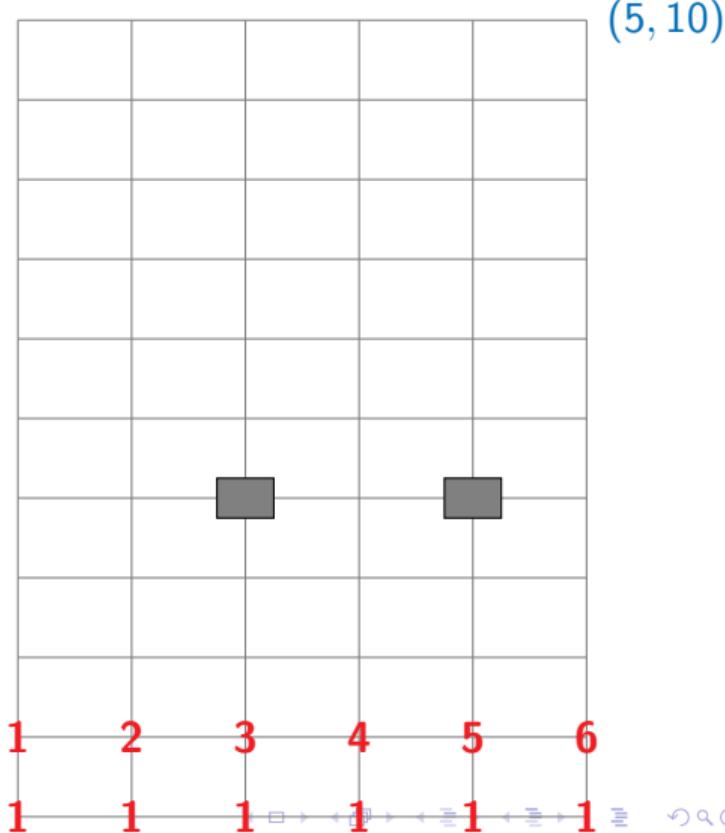
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



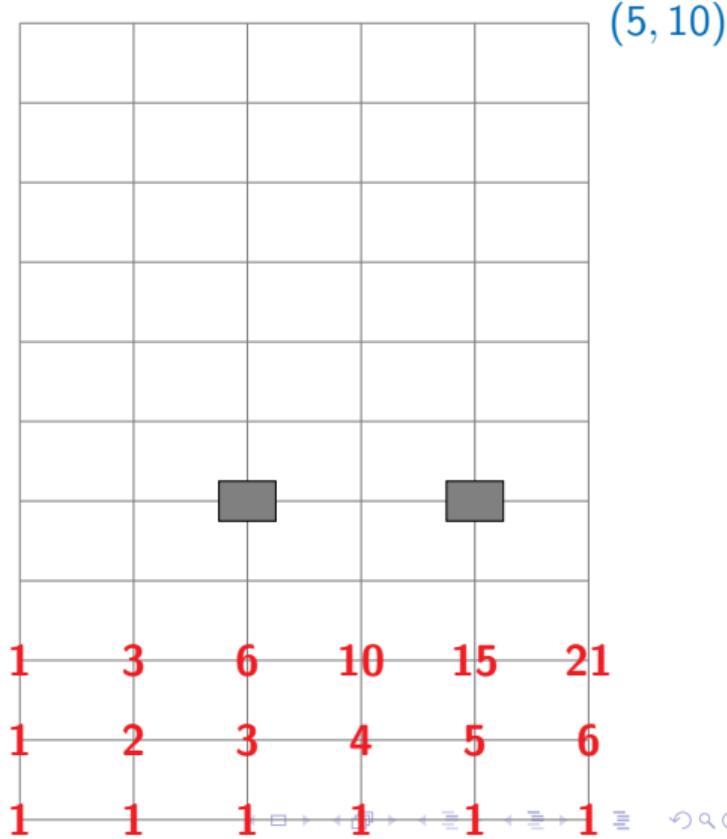
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



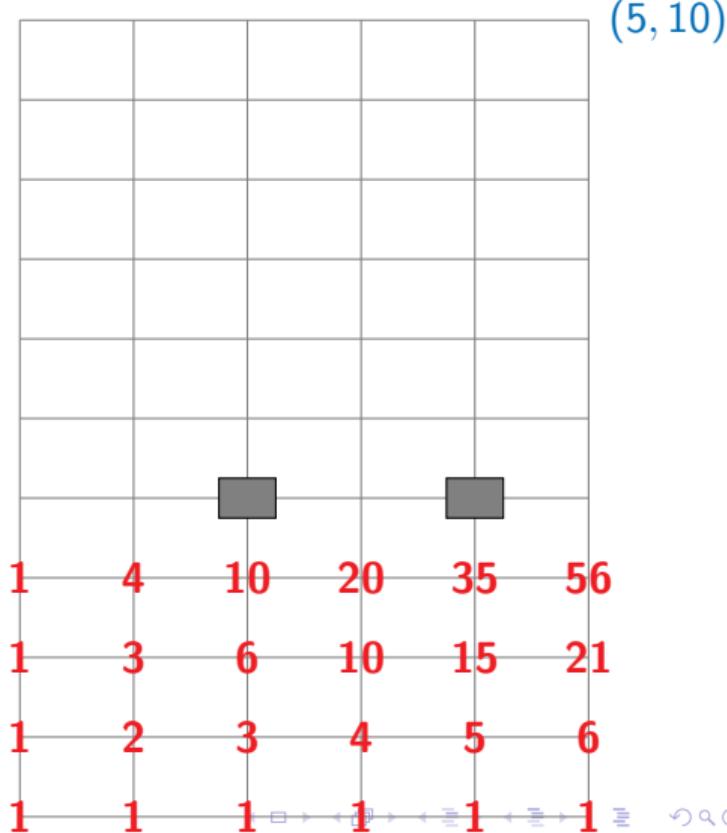
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



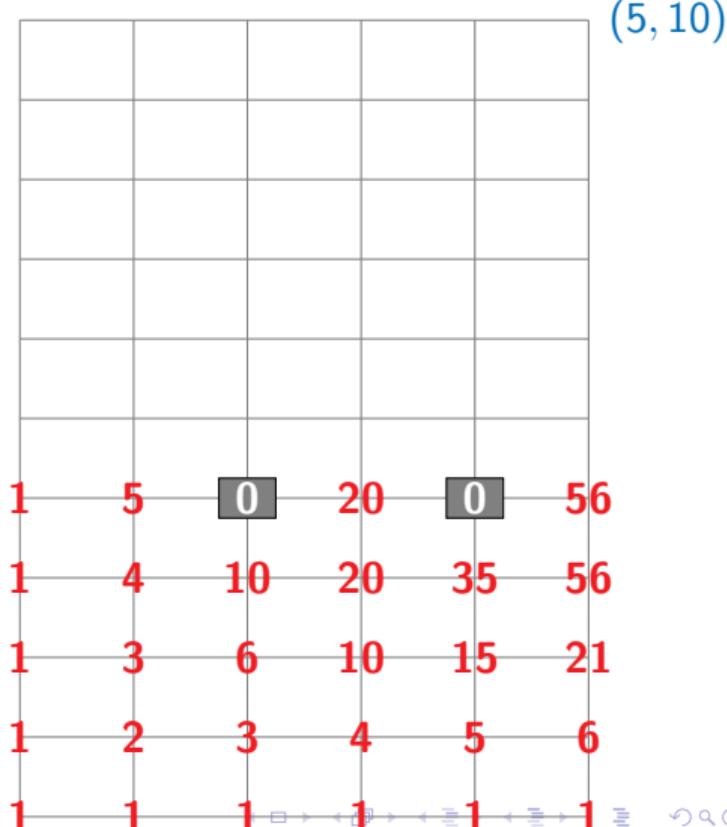
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



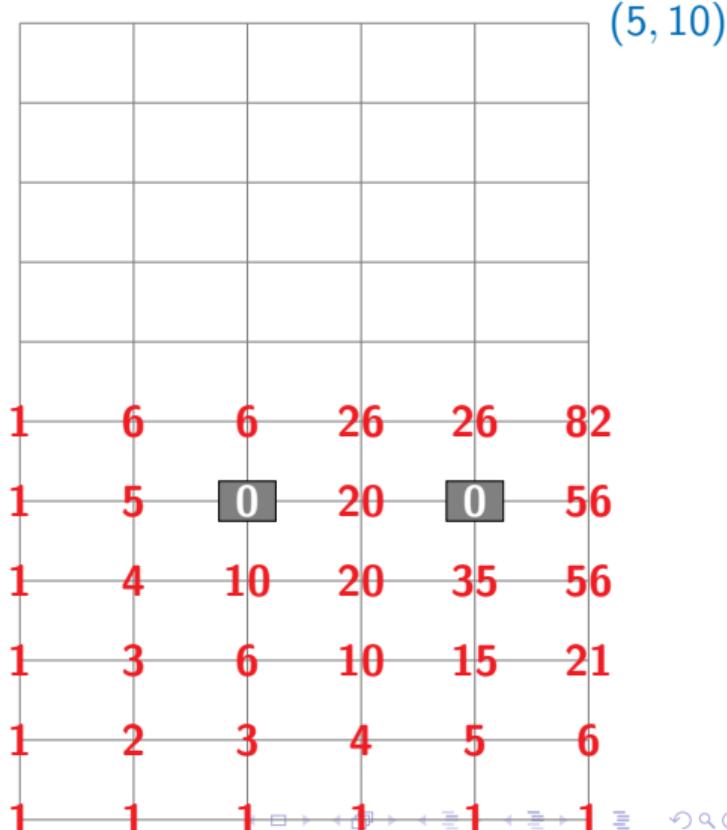
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row

1	11	51	181	526	135	8	5, 10
1	10	40	130	345	832		
1	9	30	90	215	487		
1	8	21	60	125	272		
1	7	13	39	65	147		
1	6	6	26	26	82		
1	5	0	20	0	56		
1	4	10	20	35	56		
1	3	6	10	15	21		
1	2	3	4	5	6		
1	1	1	1	1	1		

(0, 0)

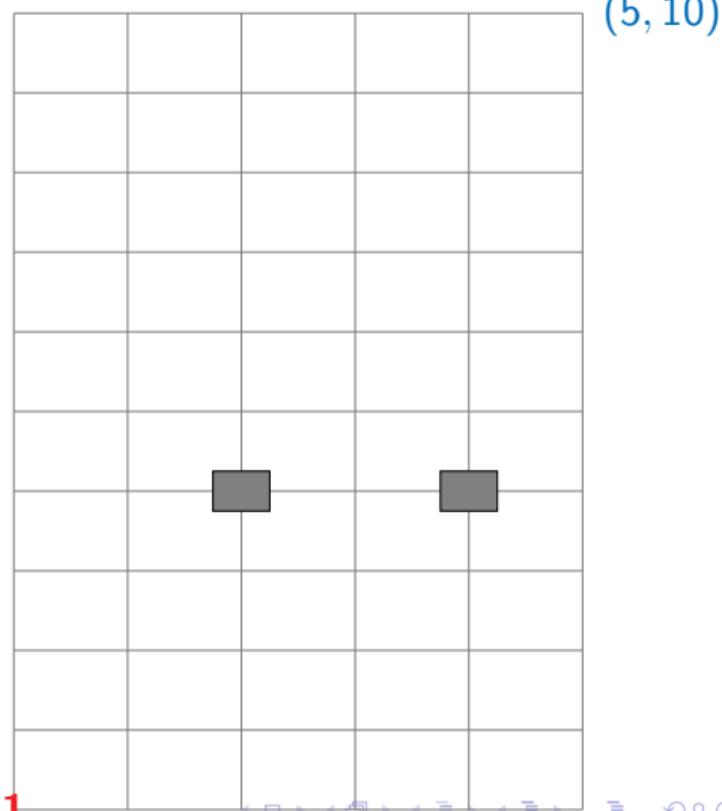
Grid Paths

PDSA using Python Week 9

10 / 13

Dynamic programming

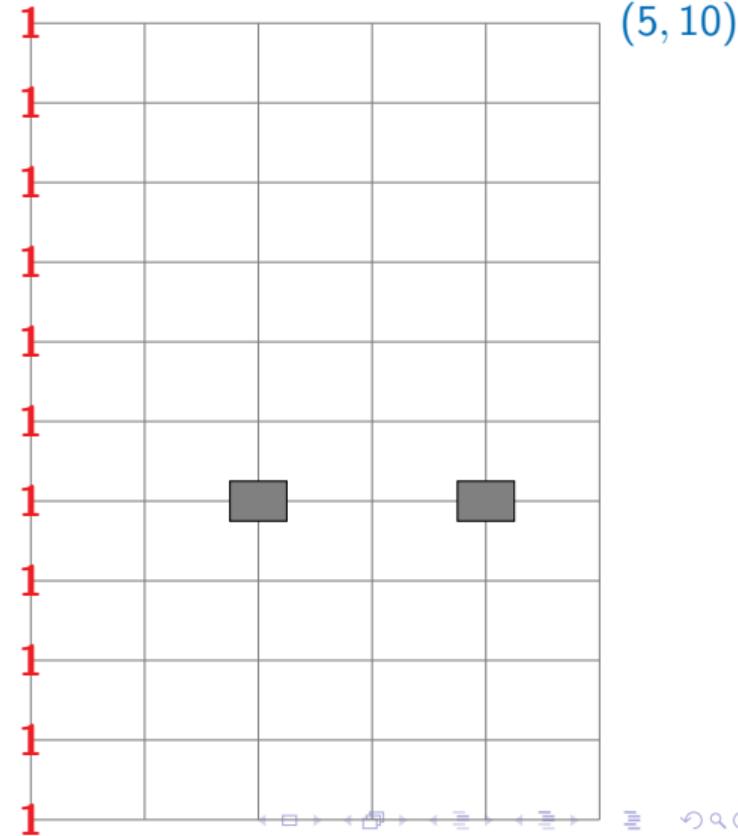
- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column



$(0, 0)$ 1

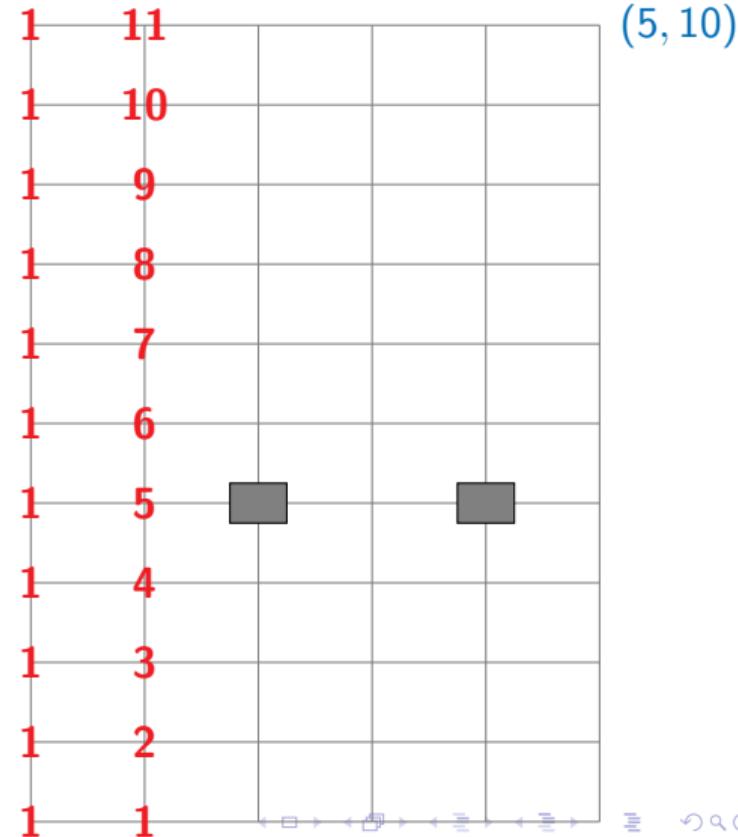
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column



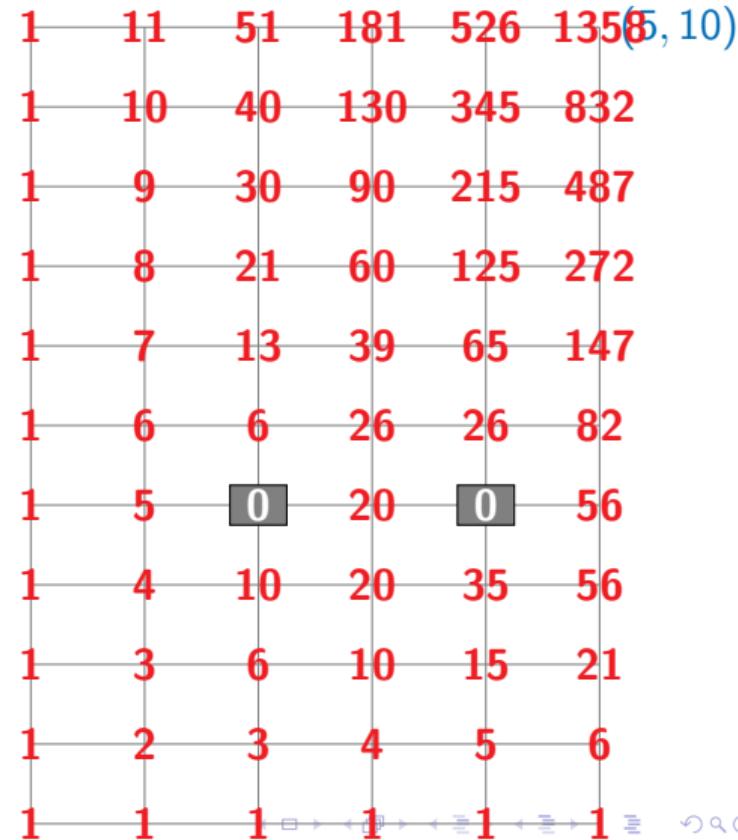
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column



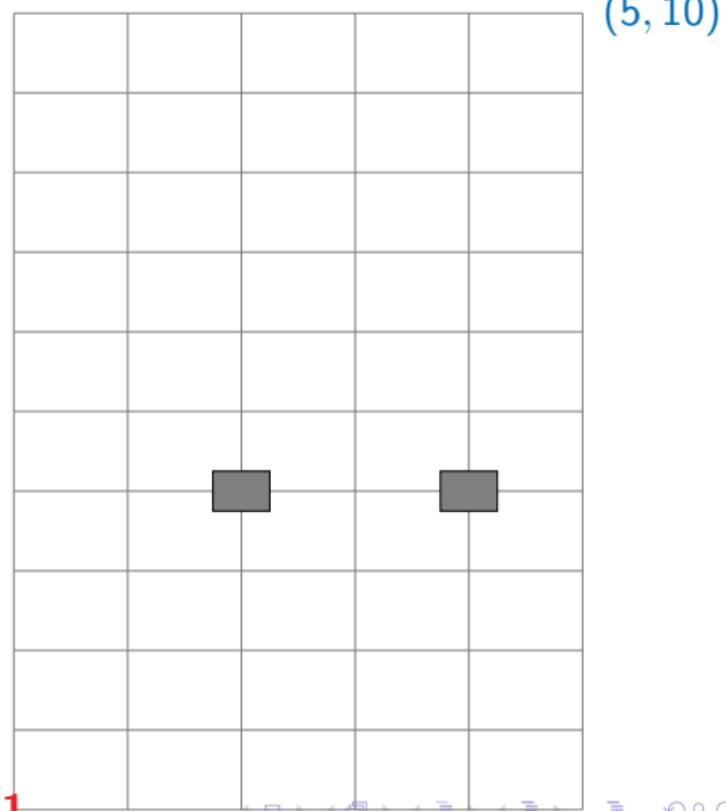
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column



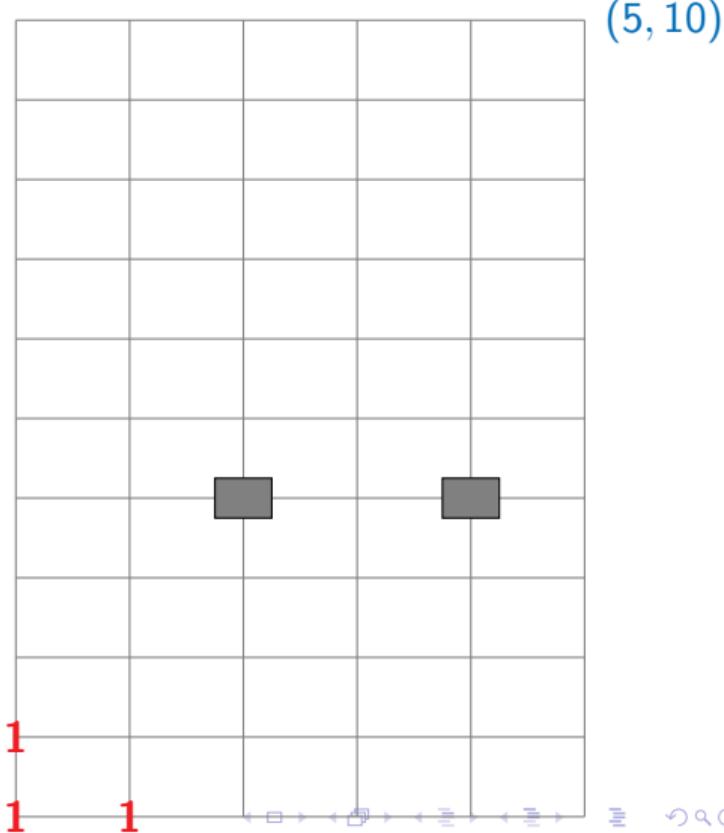
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



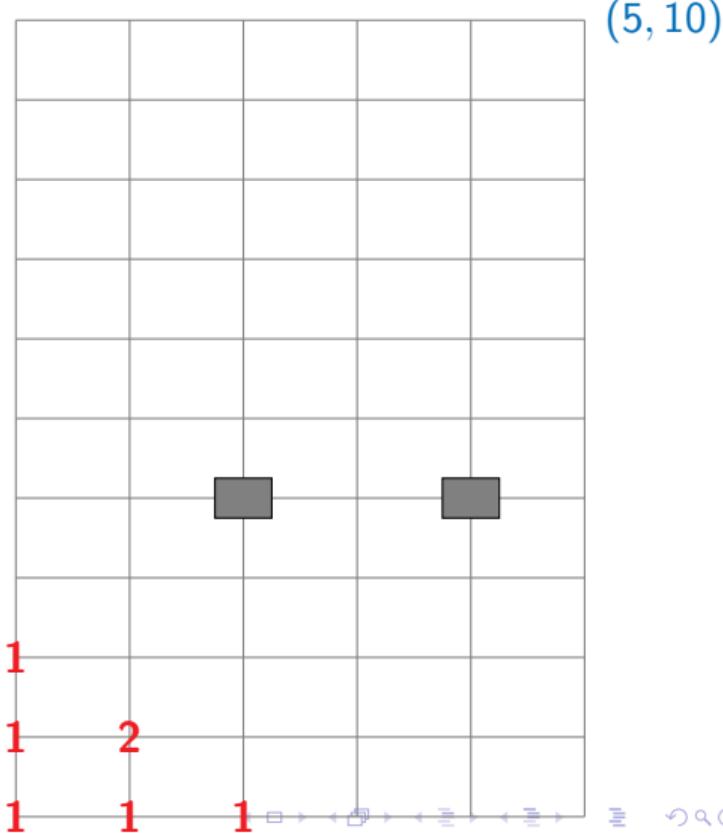
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



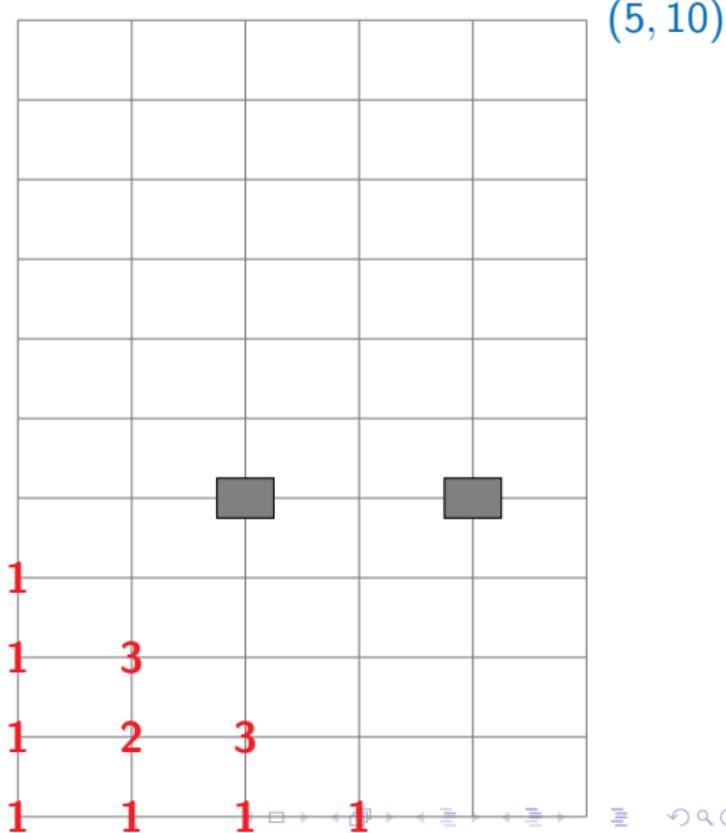
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



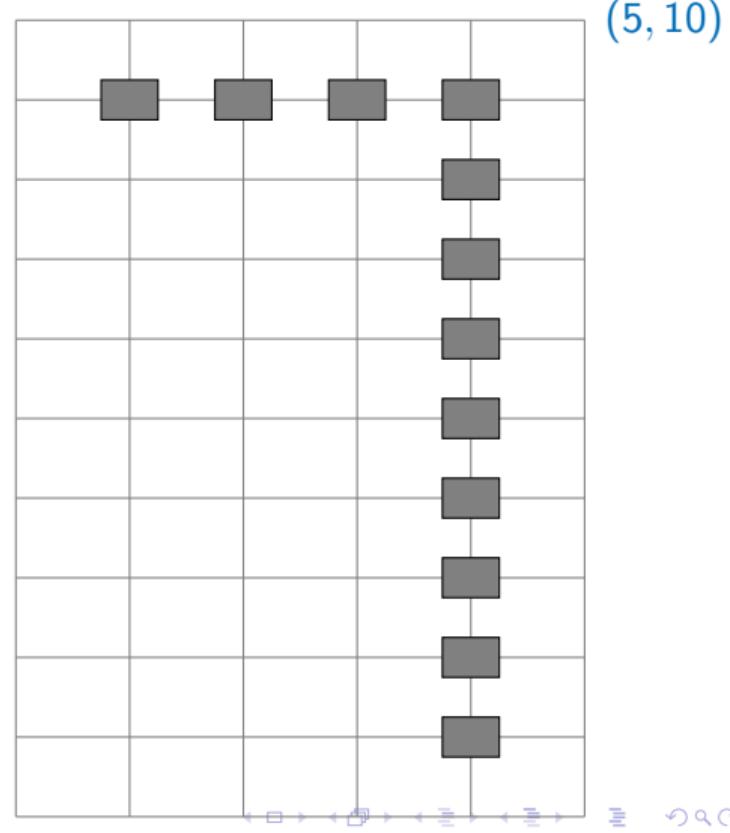
Dynamic programming

- Identify DAG structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



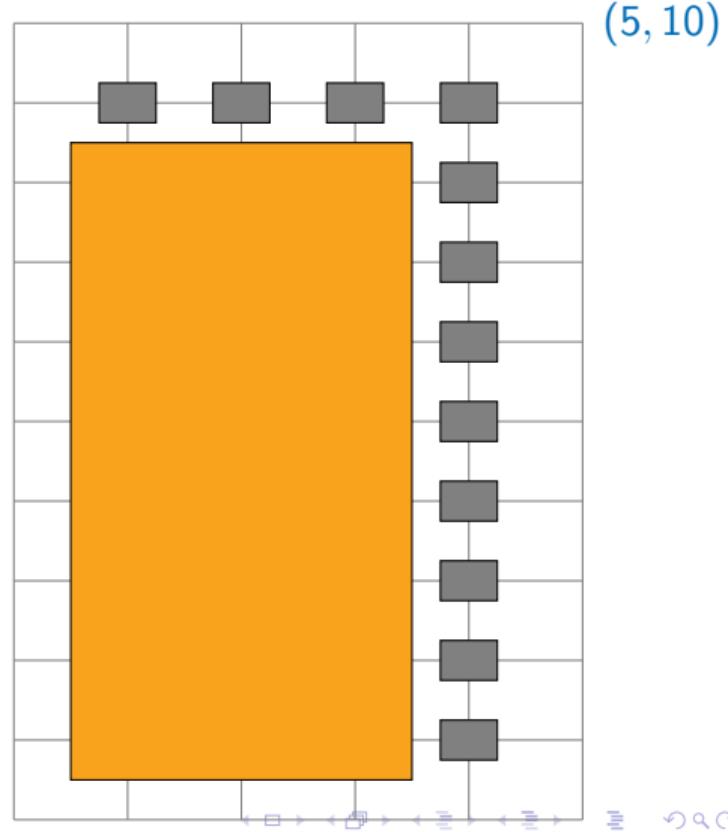
Memoization vs dynamic programming

- Barrier of holes just inside the border



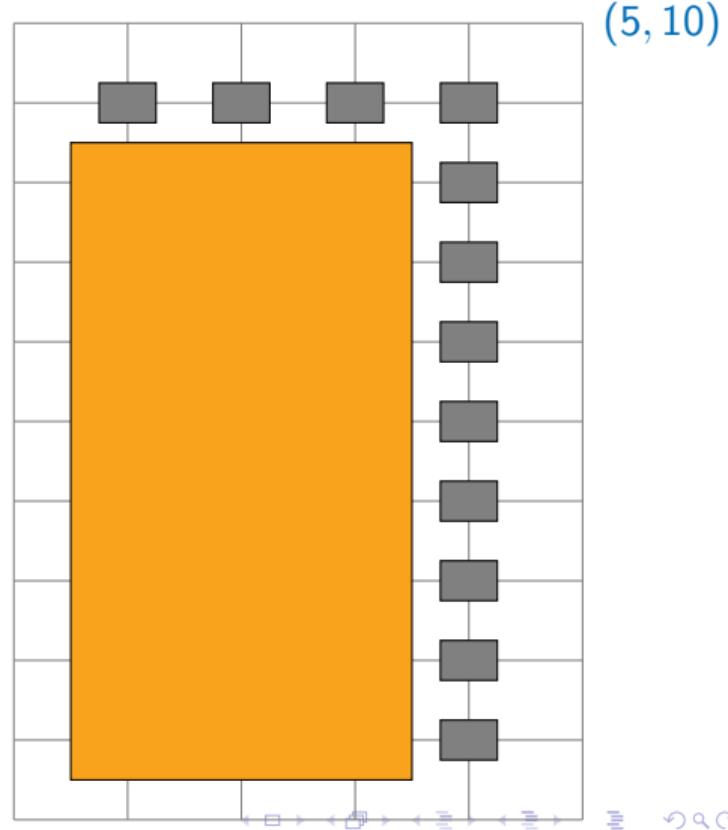
Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region



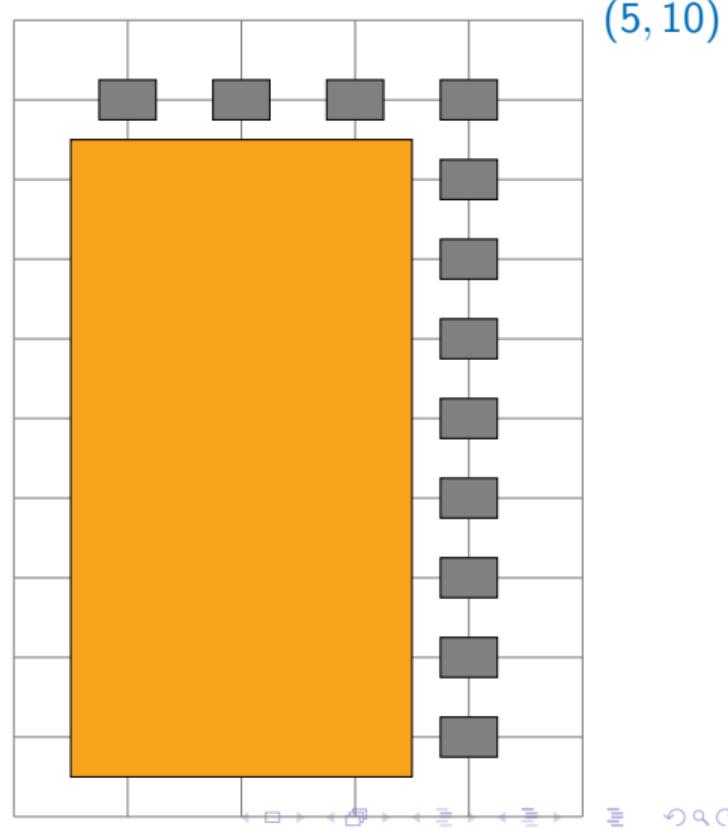
Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region
- Memo table has $O(m + n)$ entries



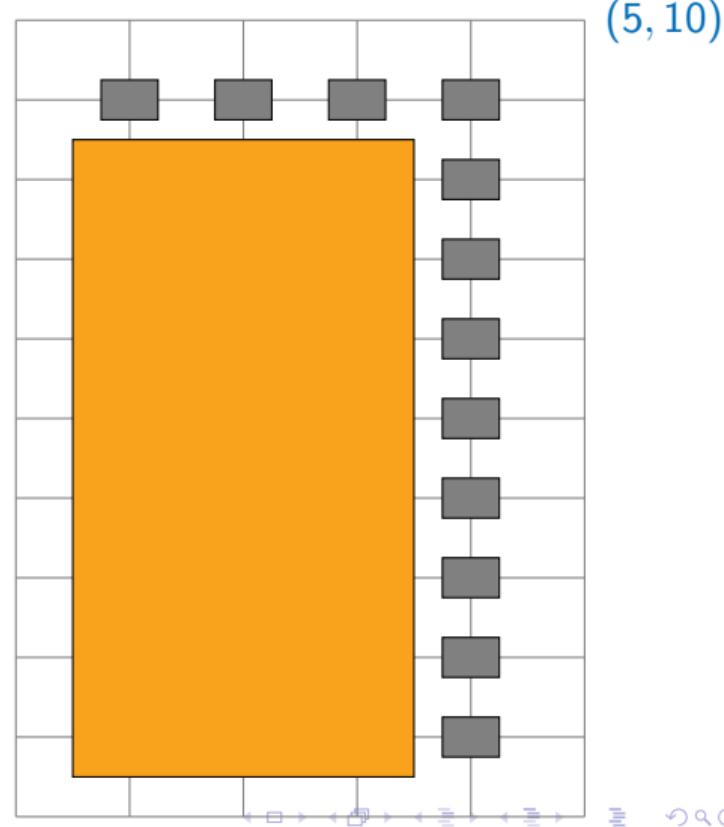
Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region
- Memo table has $O(m + n)$ entries
- Dynamic programming blindly fills all mn cells of the table



Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region
- Memo table has $O(m + n)$ entries
- Dynamic programming blindly fills all mn cells of the table
- Tradeoff between recursion and iteration
 - “Wasteful” dynamic programming still better in general



Common subwords and subsequences

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 9

Longest common subword

- Given two strings, find the (length of the) longest common subword
 - "secret", "secretary" — "secret", length 6
 - "bisect", "trisect" — "isect", length 5
 - "bisect", "secret" — "sec", length 3
 - "director", "secretary" — "ee", "re", length 2

Longest common subword

- Given two strings, find the (length of the) longest common subword
 - "secret", "secretary" — "secret", length 6
 - "bisect", "trisect" — "isect", length 5
 - "bisect", "secret" — "sec", length 3
 - "director", "secretary" — "ee", "re", length 2
- Formally
 - $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$

Longest common subword

- Given two strings, find the (length of the) longest common subword
 - "secret", "secretary" — "secret", length 6
 - "bisect", "trisect" — "isect", length 5
 - "bisect", "secret" — "sec", length 3
 - "director", "secretary" — "ee", "re", length 2
- Formally
 - $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
 - Common subword of length k — for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$

Longest common subword

- Given two strings, find the (length of the) longest common subword
 - "secret", "secretary" — "secret", length 6
 - "bisect", "trisect" — "isect", length 5
 - "bisect", "secret" — "sec", length 3
 - "director", "secretary" — "ee", "re", length 2
- Formally
 - $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
 - Common subword of length k — for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
 - Find the largest such k — length of the longest common subword

Brute force

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$

Brute force

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- Try every pair of starting positions i in u , j in v
 - Match $(a_i, b_j), (a_{i+1}, b_{j+1}), \dots$ as far as possible
 - Keep track of longest match

Brute force

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- Try every pair of starting positions i in u , j in v
 - Match $(a_i, b_j), (a_{i+1}, b_{j+1}), \dots$ as far as possible
 - Keep track of longest match
- Assuming $m > n$, this is $O(mn^2)$
 - mn pairs of starting positions
 - From each starting position, scan could be $O(n)$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_i a_{i+1} \dots a_{m-1}$, $b_j b_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $a_i = b_j$, $LCW(i, j) = 1 + LCW(i+1, j+1)$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_i a_{i+1} \dots a_{m-1}$, $b_j b_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $a_i = b_j$, $LCW(i, j) = 1 + LCW(i+1, j+1)$
 - Base case: $LCW(m, n) = 0$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_i a_{i+1} \dots a_{m-1}$, $b_j b_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $a_i = b_j$, $LCW(i, j) = 1 + LCW(i+1, j+1)$
 - Base case: $LCW(m, n) = 0$
 - In general, $LCW(i, n) = 0$ for all $0 \leq i \leq m$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_i a_{i+1} \dots a_{m-1}$, $b_j b_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $a_i = b_j$, $LCW(i, j) = 1 + LCW(i+1, j+1)$
 - Base case: $LCW(m, n) = 0$
 - In general, $LCW(i, n) = 0$ for all $0 \leq i \leq m$
 - In general, $LCW(m, j) = 0$ for all $0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCW(i, j)$, for
 $0 \leq i \leq m, 0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

Subproblem dependency

- Subproblems are $LCW(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCW(i,j)$ depends on $LCW(i+1,j+1)$

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

```
graph TD; R1(( )) --> R2(( )); R2(( )) --> R3(( )); R3(( )) --> R4(( ));
```

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b								0
1	i								0
2	s								0
3	e								0
4	c								0
5	t								0
6	•								0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b						0	0
1	i						0	0
2	s						0	0
3	e						0	0
4	c						0	0
5	t						1	0
6	•						0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b					0	0	0
1	i					0	0	0
2	s					0	0	0
3	e					1	0	0
4	c					0	0	0
5	t					0	1	0
6	•					0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b				0	0	0	0
1	i				0	0	0	0
2	s				0	0	0	0
3	e				0	1	0	0
4	c				0	0	0	0
5	t				0	0	1	0
6	•				0	0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b				0	0	0	0
1	i				0	0	0	0
2	s				0	0	0	0
3	e				0	0	1	0
4	c				1	0	0	0
5	t				0	0	0	1
6	•				0	0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	0	0	0	0	0	0	0
3	e	2	0	0	1	0	0	0
4	c	0	1	0	0	0	0	0
5	t	0	0	0	0	1	0	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

Reading off the solution

- Find entry (i, j) with largest LCW value

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

Reading off the solution

- Find entry (i, j) with largest LCW value
- Read off the actual subword diagonally

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

Reading off the solution

- Find entry (i, j) with largest LCW value
- Read off the actual subword diagonally

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	0	0	0	0	0	0	0	
1	i	0	0	0	0	0	0	0	
2	s	3	0	0	0	0	0	0	
3	e	0	2	0	0	1	0	0	
4	c	0	0	1	0	0	0	0	
5	t	0	0	0	0	0	1	0	
6	•	0	0	0	0	0	0	0	

Implementation

```
def LCW(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcw = np.zeros((m+1,n+1))

    maxlcw = 0

    for c in range(n-1,-1,-1):
        for r in range(m-1,-1,-1):
            if u[r] == v[c]:
                lcw[r,c] = 1 + lcw[r+1,c+1]
            else:
                lcw[r,c] = 0
            if lcw[r,c] > maxlcw:
                maxlcw = lcw[r,c]

    return(maxlcw)
```

Implementation

```
def LCW(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcw = np.zeros((m+1,n+1))  
  
    maxlcw = 0  
  
    for c in range(n-1,-1,-1):  
        for r in range(m-1,-1,-1):  
            if u[r] == v[c]:  
                lcw[r,c] = 1 + lcw[r+1,c+1]  
            else:  
                lcw[r,c] = 0  
            if lcw[r,c] > maxlcw:  
                maxlcw = lcw[r,c]  
  
    return(maxlcw)
```

Complexity

Implementation

```
def LCW(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcw = np.zeros((m+1,n+1))  
  
    maxlcw = 0  
  
    for c in range(n-1,-1,-1):  
        for r in range(m-1,-1,-1):  
            if u[r] == v[c]:  
                lcw[r,c] = 1 + lcw[r+1,c+1]  
            else:  
                lcw[r,c] = 0  
            if lcw[r,c] > maxlcw:  
                maxlcw = lcw[r,c]  
  
    return(maxlcw)
```

Complexity

- Recall that brute force was $O(mn^2)$

Implementation

```
def LCW(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcw = np.zeros((m+1,n+1))

    maxlcw = 0

    for c in range(n-1,-1,-1):
        for r in range(m-1,-1,-1):
            if u[r] == v[c]:
                lcw[r,c] = 1 + lcw[r+1,c+1]
            else:
                lcw[r,c] = 0
            if lcw[r,c] > maxlcw:
                maxlcw = lcw[r,c]

    return(maxlcw)
```

Complexity

- Recall that brute force was $O(mn^2)$
- Inductive solution is $O(mn)$, using dynamic programming or memoization

Implementation

```
def LCW(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcw = np.zeros((m+1,n+1))

    maxlcw = 0

    for c in range(n-1,-1,-1):
        for r in range(m-1,-1,-1):
            if u[r] == v[c]:
                lcw[r,c] = 1 + lcw[r+1,c+1]
            else:
                lcw[r,c] = 0
            if lcw[r,c] > maxlcw:
                maxlcw = lcw[r,c]

    return(maxlcw)
```

Complexity

- Recall that brute force was $O(mn^2)$
- Inductive solution is $O(mn)$, using dynamic programming or memoization
 - Fill a table of size $O(mn)$
 - Each table entry takes constant time to compute

Longest common subsequence

- Subsequence — can drop some letters in between
- Given two strings, find the (length of the) longest common subsequence
 - "secret", "secretary" —
"secret", length 6
 - "bisect", "trisect" —
"isect", length 5
 - "bisect", "secret" —
"sect", length 4
 - "director", "secretary" —
"ectr", "retr", length 4

Longest common subsequence

- Subsequence — can drop some letters in between
- Given two strings, find the (length of the) longest common subsequence
 - "secret", "secretary" — "secret", length 6
 - "bisect", "trisect" — "isect", length 5
 - "bisect", "secret" — "sect", length 4
 - "director", "secretary" — "ectr", "retr", length 4
- LCS is the longest path connecting non-zero LCW entries, moving right/down

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Longest common subsequence

- Subsequence — can drop some letters in between
- Given two strings, find the (length of the) longest common subsequence
 - "secret", "secretary" — "secret", length 6
 - "bisect", "trisect" — "isect", length 5
 - "bisect", "secret" — "sect", length 4
 - "director", "secretary" — "ectr", "retr", length 4
- LCS is the longest path connecting non-zero LCW entries, moving right/down

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Applications

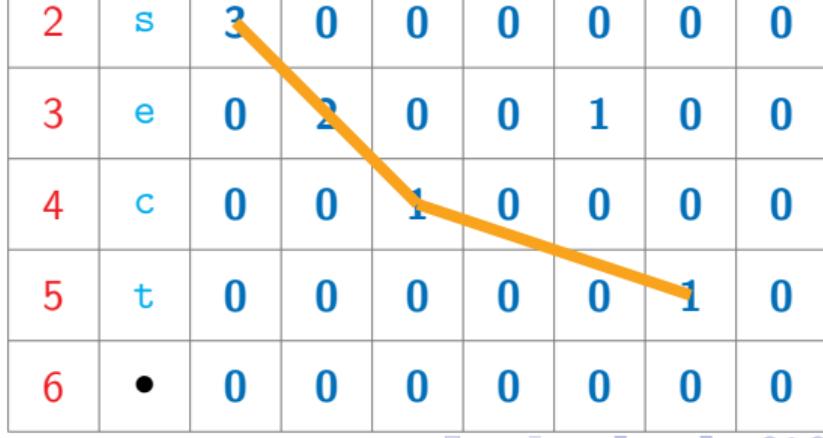
- Analyzing genes
 - DNA is a long string over A, T, G, C
 - Two species are similar if their DNA has long common subsequences

		0	1	2	3	4	5	6
		s	e	c	r	e	t	*
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	*	0	0	0	0	0	0	0

Applications

- Analyzing genes
 - DNA is a long string over A, T, G, C
 - Two species are similar if their DNA has long common subsequences
- `diff` command in Unix/Linux
 - Compares text files
 - Find the longest matching subsequence of lines
 - Each line of text is a “character”

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	0	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0	0
4	c	0	0	1	0	0	0	0	0
5	t	0	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0	0



Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS
- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS
- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS
 - Which one should we drop?

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS
- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS
 - Which one should we drop?
 - Solve $LCS(i, j+1)$ and $LCS(i+1, j)$ and take the maximum

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS
- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS
 - Which one should we drop?
 - Solve $LCS(i, j+1)$ and $LCS(i+1, j)$ and take the maximum
- Base cases as with LCW
 - $LCS(i, n) = 0$ for all $0 \leq i \leq m$
 - $LCS(m, j) = 0$ for all $0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

The diagram shows a 6x9 grid representing a dynamic programming table for the LCS problem. The columns are indexed from 0 to 6 (red numbers) and the rows are indexed from 0 to 6 (red numbers). The first row contains 's', 'e', 'c', 'r', 'e', 't', '•'. The second row contains 'b'. The third row contains 'i'. The fourth row contains 's'. The fifth row contains 'e'. The sixth row contains 'c'. The seventh row contains 't'. The eighth row contains '•'. Orange arrows point from cell (3,4) to cell (2,3) and from cell (3,4) to cell (4,3).

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							0
1	i							0
2	s							0
3	e							0
4	c							0
5	t							0
6	•							0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b						0	0
1	i						0	0
2	s						0	0
3	e						0	0
4	c						0	0
5	t						1	0
6	•						0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b					1	0	0
1	i					1	0	0
2	s					1	0	0
3	e					1	0	0
4	c					1	0	0
5	t					1	1	0
6	•					0	0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b				1	1	0	0
1	i				1	1	0	0
2	s				1	1	0	0
3	e				1	1	0	0
4	c				1	1	0	0
5	t				1	1	1	0
6	•				0	0	0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b			2	1	1	0	0
1	i			2	1	1	0	0
2	s			2	1	1	0	0
3	e			2	1	1	0	0
4	c			2	1	1	0	0
5	t			1	1	1	1	0
6	•			0	0	0	0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b		3	2	1	1	0	0
1	i		3	2	1	1	0	0
2	s		3	2	1	1	0	0
3	e		3	2	1	1	0	0
4	c		2	2	1	1	0	0
5	t		1	1	1	1	1	0
6	•		0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Trace back the path by which each entry was filled

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Trace back the path by which each entry was filled
- Each diagonal step is an element of LCS

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	3	2	1	1	0	0	
1	i	4	3	2	1	1	0	0	
2	s	4	3	2	1	1	0	0	
3	e	3	3	2	1	1	0	0	
4	c	2	2	2	1	1	0	0	
5	t	1	1	1	1	1	1	0	
6	•	0	0	0	0	0	0	0	

The diagram illustrates the trace-back path from the bottom-right cell (t) to the top-left cell (s) in a dynamic programming table for the LCS problem. The path follows a diagonal sequence of cells: (t, 5, 5) -> (c, 4, 4) -> (e, 3, 3) -> (s, 2, 2) -> (s, 1, 1) -> (e, 0, 0). The path is highlighted with orange arrows.

Implementation

```
def LCS(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcs = np.zeros((m+1,n+1))  
  
    for c in range(n-1,-1,-1):  
        for r in range(m-1,-1,-1):  
            if u[r] == v[c]:  
                lcs[r,c] = 1 + lcs[r+1,c+1]  
            else:  
                lcs[r,c] = max(lcs[r+1,c],  
                                lcs[r,c+1])  
    return(lcs[0,0])
```

Implementation

```
def LCS(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcs = np.zeros((m+1,n+1))  
  
    for c in range(n-1,-1,-1):  
        for r in range(m-1,-1,-1):  
            if u[r] == v[c]:  
                lcs[r,c] = 1 + lcs[r+1,c+1]  
            else:  
                lcs[r,c] = max(lcs[r+1,c],  
                                lcs[r,c+1])  
    return(lcs[0,0])
```

Complexity

Implementation

```
def LCS(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcs = np.zeros((m+1,n+1))  
  
    for c in range(n-1,-1,-1):  
        for r in range(m-1,-1,-1):  
            if u[r] == v[c]:  
                lcs[r,c] = 1 + lcs[r+1,c+1]  
            else:  
                lcs[r,c] = max(lcs[r+1,c],  
                                lcs[r,c+1])  
  
    return(lcs[0,0])
```

Complexity

- Again $O(mn)$, using dynamic programming or memoization

Implementation

```
def LCS(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcs = np.zeros((m+1,n+1))  
  
    for c in range(n-1,-1,-1):  
        for r in range(m-1,-1,-1):  
            if u[r] == v[c]:  
                lcs[r,c] = 1 + lcs[r+1,c+1]  
            else:  
                lcs[r,c] = max(lcs[r+1,c],  
                                lcs[r,c+1])  
  
    return(lcs[0,0])
```

Complexity

- Again $O(mn)$, using dynamic programming or memoization
 - Fill a table of size $O(mn)$
 - Each table entry takes constant time to compute

Edit Distance

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 9

Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”

Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- Edit operations to transform documents
 - Insert a character
 - Delete a character
 - Substitute one character by another

Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- Edit operations to transform documents
 - Insert a character
 - Delete a character
 - Substitute one character by another
- “The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructuress ~~property~~ can ~~d itbse~~ used in designing algorithms”
- insert, ~~delete~~, substitute

Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- Edit operations to transform documents
 - Insert a character
 - Delete a character
 - Substitute one character by another
- “The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructuress ~~property~~ can ~~itbse~~ used in designing algorithms”
- insert, ~~delete~~, substitute

Edit distance

Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- Edit operations to transform documents
 - Insert a character
 - Delete a character
 - Substitute one character by another
- “The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructuress ~~property~~ can ~~itbse~~ used in designing algorithms”
- insert, ~~delete~~, substitute

Edit distance

- Minimum number of edit operations needed

Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- Edit operations to transform documents
 - Insert a character
 - Delete a character
 - Substitute one character by another
- “The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructuress ~~property~~ can ~~itbse~~ used in designing algorithms”
- insert, ~~delete~~, substitute

Edit distance

- Minimum number of edit operations needed
- In our example, 24 characters inserted, 18 ~~deleted~~, 2 substituted

Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
 - “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
 - Edit operations to transform documents
 - Insert a character
 - Delete a character
 - Substitute one character by another
-
- “The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructuress ~~property~~ can ~~it~~ bse used in designing algorithms”
 - insert, ~~delete~~, substitute

Edit distance

- Minimum number of edit operations needed
- In our example, 24 characters inserted, 18 ~~deleted~~, 2 substituted
- Edit distance is at most 44

Edit distance

- Minimum number of editing operations needed to transform one document to the other
 - Insert a character
 - Delete a character
 - Substitute one character by another

Edit distance

- Minimum number of editing operations needed to transform one document to the other
 - Insert a character
 - Delete a character
 - Substitute one character by another
- Also called Levenshtein distance
 - Vladimir Levenshtein, 1965

Edit distance

- Minimum number of editing operations needed to transform one document to the other
 - Insert a character
 - Delete a character
 - Substitute one character by another
- Also called Levenshtein distance
 - Vladimir Levenshtein, 1965
- Applications
 - Suggestions for spelling correction
 - Genetic similarity of species

Edit distance

- Minimum number of editing operations needed to transform one document to the other
 - Insert a character
 - Delete a character
 - Substitute one character by another
- Also called Levenshtein distance
 - Vladimir Levenshtein, 1965
- Applications
 - Suggestions for spelling correction
 - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u, v

Edit distance

- Minimum number of editing operations needed to transform one document to the other
 - Insert a character
 - Delete a character
 - Substitute one character by another
- Also called Levenshtein distance
 - Vladimir Levenshtein, 1965
- Applications
 - Suggestions for spelling correction
 - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u , v
- Minimum number of deletes needed to make them equal

Edit distance

- Minimum number of editing operations needed to transform one document to the other
 - Insert a character
 - Delete a character
 - Substitute one character by another
- Also called Levenshtein distance
 - Vladimir Levenshtein, 1965
- Applications
 - Suggestions for spelling correction
 - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u , v
 - Minimum number of deletes needed to make them equal
 - Deleting a letter from u is equivalent to inserting it in v

Edit distance

- Minimum number of editing operations needed to transform one document to the other
 - Insert a character
 - Delete a character
 - Substitute one character by another
- Also called Levenshtein distance
 - Vladimir Levenshtein, 1965
- Applications
 - Suggestions for spelling correction
 - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u , v
 - Minimum number of deletes needed to make them equal
- Deleting a letter from u is equivalent to inserting it in v
 - `bisect`, `secret` — LCS is `sect`

Edit distance

- Minimum number of editing operations needed to transform one document to the other
 - Insert a character
 - Delete a character
 - Substitute one character by another
- Also called Levenshtein distance
 - Vladimir Levenshtein, 1965
- Applications
 - Suggestions for spelling correction
 - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u , v
 - Minimum number of deletes needed to make them equal
- Deleting a letter from u is equivalent to inserting it in v
 - `bisect`, `secret` — LCS is `sect`
 - Delete `b`, `i` in `bisect` and `r`, `e` in `secret`

Edit distance

- Minimum number of editing operations needed to transform one document to the other
 - Insert a character
 - Delete a character
 - Substitute one character by another
- Also called Levenshtein distance
 - Vladimir Levenshtein, 1965
- Applications
 - Suggestions for spelling correction
 - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u , v
 - Minimum number of deletes needed to make them equal
- Deleting a letter from u is equivalent to inserting it in v
 - `bisect`, `secret` — LCS is `sect`
 - Delete `b`, `i` in `bisect` and `r`, `e` in `secret`
 - Delete `b`, `i` and then insert `r`, `e` in `bisect`

Edit distance

- Minimum number of editing operations needed to transform one document to the other
 - Insert a character
 - Delete a character
 - Substitute one character by another
- Also called Levenshtein distance
 - Vladimir Levenshtein, 1965
- Applications
 - Suggestions for spelling correction
 - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u , v
 - Minimum number of deletes needed to make them equal
- Deleting a letter from u is equivalent to inserting it in v
 - `bisect`, `secret` — LCS is `sect`
 - Delete `b`, `i` in `bisect` and `r`, `e` in `secret`
 - Delete `b`, `i` and then insert `r`, `e` in `bisect`
- LCS equivalent to edit distance without substitution

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
-
- Recall LCS

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$

- Recall LCS

- If $a_i = b_j$,

$$LCS(i, j) = 1 + LCS(i+1, j+1)$$

- If $a_i \neq b_j$,

$$LCS(i, j) = \max[LCS(i, j+1), LCS(i+1, j)]$$

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
-
- Recall LCS
 - Edit distance — aim is to transform u to v
-
- If $a_i = b_j$,
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
 - If $a_i \neq b_j$,
$$LCS(i, j) = \max[LCS(i, j+1), LCS(i+1, j)]$$

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
-
- Recall LCS
 - Edit distance — aim is to transform u to v
-
- If $a_i = b_j$,
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
 - If $a_i \neq b_j$,
$$LCS(i, j) = \max[LCS(i, j+1), LCS(i+1, j)]$$
 - If $a_i = b_j$, nothing to be done

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
-
- Recall LCS
 - If $a_i = b_j$,
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
 - If $a_i \neq b_j$,
$$LCS(i, j) = \max[LCS(i, j+1), LCS(i+1, j)]$$
-
- Edit distance — aim is to transform u to v
 - If $a_i = b_j$, nothing to be done
 - If $a_i \neq b_j$, best among

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
-
- Recall LCS
 - If $a_i = b_j$,
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
 - If $a_i \neq b_j$,
$$LCS(i, j) = \max[LCS(i, j+1), LCS(i+1, j)]$$
-
- Edit distance — aim is to transform u to v
 - If $a_i = b_j$, nothing to be done
 - If $a_i \neq b_j$, best among
 - Substitute a_i by b_j

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
-
- Recall LCS
 - If $a_i = b_j$,
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
 - If $a_i \neq b_j$,
$$LCS(i, j) = \max[LCS(i, j+1), LCS(i+1, j)]$$
 - Edit distance — aim is to transform u to v
 - If $a_i = b_j$, nothing to be done
 - If $a_i \neq b_j$, best among
 - Substitute a_i by b_j
 - Delete a_i

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
-
- Recall LCS
 - If $a_i = b_j$,
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
 - If $a_i \neq b_j$,
$$LCS(i, j) = \max[LCS(i, j+1), LCS(i+1, j)]$$
-
- Edit distance — aim is to transform u to v
 - If $a_i = b_j$, nothing to be done
 - If $a_i \neq b_j$, best among
 - Substitute a_i by b_j
 - Delete a_i
 - Insert b_j before a_i

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Edit distance — transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
 - Substitute a_i by b_j
 - Delete a_i
 - Insert b_j before a_i

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Edit distance — transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
 - Substitute a_i by b_j
 - Delete a_i
 - Insert b_j before a_i
- $ED(i, j)$ — edit distance for
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Edit distance — transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
 - Substitute a_i by b_j
 - Delete a_i
 - Insert b_j before a_i
- $ED(i, j)$ — edit distance for
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
 - Edit distance — transform u to v
 - If $a_i = b_j$, nothing to be done
 - If $a_i \neq b_j$, best among
 - Substitute a_i by b_j
 - Delete a_i
 - Insert b_j before a_i
 - $ED(i, j)$ — edit distance for
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$,
 $ED(i, j) = ED(i+1, j+1)$
 - If $a_i \neq b_j$,
 $ED(i, j) = 1 + \min[ED(i+1, j+1),$
 $ED(i+1, j),$
 $ED(i, j+1)]$

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
 - Edit distance — transform u to v
 - If $a_i = b_j$, nothing to be done
 - If $a_i \neq b_j$, best among
 - Substitute a_i by b_j
 - Delete a_i
 - Insert b_j before a_i
 - $ED(i, j)$ — edit distance for
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$,
 $ED(i, j) = ED(i+1, j+1)$
 - If $a_i \neq b_j$,
 $ED(i, j) = 1 + \min[ED(i+1, j+1),$
 $ED(i+1, j),$
 $ED(i, j+1)]$
 - Base cases
 - $ED(m, n) = 0$

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
 - Edit distance — transform u to v
 - If $a_i = b_j$, nothing to be done
 - If $a_i \neq b_j$, best among
 - Substitute a_i by b_j
 - Delete a_i
 - Insert b_j before a_i
 - $ED(i, j)$ — edit distance for $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$,
 $ED(i, j) = ED(i+1, j+1)$
 - If $a_i \neq b_j$,
 $ED(i, j) = 1 + \min[ED(i+1, j+1), ED(i+1, j), ED(i, j+1)]$
 - Base cases
 - $ED(m, n) = 0$
 - $ED(i, n) = m - i$ for all $0 \leq i \leq m$
Delete $a_i a_{i+1} \dots a_{m-1}$ from u

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
 - Edit distance — transform u to v
 - If $a_i = b_j$, nothing to be done
 - If $a_i \neq b_j$, best among
 - Substitute a_i by b_j
 - Delete a_i
 - Insert b_j before a_i
 - $ED(i, j)$ — edit distance for $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$,
 $ED(i, j) = ED(i+1, j+1)$
 - If $a_i \neq b_j$,
 $ED(i, j) = 1 + \min[ED(i+1, j+1), ED(i+1, j), ED(i, j+1)]$
 - Base cases
 - $ED(m, n) = 0$
 - $ED(i, n) = m - i$ for all $0 \leq i \leq m$
Delete $a_i a_{i+1} \dots a_{m-1}$ from u
 - $ED(m, j) = n - j$ for all $0 \leq j \leq n$
Insert $b_j b_{j+1} \dots b_{n-1}$ into u

Subproblem dependency

- Subproblems are $ED(i, j)$, for
 $0 \leq i \leq m, 0 \leq j \leq n$

Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

The diagram shows a 7x9 grid representing a dynamic programming table for Edit Distance. The columns are labeled 0 through 6 at the top, and the rows are labeled 0 through 6 on the left. The first six columns contain letters: 'b' at (0,0), 'i' at (1,0), 's' at (2,0), 'e' at (3,0), 'c' at (4,0), and 't' at (5,0). The last column contains a black dot at (6,0). Orange arrows indicate dependencies from row 1 to 2, from row 3 to 4, and from row 5 to 6. Specifically, there are two arrows from 'i' to 's', one arrow from 'e' to 'c', and one arrow from 't' to the dot.

Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							6
1	i							5
2	s							4
3	e							3
4	c							2
5	t							1
6	•							0

Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b						5	6
1	i						4	5
2	s						3	4
3	e						2	3
4	c						1	2
5	t						0	1
6	•						1	0

Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b					4	5	6
1	i					3	4	5
2	s					2	3	4
3	e					1	2	3
4	c					1	1	2
5	t					1	0	1
6	•					2	1	0

Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b					4	4	5	6
1	i					3	3	4	5
2	s					2	2	3	4
3	e					2	1	2	3
4	c					2	1	1	2
5	t					2	1	0	1
6	•					3	2	1	0

Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b				4	4	4	5	6
1	i				3	3	3	4	5
2	s				3	2	2	3	4
3	e				3	2	1	2	3
4	c				2	2	1	1	2
5	t				3	2	1	0	1
6	•				4	3	2	1	0

Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	4	4	4	5	6	
1	i	4	3	3	3	4	5	
2	s	3	3	2	2	3	4	
3	e	2	3	2	1	2	3	
4	c	3	2	2	1	1	2	
5	t	4	3	2	1	0	1	
6	•	5	4	3	2	1	0	

Subproblem dependency

- Subproblems are $ED(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i,j)$ depends on $ED(i+1,j+1), ED(i,j+1), ED(i+1,j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	4	4	4	4	5	6
1	i	3	4	3	3	3	4	5
2	s	2	3	3	2	2	3	4
3	e	3	2	3	2	1	2	3
4	c	4	3	2	2	1	1	2
5	t	5	4	3	2	1	0	1
6	•	6	5	4	3	2	1	0

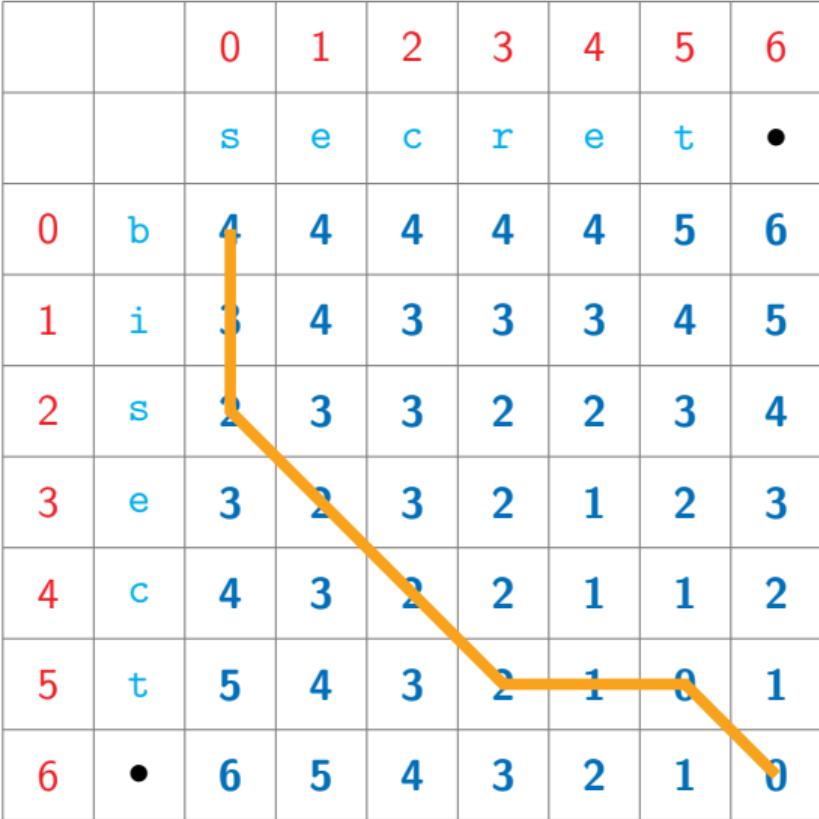
Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Transform `bisect` to `secret`

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	4	4	4	4	5	6	
1	i	3	4	3	3	3	4	5	
2	s	2	3	3	2	2	3	4	
3	e	3	2	3	2	1	2	3	
4	c	4	3	2	2	1	1	2	
5	t	5	4	3	2	1	0	1	
6	•	6	5	4	3	2	1	0	



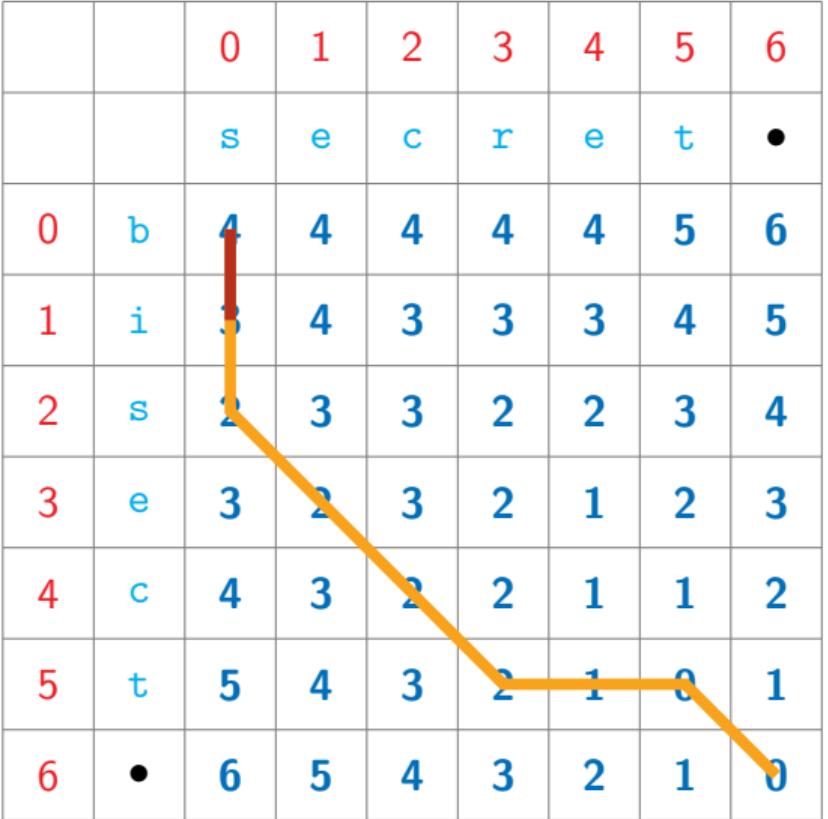
Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b`

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	4	4	4	4	5	6	
1	i	3	4	3	3	3	4	5	
2	s	2	3	3	2	2	3	4	
3	e	3	2	3	2	1	2	3	
4	c	4	3	2	2	1	1	2	
5	t	5	4	3	2	1	0	1	
6	•	6	5	4	3	2	1	0	



Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b`, Delete `i`

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	4	4	4	4	5	6	
1	i	3	4	3	3	3	4	5	
2	s	2	3	3	2	2	3	4	
3	e	3	2	3	2	1	2	3	
4	c	4	3	2	2	1	1	2	
5	t	5	4	3	2	1	0	1	
6	•	6	5	4	3	2	1	0	

Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b`, Delete `i`, Insert `r`

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	4	4	4	4	5	6
1	i	3	4	3	3	3	4	5
2	s	2	3	3	2	2	3	4
3	e	3	2	3	2	1	2	3
4	c	4	3	2	2	1	1	2
5	t	5	4	3	2	1	0	1
6	•	6	5	4	3	2	1	0

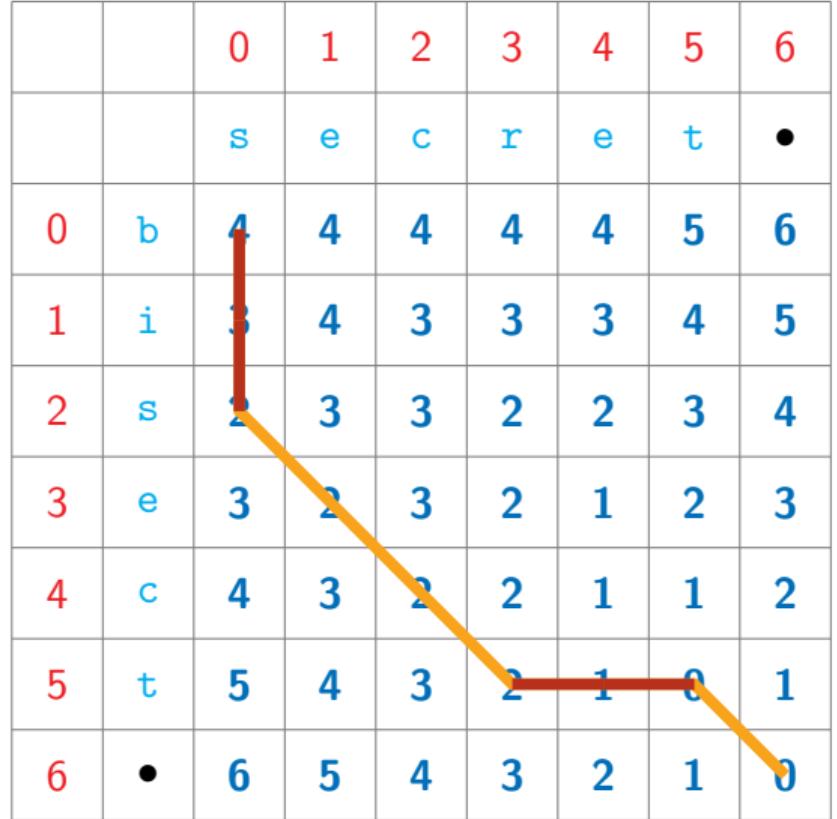
Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- Like LCS, $ED(i, j)$ depends on $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for $ED(m, n)$ — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b`, Delete `i`, Insert `r`, Insert `e`

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	4	4	4	4	5	6
1	i	3	4	3	3	3	4	5
2	s	2	3	3	2	2	3	4
3	e	3	2	3	2	1	2	3
4	c	4	3	2	2	1	1	2
5	t	5	4	3	2	1	0	1
6	•	6	5	4	3	2	1	0



Implementation

```
def ED(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    ed = np.zeros((m+1,n+1))  
  
    for i in range(m-1,-1,-1):  
        ed[i,n] = m-i  
    for j in range(n-1,-1,-1):  
        ed[m,j] = n-j  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                ed[i,j] = ed[i+1,j+1]  
            else:  
                ed[i,j] = 1 + min(ed[i+1,j+1],  
                                  ed[i,j+1],  
                                  ed[i+1,j])  
  
    return(ed[0,0])
```

Implementation

```
def ED(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    ed = np.zeros((m+1,n+1))  
  
    for i in range(m-1,-1,-1):  
        ed[i,n] = m-i  
    for j in range(n-1,-1,-1):  
        ed[m,j] = n-j  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                ed[i,j] = ed[i+1,j+1]  
            else:  
                ed[i,j] = 1 + min(ed[i+1,j+1],  
                                  ed[i,j+1],  
                                  ed[i+1,j])  
  
    return(ed[0,0])
```

Complexity

Implementation

```
def ED(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    ed = np.zeros((m+1,n+1))

    for i in range(m-1,-1,-1):
        ed[i,n] = m-i
    for j in range(n-1,-1,-1):
        ed[m,j] = n-j

    for j in range(n-1,-1,-1):
        for i in range(m-1,-1,-1):
            if u[i] == v[j]:
                ed[i,j] = ed[i+1,j+1]
            else:
                ed[i,j] = 1 + min(ed[i+1,j+1],
                                   ed[i,j+1],
                                   ed[i+1,j])
    return(ed[0,0])
```

Complexity

- Again $O(mn)$, using dynamic programming or memoization

Implementation

```
def ED(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    ed = np.zeros((m+1,n+1))

    for i in range(m-1,-1,-1):
        ed[i,n] = m-i
    for j in range(n-1,-1,-1):
        ed[m,j] = n-j

    for j in range(n-1,-1,-1):
        for i in range(m-1,-1,-1):
            if u[i] == v[j]:
                ed[i,j] = ed[i+1,j+1]
            else:
                ed[i,j] = 1 + min(ed[i+1,j+1],
                                  ed[i,j+1],
                                  ed[i+1,j])
    return(ed[0,0])
```

Complexity

- Again $O(mn)$, using dynamic programming or memoization
 - Fill a table of size $O(mn)$
 - Each table entry takes constant time to compute

Matrix Multiplication

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 9

Multiplying matrices

- Multiply matrices A , B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

Multiplying matrices

- Multiply matrices A , B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n$, $B : n \times p$
- $AB : m \times p$

Multiplying matrices

- Multiply matrices A , B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
 - $A : m \times n$, $B : n \times p$
 - $AB : m \times p$
- Computing each entry in AB is $O(n)$

Multiplying matrices

- Multiply matrices A , B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
 - $A : m \times n$, $B : n \times p$
 - $AB : m \times p$
- Computing each entry in AB is $O(n)$
- Overall, computing AB is $O(mnp)$

Multiplying matrices

- Multiply matrices A , B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
 - $A : m \times n$, $B : n \times p$
 - $AB : m \times p$
- Computing each entry in AB is $O(n)$
- Overall, computing AB is $O(mnp)$
- Matrix multiplication is associative
 - $ABC = (AB)C = A(BC)$

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $$A : m \times n, B : n \times p$$
 - $$AB : m \times p$$

- Computing each entry in AB is $O(n)$

- Overall, computing AB is $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
 - Bracketing does not change answer

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
 - $A : m \times n, B : n \times p$
 - $AB : m \times p$
- Computing each entry in AB is $O(n)$
- Overall, computing AB is $O(mnp)$
- Matrix multiplication is associative
 - $ABC = (AB)C = A(BC)$
 - Bracketing does not change answer
 - ... but can affect the complexity!

Multiplying matrices

- Multiply matrices A, B
 - $AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$
- Dimensions must be compatible
 - $A : m \times n, B : n \times p$
 - $AB : m \times p$
- Computing each entry in AB is $O(n)$
- Overall, computing AB is $O(mnp)$
- Matrix multiplication is associative
 - $ABC = (AB)C = A(BC)$
 - Bracketing does not change answer
 - ... but can affect the complexity!

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
 - $A : m \times n, B : n \times p$
 - $AB : m \times p$
- Computing each entry in AB is $O(n)$
- Overall, computing AB is $O(mnp)$
- Matrix multiplication is associative
 - $ABC = (AB)C = A(BC)$
 - Bracketing does not change answer
 - ... but can affect the complexity!

- Let $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$
- Computing $A(BC)$

Multiplying matrices

- Multiply matrices A, B

- $AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in AB is $O(n)$

- Overall, computing AB is $O(mnp)$

- Matrix multiplication is associative

- $ABC = (AB)C = A(BC)$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing $A(BC)$

- $BC : 100 \times 100$, takes
 $100 \cdot 1 \cdot 100 = 10000$ steps to compute

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in AB is $O(n)$

- Overall, computing AB is $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing $A(BC)$

- $BC : 100 \times 100$, takes
 $100 \cdot 1 \cdot 100 = 10000$ steps to compute
- $A(BC) : 1 \times 100$, takes
 $1 \cdot 100 \cdot 100 = 10000$ steps to compute

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in AB is $O(n)$

- Overall, computing AB is $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing $A(BC)$

- $BC : 100 \times 100$, takes
 $100 \cdot 1 \cdot 100 = 10000$ steps to compute

- $A(BC) : 1 \times 100$, takes
 $1 \cdot 100 \cdot 100 = 10000$ steps to compute

- Computing $(AB)C$

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in AB is $O(n)$

- Overall, computing AB is $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing $A(BC)$

- $BC : 100 \times 100$, takes
 $100 \cdot 1 \cdot 100 = 10000$ steps to compute
- $A(BC) : 1 \times 100$, takes
 $1 \cdot 100 \cdot 100 = 10000$ steps to compute

- Computing $(AB)C$

- $AB : 1 \times 1$, takes
 $1 \cdot 100 \cdot 1 = 100$ steps to compute

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in AB is $O(n)$

- Overall, computing AB is $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing $A(BC)$

- $BC : 100 \times 100$, takes
 $100 \cdot 1 \cdot 100 = 10000$ steps to compute
- $A(BC) : 1 \times 100$, takes
 $1 \cdot 100 \cdot 100 = 10000$ steps to compute

- Computing $(AB)C$

- $AB : 1 \times 1$, takes
 $1 \cdot 100 \cdot 1 = 100$ steps to compute
- $(AB)C : 1 \times 100$, takes
 $1 \cdot 1 \cdot 100 = 100$ steps to compute

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in AB is $O(n)$

- Overall, computing AB is $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing $A(BC)$

- $BC : 100 \times 100$, takes
 $100 \cdot 1 \cdot 100 = 10000$ steps to compute
- $A(BC) : 1 \times 100$, takes
 $1 \cdot 100 \cdot 100 = 10000$ steps to compute

- Computing $(AB)C$

- $AB : 1 \times 1$, takes
 $1 \cdot 100 \cdot 1 = 100$ steps to compute
- $(AB)C : 1 \times 100$, takes
 $1 \cdot 1 \cdot 100 = 100$ steps to compute

- 20000 steps vs 200 steps!

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
 - $A : m \times n, B : n \times p$
 - $AB : m \times p$
- Computing each entry in AB is $O(n)$
- Overall, computing AB is $O(mnp)$
- Matrix multiplication is associative
 - $ABC = (AB)C = A(BC)$
 - Bracketing does not change answer
 - ... but can affect the complexity!

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
 - $A : m \times n, B : n \times p$
 - $AB : m \times p$
- Computing each entry in AB is $O(n)$
- Overall, computing AB is $O(mnp)$
- Matrix multiplication is associative
 - $ABC = (AB)C = A(BC)$
 - Bracketing does not change answer
 - ... but can affect the complexity!

- Given n matrices $M_0 : r_0 \times c_0, M_1 : r_1 \times c_1, \dots, M_{n-1} : r_{n-1} \times c_{n-1}$

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
 - $A : m \times n, B : n \times p$
 - $AB : m \times p$
- Computing each entry in AB is $O(n)$
- Overall, computing AB is $O(mnp)$
- Matrix multiplication is associative
 - $ABC = (AB)C = A(BC)$
 - Bracketing does not change answer
 - ... but can affect the complexity!

- Given n matrices $M_0 : r_0 \times c_0, M_1 : r_1 \times c_1, \dots, M_{n-1} : r_{n-1} \times c_{n-1}$
 - Dimensions match: $r_j = c_{j-1}, 0 < j < n$

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in AB is $O(n)$

- Overall, computing AB is $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Given n matrices $M_0 : r_0 \times c_0,$

- $M_1 : r_1 \times c_1, \dots, M_{n-1} : r_{n-1} \times c_{n-1}$

- Dimensions match: $r_j = c_{j-1}, 0 < j < n$
- Product $M_0 \cdot M_1 \cdots M_{n-1}$ can be computed

Multiplying matrices

- Multiply matrices A, B

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
 - $A : m \times n, B : n \times p$
 - $AB : m \times p$
- Computing each entry in AB is $O(n)$
- Overall, computing AB is $O(mnp)$
- Matrix multiplication is associative
 - $ABC = (AB)C = A(BC)$
 - Bracketing does not change answer
 - ... but can affect the complexity!

- Given n matrices $M_0 : r_0 \times c_0, M_1 : r_1 \times c_1, \dots, M_{n-1} : r_{n-1} \times c_{n-1}$
 - Dimensions match: $r_j = c_{j-1}, 0 < j < n$
 - Product $M_0 \cdot M_1 \cdots M_{n-1}$ can be computed
- Find an optimal order to compute the product
 - Multiply two matrices at a time
 - Bracket the expression optimally

Inductive structure

- Final step combines two subproducts

$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$

for some $0 < k < n$

Inductive structure

- Final step combines two subproducts

$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$

for some $0 < k < n$

- First factor is $r_0 \times c_{k-1}$, second is
 $r_k \times c_{n-1}$, where $r_k = c_{k-1}$

Inductive structure

- Final step combines two subproducts

$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$

for some $0 < k < n$

- First factor is $r_0 \times c_{k-1}$, second is

$r_k \times c_{n-1}$, where $r_k = c_{k-1}$

- Let $C(0, n-1)$ denote the overall cost

Inductive structure

- Final step combines two subproducts

$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$

for some $0 < k < n$

- First factor is $r_0 \times c_{k-1}$, second is
 $r_k \times c_{n-1}$, where $r_k = c_{k-1}$
- Let $C(0, n-1)$ denote the overall cost
- Final multiplication is $O(r_0 r_k c_{n-1})$

Inductive structure

- Final step combines two subproducts

$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$

for some $0 < k < n$

- First factor is $r_0 \times c_{k-1}$, second is

$r_k \times c_{n-1}$, where $r_k = c_{k-1}$

- Let $C(0, n-1)$ denote the overall cost

- Final multiplication is $O(r_0 r_k c_{n-1})$

- Inductively, costs of factors are $C(0, k-1)$ and $C(k, n-1)$

Inductive structure

- Final step combines two subproducts

$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$

for some $0 < k < n$

- First factor is $r_0 \times c_{k-1}$, second is $r_k \times c_{n-1}$, where $r_k = c_{k-1}$
- Let $C(0, n-1)$ denote the overall cost
- Final multiplication is $O(r_0 r_k c_{n-1})$
- Inductively, costs of factors are $C(0, k-1)$ and $C(k, n-1)$
- $C(0, n-1) = C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$

Inductive structure

- Final step combines two subproducts
 $(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$
for some $0 < k < n$
 - First factor is $r_0 \times c_{k-1}$, second is
 $r_k \times c_{n-1}$, where $r_k = c_{k-1}$
 - Let $C(0, n-1)$ denote the overall cost
 - Final multiplication is $O(r_0 r_k c_{n-1})$
 - Inductively, costs of factors are $C(0, k-1)$
and $C(k, n-1)$
 - $C(0, n-1) =$
 $C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which k should we choose?

Inductive structure

- Final step combines two subproducts
$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$
for some $0 < k < n$
 - First factor is $r_0 \times c_{k-1}$, second is $r_k \times c_{n-1}$, where $r_k = c_{k-1}$
 - Let $C(0, n-1)$ denote the overall cost
 - Final multiplication is $O(r_0 r_k c_{n-1})$
 - Inductively, costs of factors are $C(0, k-1)$ and $C(k, n-1)$
 - $C(0, n-1) = C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which k should we choose?
 - Try all and choose the minimum!

Inductive structure

- Final step combines two subproducts
$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$
for some $0 < k < n$
 - First factor is $r_0 \times c_{k-1}$, second is $r_k \times c_{n-1}$, where $r_k = c_{k-1}$
 - Let $C(0, n-1)$ denote the overall cost
 - Final multiplication is $O(r_0 r_k c_{n-1})$
 - Inductively, costs of factors are $C(0, k-1)$ and $C(k, n-1)$
 - $C(0, n-1) = C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which k should we choose?
 - Try all and choose the minimum!
 - Subproblems?

Inductive structure

- Final step combines two subproducts
$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$
for some $0 < k < n$
- First factor is $r_0 \times c_{k-1}$, second is $r_k \times c_{n-1}$, where $r_k = c_{k-1}$
- Let $C(0, n-1)$ denote the overall cost
- Final multiplication is $O(r_0 r_k c_{n-1})$
- Inductively, costs of factors are $C(0, k-1)$ and $C(k, n-1)$
- $C(0, n-1) = C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which k should we choose?
 - Try all and choose the minimum!
- Subproblems?
 - $M_0 \cdot M_1 \cdots M_{k-1}$ would decompose as $(M_0 \cdots M_{j-1}) \cdot (M_j \cdots M_{k-1})$
 - Generic subproblem is $M_j \cdot M_{j+1} \cdots M_k$

Inductive structure

- Final step combines two subproducts
$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$
for some $0 < k < n$
- First factor is $r_0 \times c_{k-1}$, second is $r_k \times c_{n-1}$, where $r_k = c_{k-1}$
- Let $C(0, n-1)$ denote the overall cost
- Final multiplication is $O(r_0 r_k c_{n-1})$
- Inductively, costs of factors are $C(0, k-1)$ and $C(k, n-1)$
- $C(0, n-1) = C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which k should we choose?
 - Try all and choose the minimum!
- Subproblems?
 - $M_0 \cdot M_1 \cdots M_{k-1}$ would decompose as $(M_0 \cdots M_{j-1}) \cdot (M_j \cdots M_{k-1})$
 - Generic subproblem is $M_j \cdot M_{j+1} \cdots M_k$
- $C(j, k) = \min_{j < \ell \leq k} [C(j, \ell-1) + C(\ell, k) + r_j r_\ell c_k]$

Inductive structure

- Final step combines two subproducts
$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$
for some $0 < k < n$
- First factor is $r_0 \times c_{k-1}$, second is $r_k \times c_{n-1}$, where $r_k = c_{k-1}$
- Let $C(0, n-1)$ denote the overall cost
- Final multiplication is $O(r_0 r_k c_{n-1})$
- Inductively, costs of factors are $C(0, k-1)$ and $C(k, n-1)$
- $C(0, n-1) = C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which k should we choose?
 - Try all and choose the minimum!
- Subproblems?
 - $M_0 \cdot M_1 \cdots M_{k-1}$ would decompose as $(M_0 \cdots M_{j-1}) \cdot (M_j \cdots M_{k-1})$
 - Generic subproblem is $M_j \cdot M_{j+1} \cdots M_k$
- $C(j, k) = \min_{j < \ell \leq k} [C(j, \ell-1) + C(\ell, k) + r_j r_\ell c_k]$
- Base case: $C(j, j) = 0$ for $0 \leq j < n$

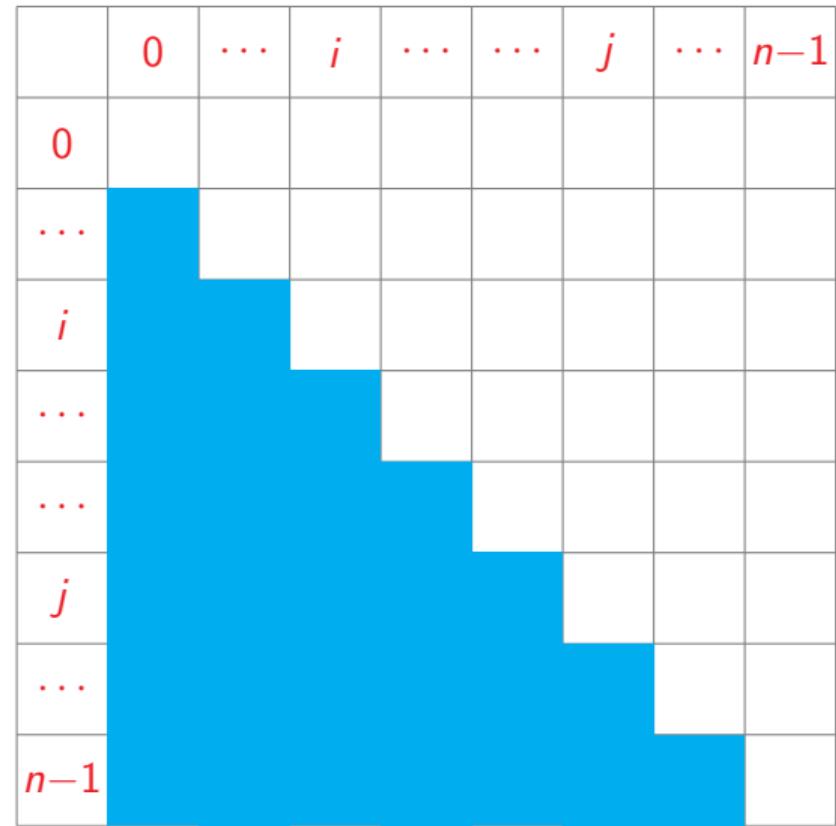
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$

	0	...	i	j	...	$n-1$
0								
...								
i								
...								
j								
...								
$n-1$								

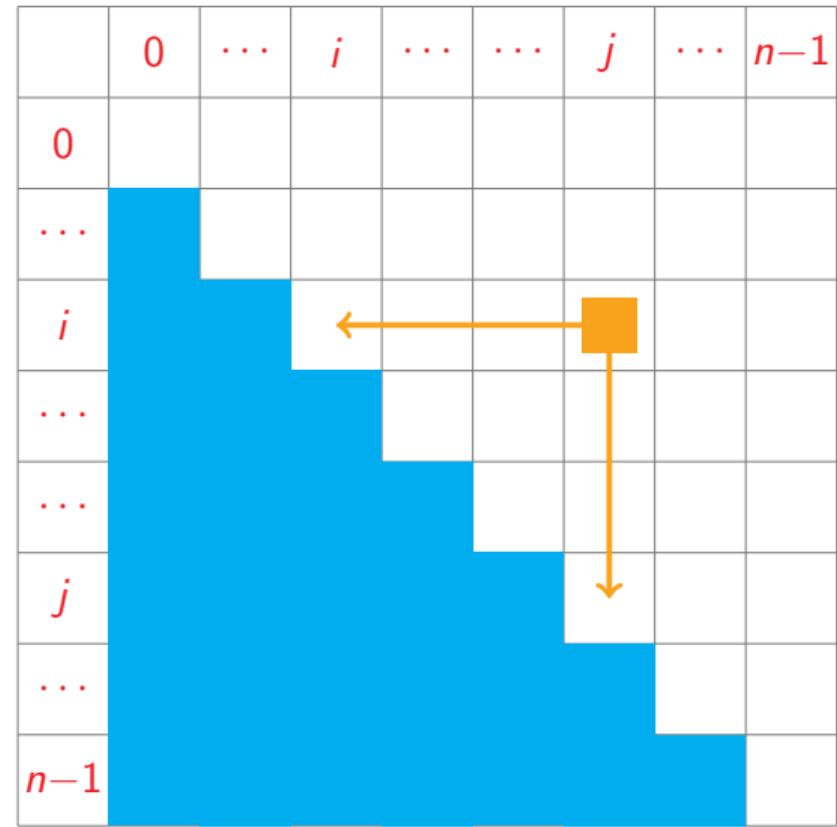
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal



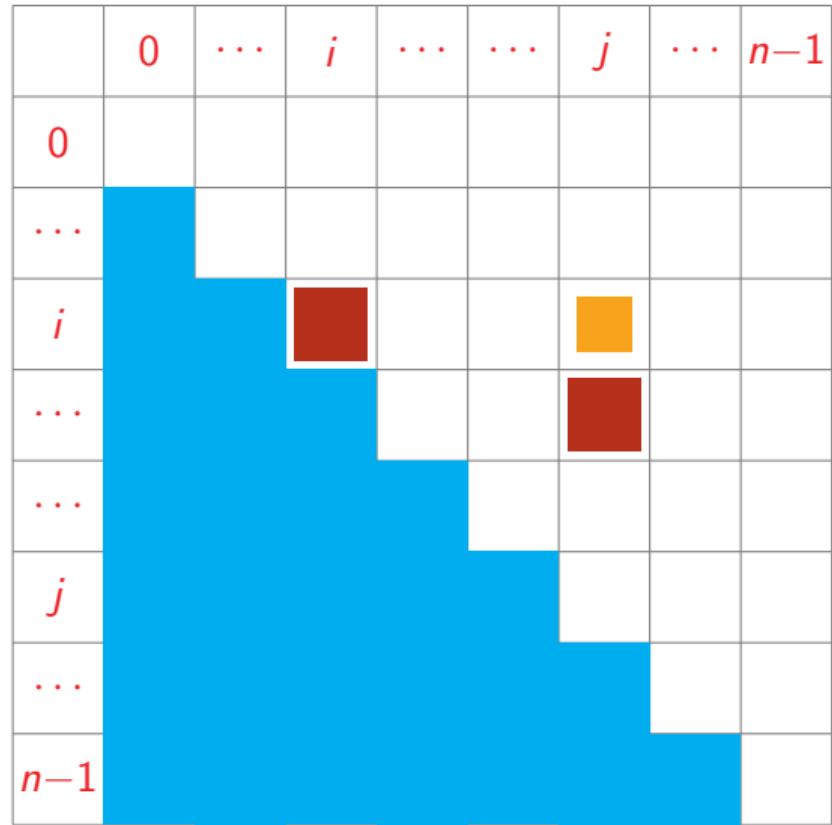
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$



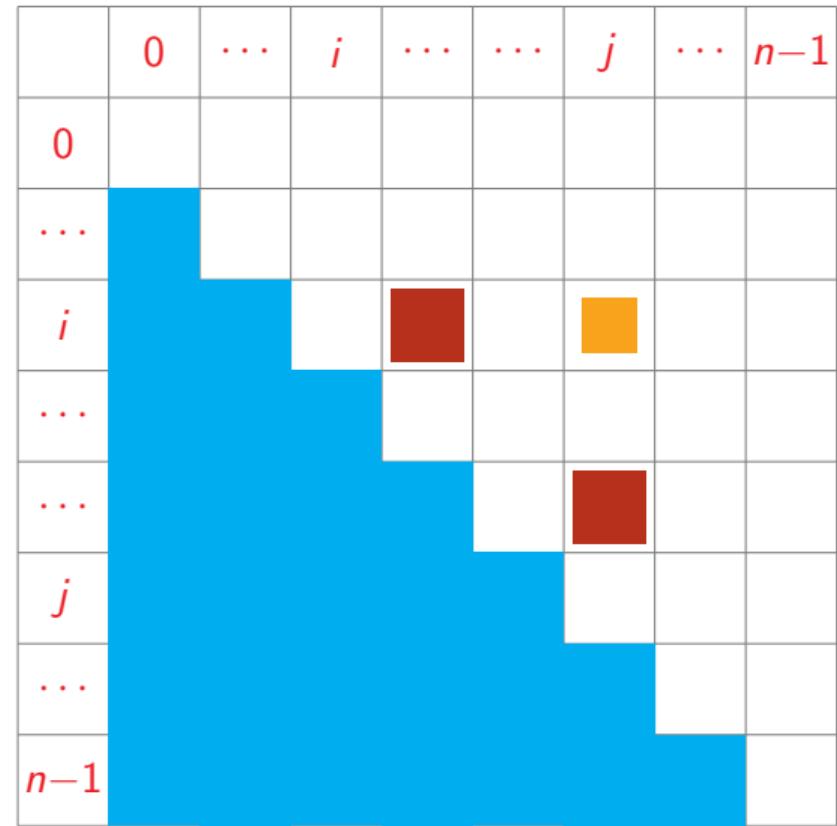
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$



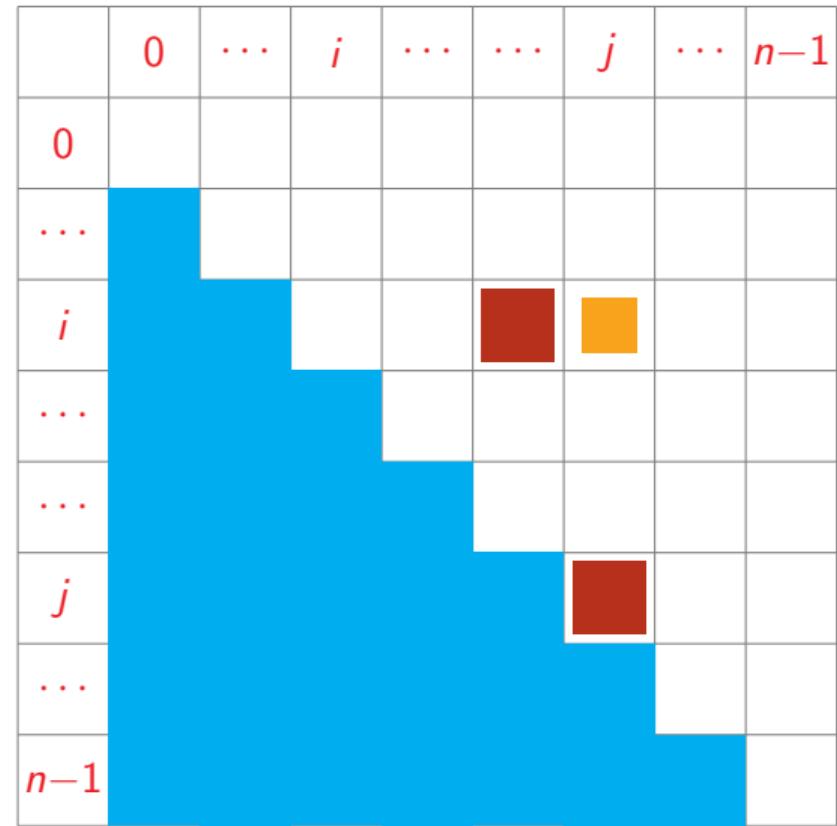
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$



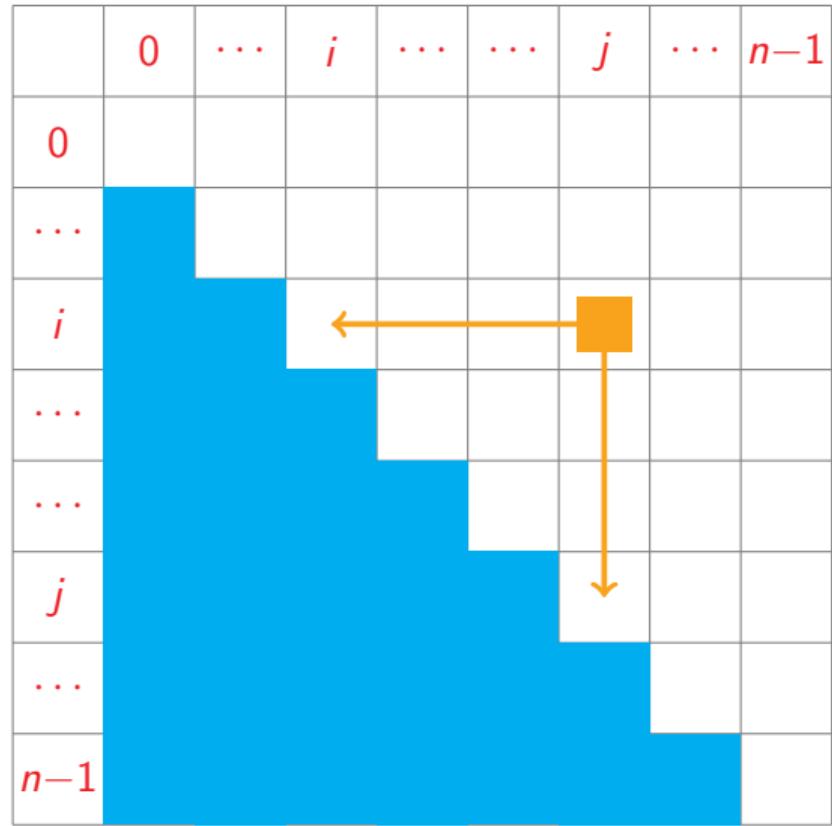
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$



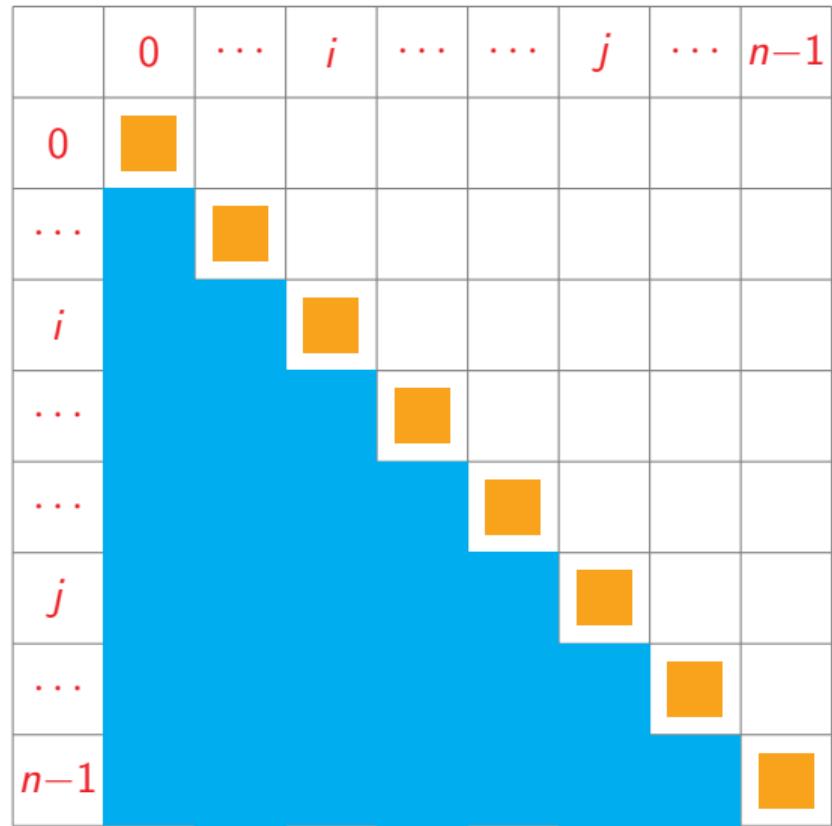
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$
 - $O(n)$ dependencies per entry, unlike LCW, LCS and ED



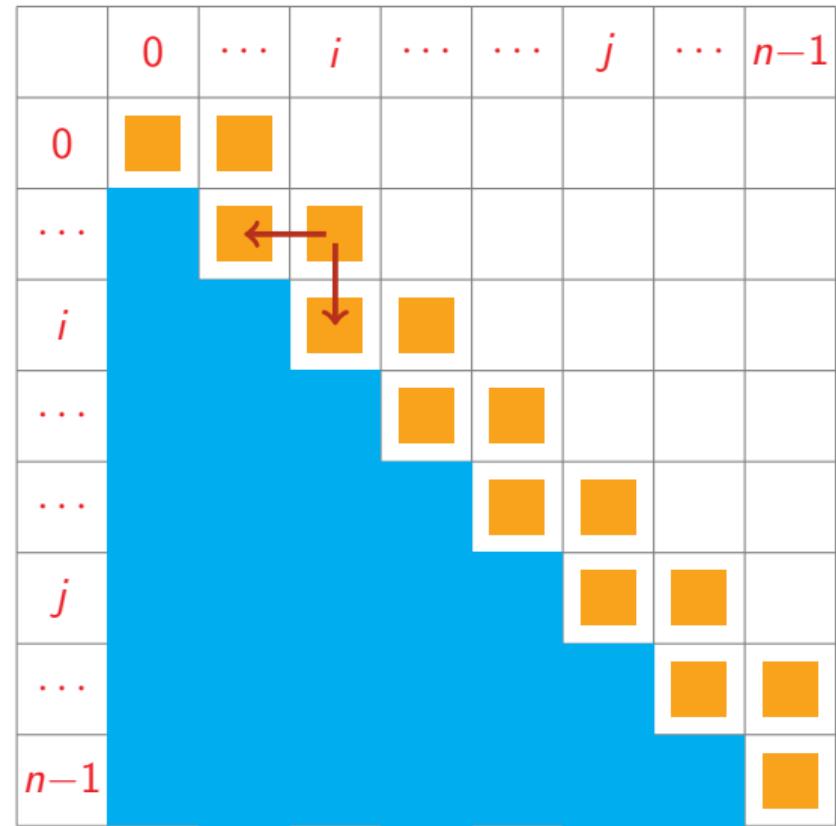
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$
 - $O(n)$ dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case



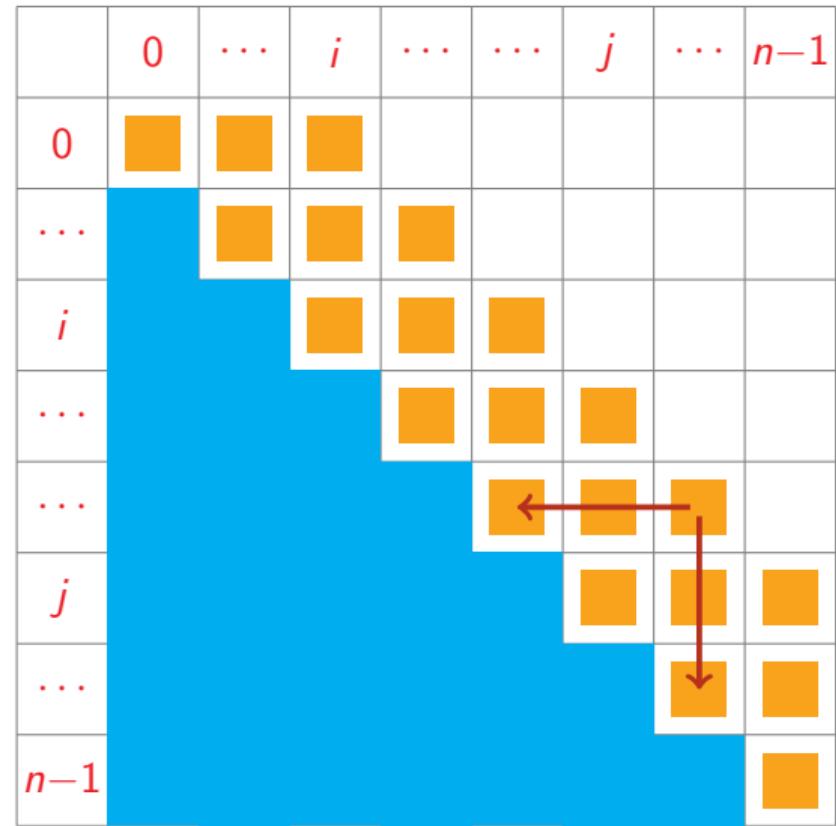
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$
 - $O(n)$ dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



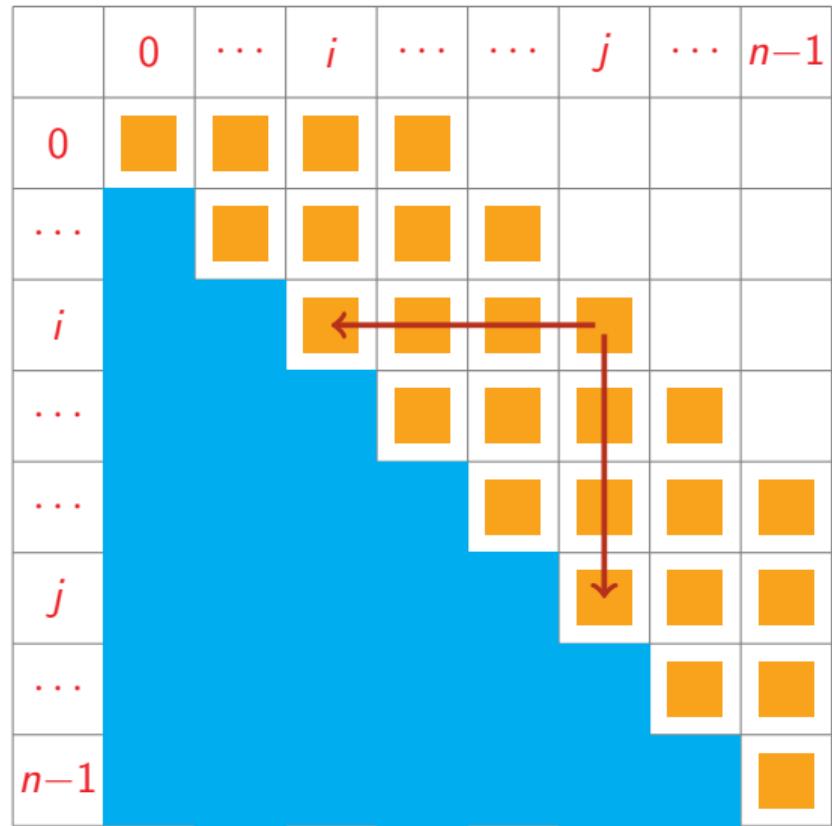
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$
 - $O(n)$ dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



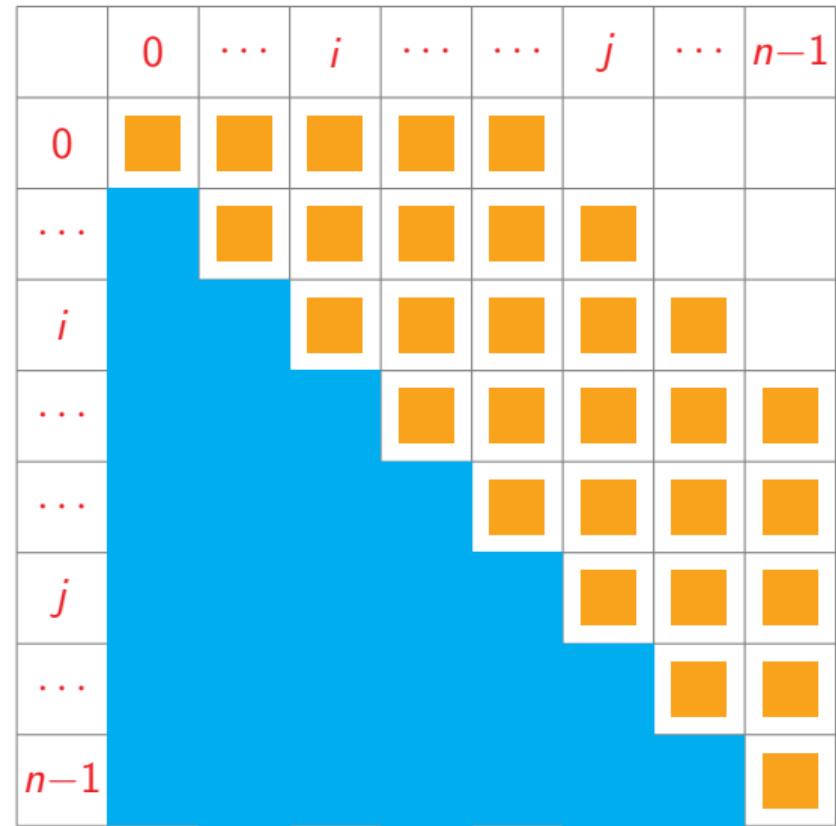
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$
 - $O(n)$ dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



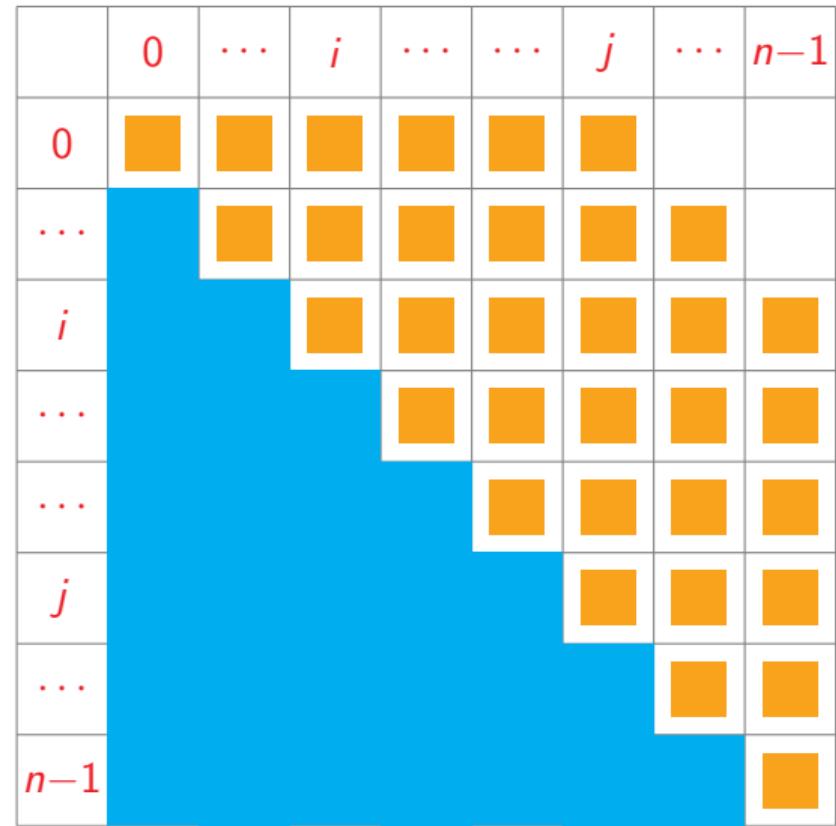
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$
 - $O(n)$ dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



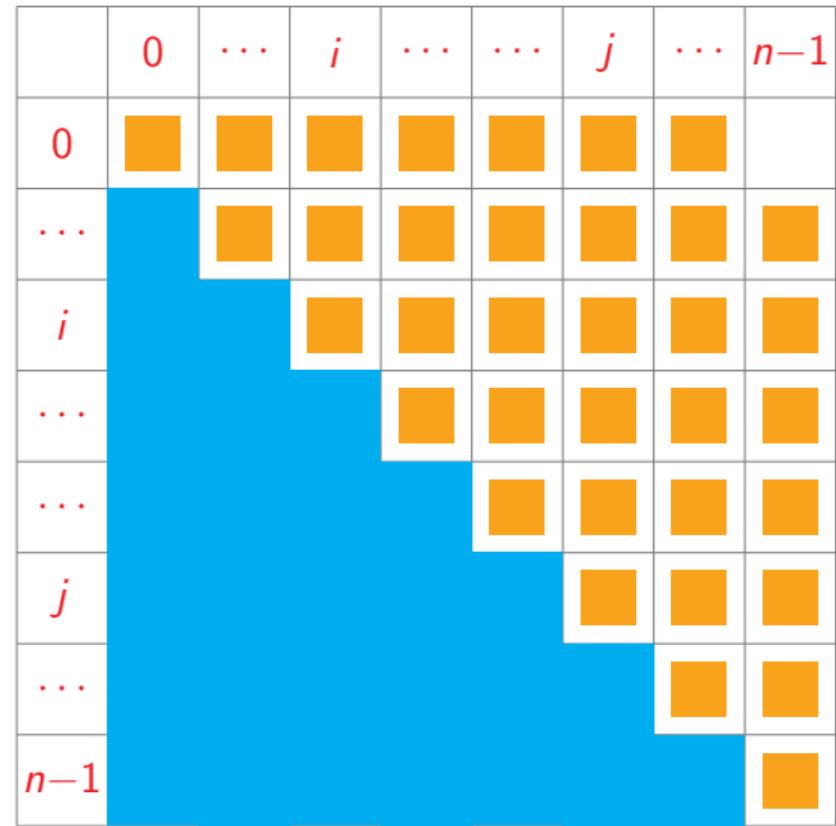
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$
 - $O(n)$ dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



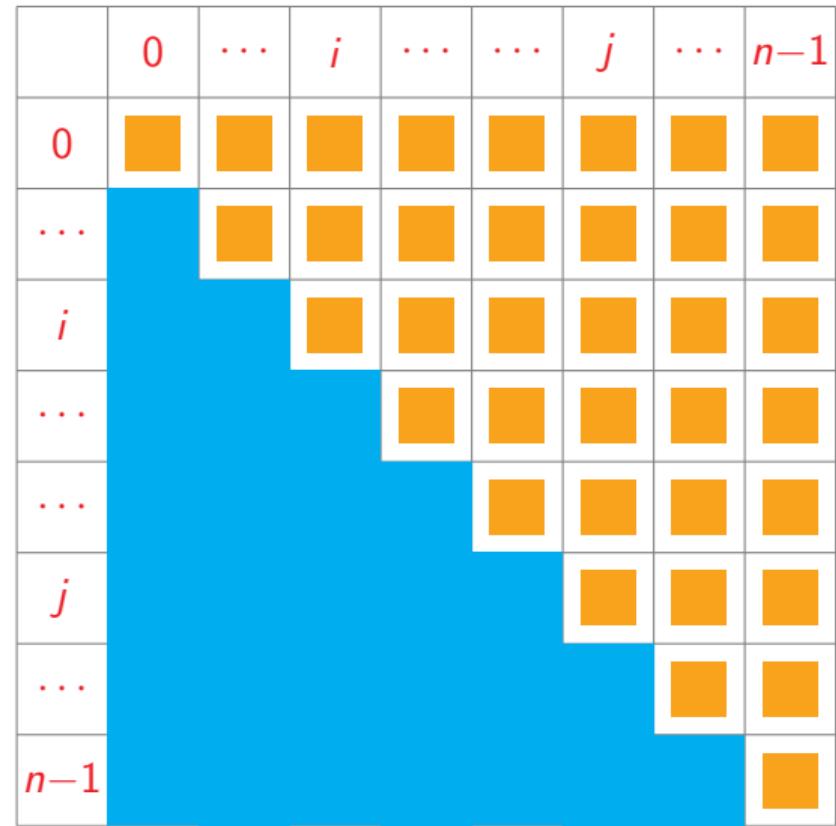
Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$
 - $O(n)$ dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



Subproblem dependency

- Compute $C(i,j)$, $0 \leq i,j < n$
 - Only for $i \leq j$
 - Entries above main diagonal
- $C(i,j)$ depends on $C(i, k-1)$, $C(k,j)$ for every $i < k \leq j$
 - $O(n)$ dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



Implementation

```
def C(dim):
    # dim: dimension matrix,
    #       entries are pairs (r_i,c_i)
    import numpy as np
    n = dim.shape[0]
    C = np.zeros((n,n))
    for i in range(n):
        C[i,i] = 0
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            C[i,j] = C[i,i] +
                      C[i+1,j] +
                      dim[i][0]*dim[i+1][0]*dim[j][1]
            for k in range(i+1,j+1):
                C[i,j] = min(C[i,j],
                              C[i,k-1] + C[k,j] +
                              dim[i][0]*dim[k][0]*dim[j][1])
    return(C[0,n-1])
```

Implementation

```
def C(dim):
    # dim: dimension matrix,
    #       entries are pairs (r_i,c_i)
    import numpy as np
    n = dim.shape[0]
    C = np.zeros((n,n))
    for i in range(n):
        C[i,i] = 0
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            C[i,j] = C[i,i] +
                      C[i+1,j] +
                      dim[i][0]*dim[i+1][0]*dim[j][1]
            for k in range(i+1,j+1):
                C[i,j] = min(C[i,j],
                              C[i,k-1] + C[k,j] +
                              dim[i][0]*dim[k][0]*dim[j][1])
    return(C[0,n-1])
```

Complexity

Implementation

```
def C(dim):
    # dim: dimension matrix,
    #       entries are pairs (r_i,c_i)
    import numpy as np
    n = dim.shape[0]
    C = np.zeros((n,n))
    for i in range(n):
        C[i,i] = 0
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            C[i,j] = C[i,i] +
                      C[i+1,j] +
                      dim[i][0]*dim[i+1][0]*dim[j][1]
            for k in range(i+1,j+1):
                C[i,j] = min(C[i,j],
                              C[i,k-1] + C[k,j] +
                              dim[i][0]*dim[k][0]*dim[j][1])
    return(C[0,n-1])
```

Complexity

- We have to fill a table of size $O(n^2)$

Implementation

```
def C(dim):
    # dim: dimension matrix,
    #       entries are pairs (r_i,c_i)
    import numpy as np
    n = dim.shape[0]
    C = np.zeros((n,n))
    for i in range(n):
        C[i,i] = 0
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            C[i,j] = C[i,i] +
                C[i+1,j] +
                dim[i][0]*dim[i+1][0]*dim[j][1]
            for k in range(i+1,j+1):
                C[i,j] = min(C[i,j],
                    C[i,k-1] + C[k,j] +
                    dim[i][0]*dim[k][0]*dim[j][1])
    return(C[0,n-1])
```

Complexity

- We have to fill a table of size $O(n^2)$
- Filling each entry takes $O(n)$

Implementation

```
def C(dim):
    # dim: dimension matrix,
    #       entries are pairs (r_i,c_i)
    import numpy as np
    n = dim.shape[0]
    C = np.zeros((n,n))
    for i in range(n):
        C[i,i] = 0
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            C[i,j] = C[i,i] +
                C[i+1,j] +
                dim[i][0]*dim[i+1][0]*dim[j][1]
            for k in range(i+1,j+1):
                C[i,j] = min(C[i,j],
                              C[i,k-1] + C[k,j] +
                              dim[i][0]*dim[k][0]*dim[j][1])
    return(C[0,n-1])
```

Complexity

- We have to fill a table of size $O(n^2)$
- Filling each entry takes $O(n)$
- Overall, $O(n^3)$

String Matching

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 10

String matching

- Searching for a pattern is a fundamental problem when dealing with text
 - Editing a document
 - Answering an internet search query
 - Looking for a match in a gene sequence

String matching

- Searching for a pattern is a fundamental problem when dealing with text
 - Editing a document
 - Answering an internet search query
 - Looking for a match in a gene sequence
- Example
 - **an** occurs in **banana** at two positions

String matching

- Searching for a pattern is a fundamental problem when dealing with text
 - Editing a document
 - Answering an internet search query
 - Looking for a match in a gene sequence
- Example
 - `an` occurs in `banana` at two positions
- Formally
 - A `text` string `t` of length `n`
 - A `pattern` string `p` of length `m`
 - Both `t` and `p` are drawn from an `alphabet` of valid letters, denoted Σ
 - Find every position `i` in `t` such that `t[i:i+m] == p`

Brute force

- Nested loop

- For each starting position i in t ,
compare $t[i:i+m]$ with p

```
def stringmatch(t,p):  
    poslist = []  
    for i in range(len(t)-len(p)+1):  
        matched = True  
        j = 0  
        while j < len(p) and matched:  
            if t[i+j] != p[j]:  
                matched = False  
            j = j+1  
        if matched:  
            poslist.append(i)  
    return(poslist)
```

Brute force

- Nested loop
 - For each starting position i in t , compare $t[i:i+m]$ with p
- Nested search bails out at first mismatch

```
def stringmatch(t,p):  
    poslist = []  
    for i in range(len(t)-len(p)+1):  
        matched = True  
        j = 0  
        while j < len(p) and matched:  
            if t[i+j] != p[j]:  
                matched = False  
            j = j+1  
        if matched:  
            poslist.append(i)  
    return(poslist)
```

Brute force

- Nested loop
 - For each starting position i in t , compare $t[i:i+m]$ with p
- Nested search bails out at first mismatch
- Worst case is $O(nm)$, for example
 - $t = \text{aaa...a}$, $p = \text{aaab}$

```
def stringmatch(t,p):  
    poslist = []  
    for i in range(len(t)-len(p)+1):  
        matched = True  
        j = 0  
        while j < len(p) and matched:  
            if t[i+j] != p[j]:  
                matched = False  
            j = j+1  
        if matched:  
            poslist.append(i)  
    return(poslist)
```

Brute force

- Nested loop
 - For each starting position i in t , compare $t[i:i+m]$ with p
- Nested search bails out at first mismatch
- Worst case is $O(nm)$, for example
 - $t = \text{aaa...a}$, $p = \text{aaab}$
- Can also do nested scan from right to left
 - Worst case still $O(nm)$,
 $t = \text{aaa...a}$, $p = \text{baaa}$
 - Can reversing the scan help?

```
def stringmatchrev(t,p):  
    poslist = []  
    for i in range(len(t)-len(p)+1):  
        matched = True  
        j = len(p)-1  
        while j >= 0 and matched:  
            if t[i+j] != p[j]:  
                matched = False  
            j = j-1  
        if matched:  
            poslist.append(i)  
    return(poslist)
```

Speeding things up

- While matching, we find a letter in `t` that does not appear in `p`
 - `t = bananamania, p = bulk`

Speeding things up

- While matching, we find a letter in **t** that does not appear in **p**
 - **t = bananamania, p = bulk**
- When we see **a** in **t**, we can shift the next scan to after **a**



Speeding things up

- While matching, we find a letter in **t** that does not appear in **p**
 - **t = bananamania, p = bulk**
- When we see **a** in **t**, we can shift the next scan to after **a**
 - If we scan from the left, we skip one position

b	a	n	a	n	a	m	a	n	i	a
---	---	---	---	---	---	---	---	---	---	---

b	u	l	k
---	---	---	---

Speeding things up

- While matching, we find a letter in **t** that does not appear in **p**
 - **t = bananamania, p = bulk**
- When we see **a** in **t**, we can shift the next scan to after **a**
 - If we scan from the left, we skip one position
 - If we scan from the right, we skip three positions



Speeding things up

- While matching, we find a letter in **t** that does not appear in **p**
 - **t = bananamania, p = bulk**
- When we see **a** in **t**, we can shift the next scan to after **a**
 - If we scan from the left, we skip one position
 - If we scan from the right, we skip three positions

b	a	n	a	n	a	m	a	n	i	a
---	---	---	---	---	---	---	---	---	---	---

b	u	l	k
---	---	---	---

Speeding things up

- While matching, we find a letter in t that does not appear in p
 - $t = \text{bananamania}$, $p = \text{bulk}$
- When we see a in t , we can shift the next scan to after a
 - If we scan from the left, we skip one position
 - If we scan from the right, we skip three positions
- Don't need to check all of t to search for all occurrences of p !
- Formalized in Boyer-Moore algorithm

b	a	n	a	n	a	m	a	n	i	a
---	---	---	---	---	---	---	---	---	---	---

b	u	l	k
---	---	---	---

String Matching: Boyer-Moore algorithm

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 10

Speeding up the brute force algorithm

- Text t , pattern p of lengths n, m
- For each starting position i in t ,
compare $t[i:i+m]$ with p
 - Scan $t[i:i+m]$ right to left

Speeding up the brute force algorithm

- Text t , pattern p of lengths n, m
- For each starting position i in t , compare $t[i:i+m]$ with p
 - Scan $t[i:i+m]$ right to left
- While matching, we find a letter in t that does not appear in p
 - $t = \text{bananamania}, p = \text{bulk}$



Speeding up the brute force algorithm

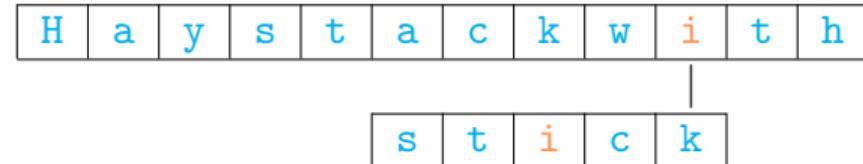
- Text t , pattern p of lengths n, m
- For each starting position i in t , compare $t[i:i+m]$ with p
 - Scan $t[i:i+m]$ right to left
- While matching, we find a letter in t that does not appear in p
 - $t = \text{bananamania}, p = \text{bulk}$
- Shift the next scan to position after mismatched letter
- What if the mismatched letter does appear in p ?

b	a	n	a	n	a	m	a	n	i	a
---	---	---	---	---	---	---	---	---	---	---

b	u	l	k
---	---	---	---

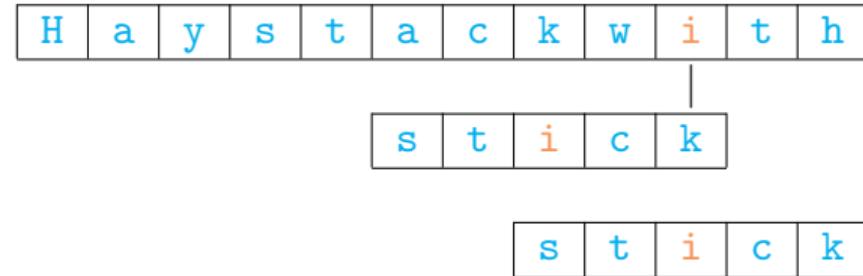
Sliding the search

- Suppose $c = t[i+j] \neq p[j]$, but c does occur somewhere in $p[j]$



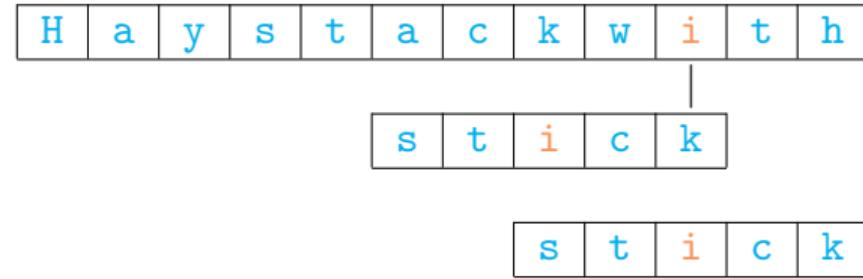
Sliding the search

- Suppose $c = t[i+j] \neq p[j]$, but c does occur somewhere in $p[j]$
- Align rightmost occurrence of c in p with $t[i+j]$



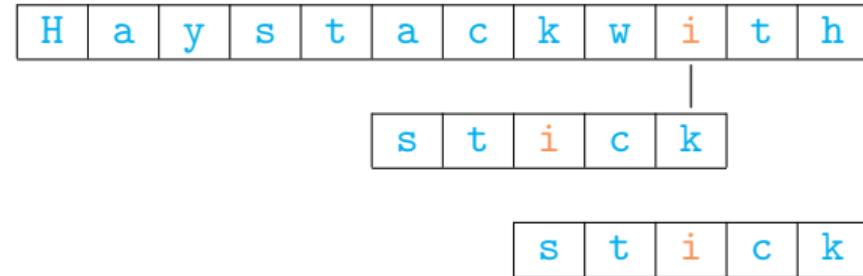
Sliding the search

- Suppose $c = t[i+j] \neq p[j]$, but c does occur somewhere in $p[j]$
- Align rightmost occurrence of c in p with $t[i+j]$
- Scan this substring of t next



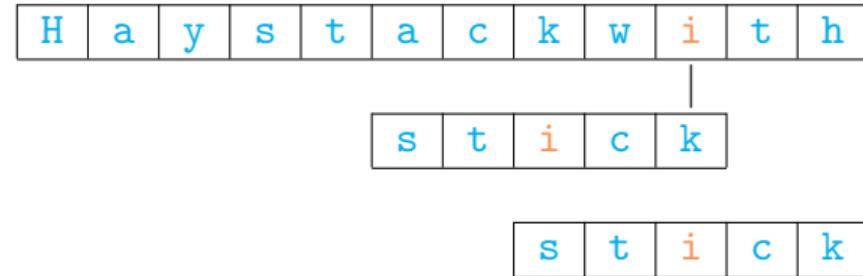
Sliding the search

- Suppose $c = t[i+j] \neq p[j]$, but c does occur somewhere in $p[j]$
- Align rightmost occurrence of c in p with $t[i+j]$
- Scan this substring of t next
- Use a dictionary `last`
 - For each c in p , `last[c]` records right most position of c in p
 - Shift pattern by $j - \text{last}[c]$



Sliding the search

- Suppose $c = t[i+j] \neq p[j]$, but c does occur somewhere in $p[j]$
- Align rightmost occurrence of c in p with $t[i+j]$
- Scan this substring of t next
- Use a dictionary `last`
 - For each c in p , $\text{last}[c]$ records right most position of c in p
 - Shift pattern by $j - \text{last}[c]$
- If c not in p , shift pattern by $j+1$

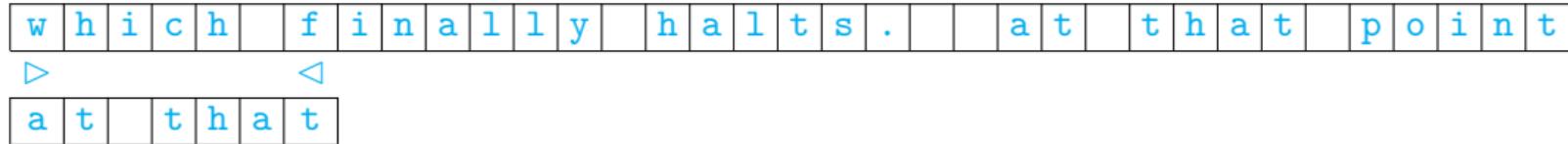


Example [Boyer, Moore 1977]

- t = "which finally halts. at that point"
p = "at that"

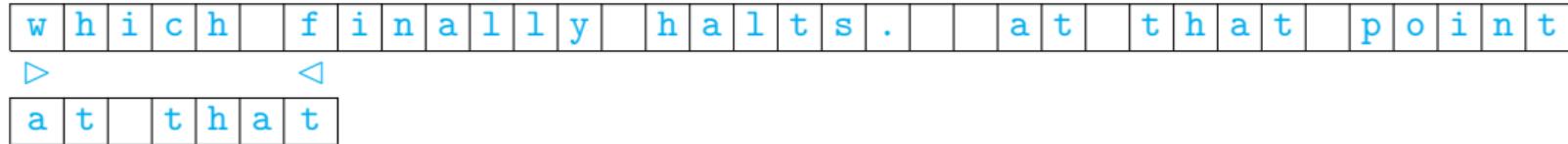
Example [Boyer, Moore 1977]

- t = "which finally halts. at that point"
p = "at that"



Example [Boyer, Moore 1977]

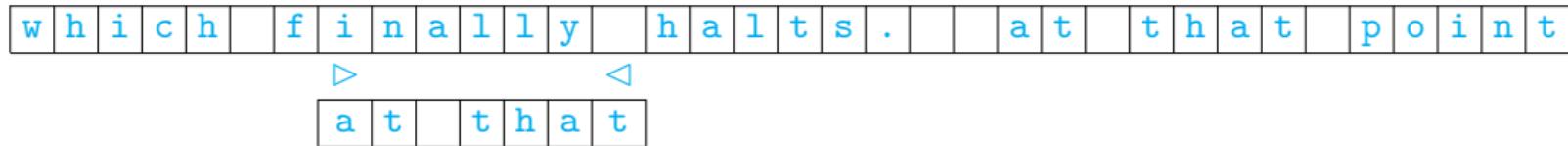
- `t = "which finally halts. at that point"`
`p = "at that"`



- `t[0:7] == "which f"`, "f" not in pattern, shift by 7, index 7

Example [Boyer, Moore 1977]

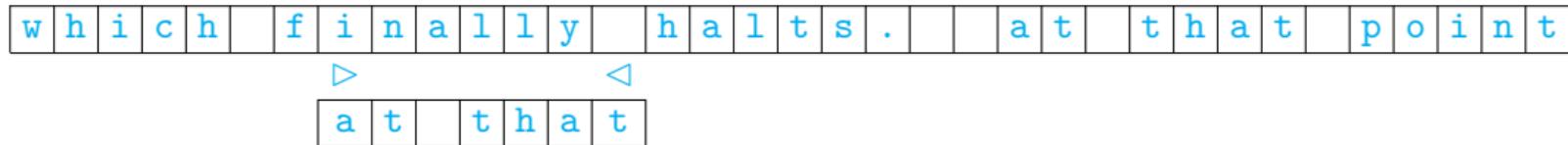
- `t = "which finally halts. at that point"`
`p = "at that"`



- `t[0:7] == "which f"`, "f" not in pattern, shift by 7, index 7

Example [Boyer, Moore 1977]

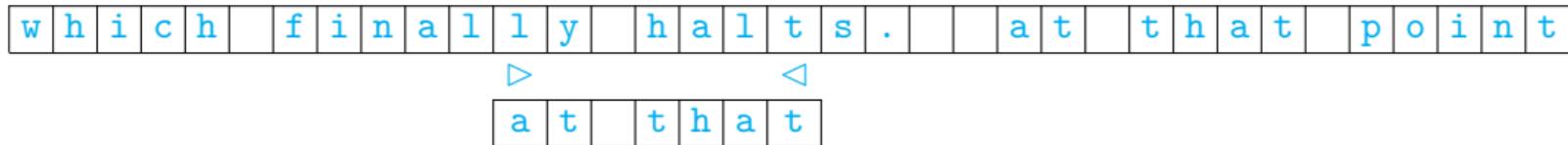
- `t = "which finally halts. at that point"`
- `p = "at that"`



- `t[0:7] == "which f"`, "f" not in pattern, shift by 7, index 7
- `t[7:14] == "inally "`, " " in pattern, shift by 4 to align " ", index 11

Example [Boyer, Moore 1977]

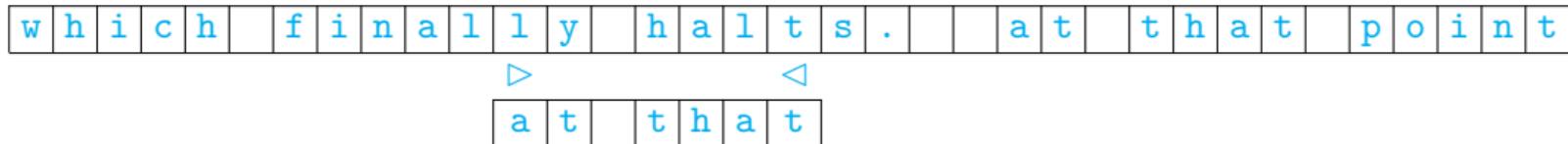
- `t = "which finally halts. at that point"`
- `p = "at that"`



- `t[0:7] == "which f"`, "f" not in pattern, shift by 7, index 7
- `t[7:14] == "inally "`, " " in pattern, shift by 4 to align " ", index 11

Example [Boyer, Moore 1977]

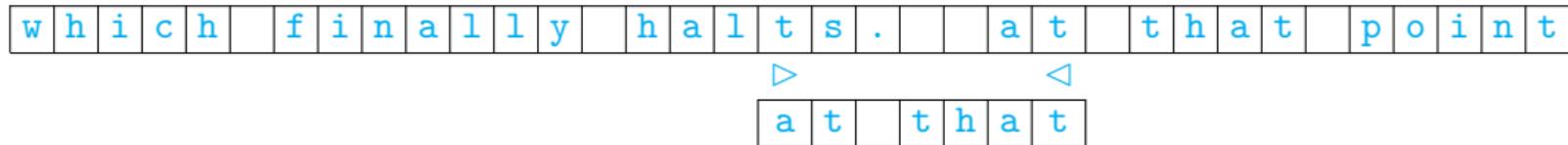
- t = "which finally halts. at that point"
p = "at that"



- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11
- t[11:18] == "ly halt", "l" not in pattern, shift by 6, index 17

Example [Boyer, Moore 1977]

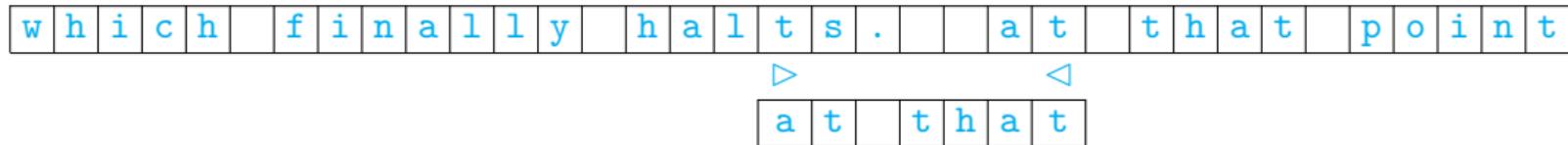
- t = "which finally halts. at that point"
p = "at that"



- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11
- t[11:18] == "ly halt", "l" not in pattern, shift by 6, index 17

Example [Boyer, Moore 1977]

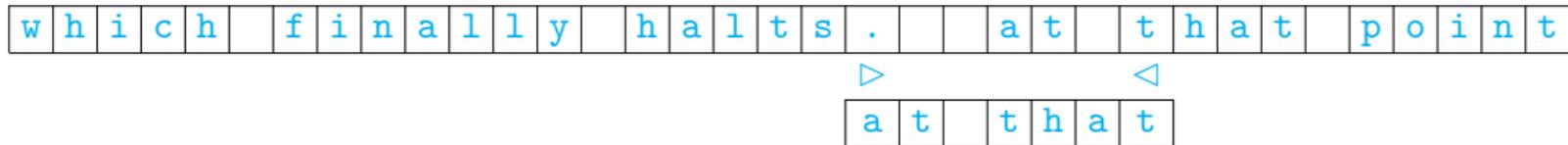
- t = "which finally halts. at that point"
p = "at that"



- `t[0:7] == "which f"`, "f" not in pattern, shift by 7, index 7
- `t[7:14] == "inally "`, " " in pattern, shift by 4 to align " ", index 11
- `t[11:18] == "ly halt"`, "l" not in pattern, shift by 6, index 17
- `t[17:24] == "ts. at"`, " " in pattern, shift by 2, index 19

Example [Boyer, Moore 1977]

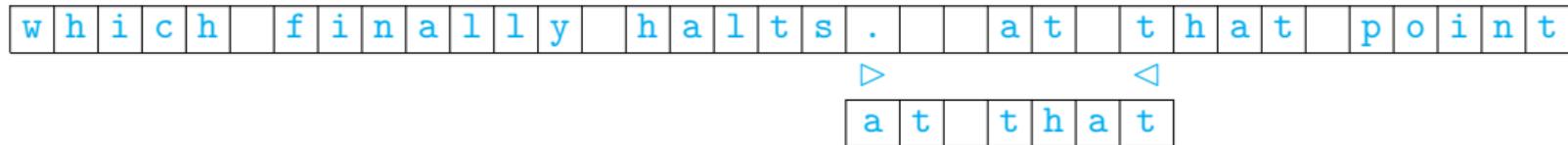
- t = "which finally halts. at that point"
p = "at that"



- `t[0:7] == "which f"`, "f" not in pattern, shift by 7, index 7
- `t[7:14] == "inally "`, " " in pattern, shift by 4 to align " ", index 11
- `t[11:18] == "ly halt"`, "l" not in pattern, shift by 6, index 17
- `t[17:24] == "ts. at"`, " " in pattern, shift by 2, index 19

Example [Boyer, Moore 1977]

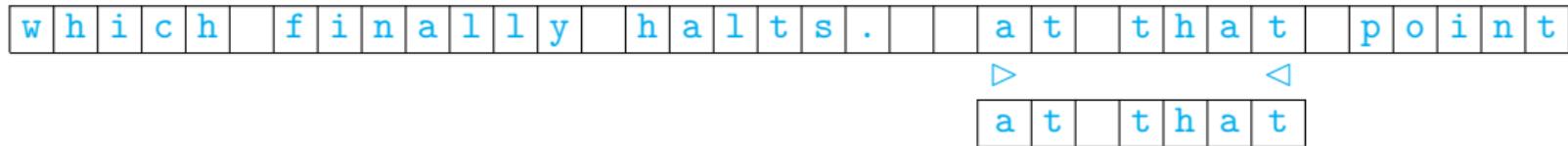
- t = "which finally halts. at that point"
p = "at that"



- `t[0:7] == "which f"`, "f" not in pattern, shift by 7, index 7
- `t[7:14] == "inally "`, " " in pattern, shift by 4 to align " ", index 11
- `t[11:18] == "ly halt"`, "l" not in pattern, shift by 6, index 17
- `t[17:24] == "ts. at"`, " " in pattern, shift by 2, index 19
- `t[19:26] == ". at t"`, " " in pattern, shift by 3, index 22

Example [Boyer, Moore 1977]

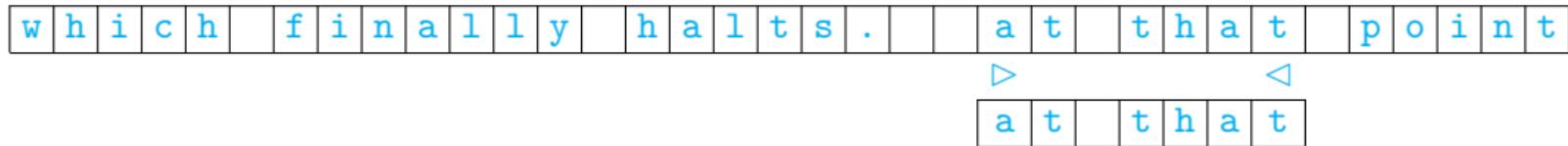
- t = "which finally halts. at that point"
p = "at that"



- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11
- t[11:18] == "ly halt", "l" not in pattern, shift by 6, index 17
- t[17:24] == "ts. at", " " in pattern, shift by 2, index 19
- t[19:26] == ". at t", " " in pattern, shift by 3, index 22

Example [Boyer, Moore 1977]

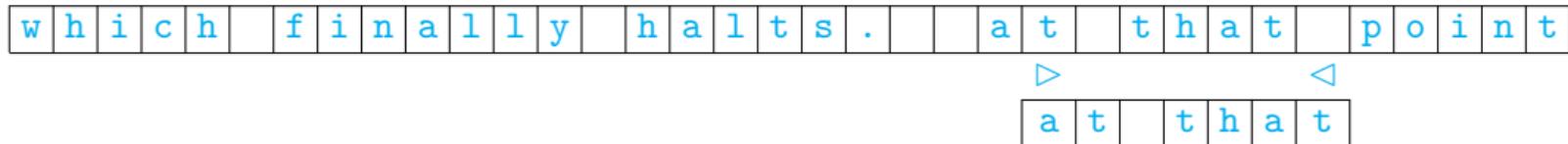
- t = "which finally halts. at that point"
p = "at that"



- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11
- t[11:18] == "ly halt", "l" not in pattern, shift by 6, index 17
- t[17:24] == "ts. at", " " in pattern, shift by 2, index 19
- t[19:26] == ". at t", " " in pattern, shift by 3, index 22
- t[22:29] == "at that", report match at index 22, shift by 1, index 23

Example [Boyer, Moore 1977]

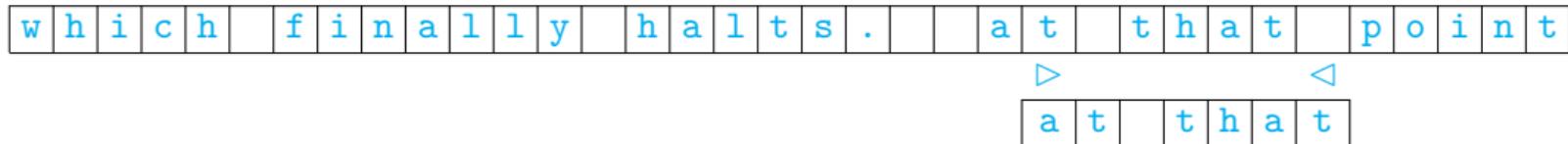
- t = "which finally halts. at that point"
p = "at that"



- `t[0:7] == "which f"`, "f" not in pattern, shift by 7, index 7
- `t[7:14] == "inally "`, " " in pattern, shift by 4 to align " ", index 11
- `t[11:18] == "ly halt"`, "l" not in pattern, shift by 6, index 17
- `t[17:24] == "ts. at"`, " " in pattern, shift by 2, index 19
- `t[19:26] == ". at t"`, " " in pattern, shift by 3, index 22
- `t[22:29] == "at that"`, report match at index 22, shift by 1, index 23

Example [Boyer, Moore 1977]

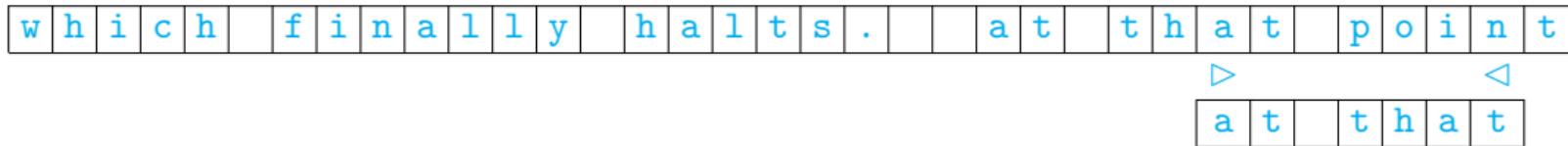
- t = "which finally halts. at that point"
p = "at that"



- `t[0:7] == "which f"`, "f" not in pattern, shift by 7, index 7
- `t[7:14] == "inally "`, " " in pattern, shift by 4 to align " ", index 11
- `t[11:18] == "ly halt"`, "l" not in pattern, shift by 6, index 17
- `t[17:24] == "ts. at"`, " " in pattern, shift by 2, index 19
- `t[19:26] == ". at t"`, " " in pattern, shift by 3, index 22
- `t[22:29] == "at that"`, report match at index 22, shift by 1, index 23
- `t[23:30] == "t that "`, " " in pattern, shift by 4, index 27

Example [Boyer, Moore 1977]

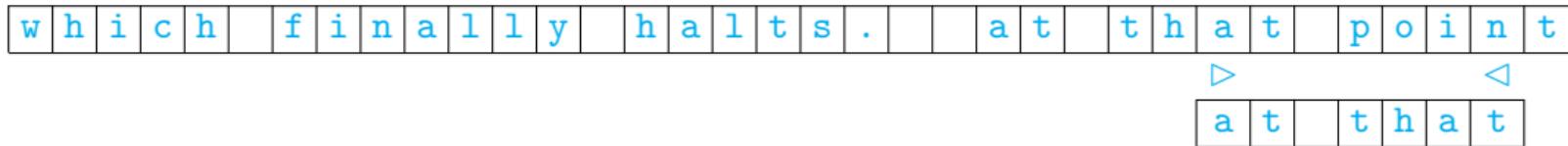
- t = "which finally halts. at that point"
p = "at that"



- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11
- t[11:18] == "ly halt", "l" not in pattern, shift by 6, index 17
- t[17:24] == "ts. at", " " in pattern, shift by 2, index 19
- t[19:26] == ". at t", " " in pattern, shift by 3, index 22
- t[22:29] == "at that", report match at index 22, shift by 1, index 23
- t[23:30] == "t that ", " " in pattern, shift by 4, index 27

Example [Boyer, Moore 1977]

- t = "which finally halts. at that point"
p = "at that"



- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11
- t[11:18] == "ly halt", "l" not in pattern, shift by 6, index 17
- t[17:24] == "ts. at", " " in pattern, shift by 2, index 19
- t[19:26] == ". at t", " " in pattern, shift by 3, index 22
- t[22:29] == "at that", report match at index 22, shift by 1, index 23
- t[23:30] == "t that ", " " in pattern, shift by 4, index 27
- t[27:34] == "at poin", "n" not in pattern, shift by 7, index 34, stop

Implementation

- Initialize `last[c]` for each `c` in `p`
 - Single scan, rightmost value is recorded

```
def boyermoore(t,p):  
    last = {}                      # Preprocess  
    for i in range(len(p)):  
        last[p[i]] = i  
  
    poslist,i = [],0                # Loop  
    while i <= (len(t)-len(p)):  
        matched,j = True,len(p)-1  
        while j >= 0 and matched:  
            if t[i+j] != p[j]:  
                matched = False  
            j = j - 1  
        if matched:  
            poslist.append(i)  
            i = i + 1  
        else:  
            j = j + 1  
            if t[i+j] in last.keys():  
                i = i + max(j-last[t[i+j]],1)  
            else:  
                i = i + j + 1  
    return(poslist)
```

Implementation

- Initialize `last[c]` for each `c` in `p`
 - Single scan, rightmost value is recorded
- Nested loop, compare each segment `t[i:i+len(p)]` with `p`

```
def boyermoore(t,p):  
    last = {}                      # Preprocess  
    for i in range(len(p)):  
        last[p[i]] = i  
  
    poslist,i = [],0                # Loop  
    while i <= (len(t)-len(p)):  
        matched,j = True,len(p)-1  
        while j >= 0 and matched:  
            if t[i+j] != p[j]:  
                matched = False  
            j = j - 1  
        if matched:  
            poslist.append(i)  
            i = i + 1  
        else:  
            j = j + 1  
            if t[i+j] in last.keys():  
                i = i + max(j-last[t[i+j]],1)  
            else:  
                i = i + j + 1  
    return(poslist)
```

Implementation

- Initialize `last[c]` for each `c` in `p`
 - Single scan, rightmost value is recorded
- Nested loop, compare each segment `t[i:i+len(p)]` with `p`
- If `p` matches, record and shift by 1

```
def boyermoore(t,p):  
    last = {}                      # Preprocess  
    for i in range(len(p)):  
        last[p[i]] = i  
  
    poslist,i = [],0                # Loop  
    while i <= (len(t)-len(p)):  
        matched,j = True,len(p)-1  
        while j >= 0 and matched:  
            if t[i+j] != p[j]:  
                matched = False  
            j = j - 1  
        if matched:  
            poslist.append(i)  
            i = i + 1  
        else:  
            j = j + 1  
            if t[i+j] in last.keys():  
                i = i + max(j-last[t[i+j]],1)  
            else:  
                i = i + j + 1  
    return(poslist)
```

Implementation

- Initialize `last[c]` for each `c` in `p`
 - Single scan, rightmost value is recorded
- Nested loop, compare each segment `t[i:i+len(p)]` with `p`
- If `p` matches, record and shift by 1
- We find a mismatch at `t[i+j]`
 - If `j > last[t[i+j]]`, shift by `j - last[t[i+j]]`
 - If `last[t[i+j]] > j`, shift by 1
 - Should not shift `p` to left!
 - If `t[i+j]` not in `p`, shift by `j+1`

```
def boyermoore(t,p):  
    last = {}                      # Preprocess  
    for i in range(len(p)):  
        last[p[i]] = i  
  
    poslist,i = [],0                # Loop  
    while i <= (len(t)-len(p)):  
        matched,j = True,len(p)-1  
        while j >= 0 and matched:  
            if t[i+j] != p[j]:  
                matched = False  
            j = j - 1  
        if matched:  
            poslist.append(i)  
            i = i + 1  
        else:  
            j = j + 1  
            if t[i+j] in last.keys():  
                i = i + max(j-last[t[i+j]],1)  
            else:  
                i = i + j + 1  
    return(poslist)
```

Summary

- Worst case remains $O(nm)$
 - $t = \text{aaa...a}, p = \text{baaa}$

Summary

- Worst case remains $O(nm)$
 - $t = \text{aaa...a}, p = \text{baaa}$
- Without dictionary, computing `last` is a bottleneck, complexity is $O(|\Sigma|)$

Summary

- Worst case remains $O(nm)$
 - $t = \text{aaa...a}$, $p = \text{baaa}$
- Without dictionary, computing `last` is a bottleneck, complexity is $O(|\Sigma|)$
- Boyer-Moore works well, in practice
 - “Sublinear”
 - Experimentally — English text, 5 character pattern, average number of comparisons is 0.24 per character
 - Performance improves as pattern length grows — more characters skipped

Summary

- Worst case remains $O(nm)$
 - $t = \text{aaa...a}$, $p = \text{baaa}$
- Without dictionary, computing `last` is a bottleneck, complexity is $O(|\Sigma|)$
- Boyer-Moore works well, in practice
 - “Sublinear”
 - Experimentally — English text, 5 character pattern, average number of comparisons is 0.24 per character
 - Performance improves as pattern length grows — more characters skipped
- Often used in practice — `grep` in Unix

String Matching: Rabin-Karp algorithm

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 10

Reducing string matching to arithmetic

- Suppose $\Sigma = \{0, 1, \dots, 9\}$

Reducing string matching to arithmetic

- Suppose $\Sigma = \{0, 1, \dots, 9\}$
- Any string over Σ can be thought of as a number in base 10

Reducing string matching to arithmetic

- Suppose $\Sigma = \{0, 1, \dots, 9\}$
- Any string over Σ can be thought of as a number in base 10
- Pattern p is an m -digit number n_p

Reducing string matching to arithmetic

- Suppose $\Sigma = \{0, 1, \dots, 9\}$
- Any string over Σ can be thought of as a number in base 10
- Pattern p is an m -digit number n_p
- Each substring of length m in the text t is again an m -digit number

Reducing string matching to arithmetic

- Suppose $\Sigma = \{0, 1, \dots, 9\}$
- Any string over Σ can be thought of as a number in base 10
- Pattern p is an m -digit number n_p
- Each substring of length m in the text t is again an m -digit number
- Scan t and compare the number n_b generated by each block of m letters with the pattern number n_p

Reducing string matching to arithmetic

- Suppose $\Sigma = \{0, 1, \dots, 9\}$

Converting a string to a number

- Any string over Σ can be thought of as a number in base 10
- Pattern p is an m -digit number n_p
- Each substring of length m in the text t is again an m -digit number
- Scan t and compare the number n_b generated by each block of m letters with the pattern number n_p

Reducing string matching to arithmetic

- Suppose $\Sigma = \{0, 1, \dots, 9\}$
- Any string over Σ can be thought of as a number in base 10
- Pattern p is an m -digit number n_p
- Each substring of length m in the text t is again an m -digit number
- Scan t and compare the number n_b generated by each block of m letters with the pattern number n_p

Converting a string to a number

- Can convert a block $t[i:i+m]$ to an integer n_i in one scan

```
num = 0  
for j in range(m):  
    num = 10*num + t[i+j]
```

Reducing string matching to arithmetic

- Suppose $\Sigma = \{0, 1, \dots, 9\}$
- Any string over Σ can be thought of as a number in base 10
- Pattern p is an m -digit number n_p
- Each substring of length m in the text t is again an m -digit number
- Scan t and compare the number n_b generated by each block of m letters with the pattern number n_p

Converting a string to a number

- Can convert a block $t[i:i+m]$ to an integer n_i in one scan
- ```
num = 0
for j in range(m):
 num = 10*num + t[i+j]
```
- Computing  $n_i$  from  $t[i:i+m]$  for each block from scratch will take time  $O(nm)$

# Reducing string matching to arithmetic

- Suppose  $\Sigma = \{0, 1, \dots, 9\}$
- Any string over  $\Sigma$  can be thought of as a number in base 10
- Pattern  $p$  is an  $m$ -digit number  $n_p$
- Each substring of length  $m$  in the text  $t$  is again an  $m$ -digit number
- Scan  $t$  and compare the number  $n_b$  generated by each block of  $m$  letters with the pattern number  $n_p$

## Converting a string to a number

- Can convert a block  $t[i:i+m]$  to an integer  $n_i$  in one scan
- ```
num = 0
for j in range(m):
    num = 10*num + t[i+j]
```
- Computing n_i from $t[i:i+m]$ for each block from scratch will take time $O(nm)$
 - Instead
 - Subtract $10^{m-1} \cdot t[i - 1]$ from n_{i-1} — drop leading digit
 - Multiply by 10 and add $t[i + m - 1]$ to get n_i

Rabin-Karp algorithm

```
def rabinkarp(t,p):
    poslist = []

    numt, nump = 0,0
    for i in range(len(p)):
        numt = 10*numt + int(t[i])
        nump = 10*nump + int(p[i])

    if numt == nump:
        poslist.append(0)

    for i in range(1,len(t)-len(p)+1):
        numt = numt - int(t[i-1])*(10**((len(p)-1)))
        numt = 10*numt + int(t[i+len(p)-1])
        if numt == nump:
            poslist.append(i)
    return(poslist)
```

Rabin-Karp algorithm

```
def rabinkarp(t,p):
    poslist = []
    numt, nump = 0,0
    for i in range(len(p)):
        numt = 10*numt + int(t[i])
        nump = 10*nump + int(p[i])
    if numt == nump:
        poslist.append(0)
    for i in range(1,len(t)-len(p)+1):
        numt = numt - int(t[i-1])*(10**((len(p)-1)))
        numt = 10*numt + int(t[i+len(p)-1])
        if numt == nump:
            poslist.append(i)
    return(poslist)
```

- First convert $t[0:m]$ to n_0 and p to n_p

Rabin-Karp algorithm

```
def rabinkarp(t,p):
    poslist = []

    numt, nump = 0,0
    for i in range(len(p)):
        numt = 10*numt + int(t[i])
        nump = 10*nump + int(p[i])

    if numt == nump:
        poslist.append(0)

    for i in range(1,len(t)-len(p)+1):
        numt = numt - int(t[i-1])*(10**((len(p)-1)))
        numt = 10*numt + int(t[i+len(p)-1])
        if numt == nump:
            poslist.append(i)
    return(poslist)
```

- First convert $t[0:m]$ to n_0 and p to n_p
- In the loop, incrementally convert n_{i-1} to n_i

Rabin-Karp algorithm

```
def rabinkarp(t,p):
    poslist = []

    numt,nump = 0,0
    for i in range(len(p)):
        numt = 10*numt + int(t[i])
        nump = 10*nump + int(p[i])

    if numt == nump:
        poslist.append(0)

    for i in range(1,len(t)-len(p)+1):
        numt = numt - int(t[i-1])*(10**((len(p)-1)))
        numt = 10*numt + int(t[i+len(p)-1])
        if numt == nump:
            poslist.append(i)
    return(poslist)
```

- First convert $t[0:m]$ to n_0 and p to n_p
- In the loop, incrementally convert n_{i-1} to n_i
- Whenever $n_i = n_p$ report a match

Rabin-Karp algorithm for general alphabets

- For $\Sigma = \{a_1, a_2, \dots, a_k\}$, treat each letter as a digit in base k

Rabin-Karp algorithm for general alphabets

- For $\Sigma = \{a_1, a_2, \dots, a_k\}$, treat each letter as a digit in base k
- Naively, repeat the previous algorithm in base k
 - Multiply/divide by k rather than 10

Rabin-Karp algorithm for general alphabets

- For $\Sigma = \{a_1, a_2, \dots, a_k\}$, treat each letter as a digit in base k
- Naively, repeat the previous algorithm in base k
 - Multiply/divide by k rather than 10
- In practice, for realistic k , the numbers are too large to work with directly

Rabin-Karp algorithm for general alphabets

- For $\Sigma = \{a_1, a_2, \dots, a_k\}$, treat each letter as a digit in base k
- Naively, repeat the previous algorithm in base k
 - Multiply/divide by k rather than 10
- In practice, for realistic k , the numbers are too large to work with directly
- Instead, do all computations and comparisons modulo a suitable prime q

Rabin-Karp algorithm for general alphabets

- For $\Sigma = \{a_1, a_2, \dots, a_k\}$, treat each letter as a digit in base k
 - Naively, repeat the previous algorithm in base k
 - Multiply/divide by k rather than 10
 - In practice, for realistic k , the numbers are too large to work with directly
 - Instead, do all computations and comparisons modulo a suitable prime q
- $p = 31415, t = 2359023141526739921$
 - $q = 13$

Rabin-Karp algorithm for general alphabets

- For $\Sigma = \{a_1, a_2, \dots, a_k\}$, treat each letter as a digit in base k
 - Naively, repeat the previous algorithm in base k
 - Multiply/divide by k rather than 10
 - In practice, for realistic k , the numbers are too large to work with directly
 - Instead, do all computations and comparisons modulo a suitable prime q
-
- $p = 31415, t = 2359023141526739921$
 - $q = 13$
 - $31415 \bmod 13 = 7$

Rabin-Karp algorithm for general alphabets

- For $\Sigma = \{a_1, a_2, \dots, a_k\}$, treat each letter as a digit in base k
- Naively, repeat the previous algorithm in base k
 - Multiply/divide by k rather than 10
- In practice, for realistic k , the numbers are too large to work with directly
- Instead, do all computations and comparisons modulo a suitable prime q
- $p = 31415, t = 2359023141526739921$
- $q = 13$
- $31415 \bmod 13 = 7$
- $23590 \bmod 13 = 8$
 $35902 \bmod 13 = 9$
...
- $31415 \bmod 13 = 7$
- ...
 $67399 \bmod 13 = 7$
- ...

Rabin-Karp algorithm for general alphabets

- For $\Sigma = \{a_1, a_2, \dots, a_k\}$, treat each letter as a digit in base k
- Naively, repeat the previous algorithm in base k
 - Multiply/divide by k rather than 10
- In practice, for realistic k , the numbers are too large to work with directly
- Instead, do all computations and comparisons modulo a suitable prime q
- $p = 31415, t = 2359023141526739921$
- $q = 13$
- $31415 \bmod 13 = 7$
- $23590 \bmod 13 = 8$
 $35902 \bmod 13 = 9$
...
- $31415 \bmod 13 = 7$
- $67399 \bmod 13 = 7$
- ...
- False positives — must scan and validate each block that appears to match

Summary

- Preprocessing time is $O(m)$
 - To convert `t[0:m]`, `p` to numbers

Summary

- Preprocessing time is $O(m)$
 - To convert `t[0:m]`, `p` to numbers
- Worst case for general alphabets is $O(nm)$
 - Every block `t[i:i+m]` may have same remainder modulo `q` as the pattern `p`
 - Must validate each block explicitly, like brute force

Summary

- Preprocessing time is $O(m)$
 - To convert $t[0:m]$, p to numbers
- Worst case for general alphabets is $O(nm)$
 - Every block $t[i:i+m]$ may have same remainder modulo q as the pattern p
 - Must validate each block explicitly, like brute force
- In practice number of spurious matches will be small

Summary

- Preprocessing time is $O(m)$
 - To convert $t[0:m]$, p to numbers
- Worst case for general alphabets is $O(nm)$
 - Every block $t[i:i+m]$ may have same remainder modulo q as the pattern p
 - Must validate each block explicitly, like brute force
- In practice number of spurious matches will be small
- If $|\Sigma|$ is small enough to not require modulo arithmetic, overall time is $O(n + m)$, or $O(n)$, since $m \ll n$
 - Also if we can choose q carefully to ensure $O(1)$ spurious matches

String matching using automata

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 10

Traditional string matching

- Text t , pattern p of lengths n, m
- For each starting position i in t , compare $t[i:i+m]$ with p
 - Scan $t[i:i+m]$ right to left

Traditional string matching

- Text t , pattern p of lengths n, m
- For each starting position i in t , compare $t[i:i+m]$ with p
 - Scan $t[i:i+m]$ right to left
- Can skip some positions i on mismatch [Boyer,Mooore]
 - Case 1: $t[i+j]$ does not appear in p
 - Update i to $i+j+1$
 - Case 2: $t[i+j]$ appears in p before $p[j]$
 - Precompute $\text{last}[c]$ for each c in p
 - Align $p[\text{last}[c]]$ with $t[i+j]$
 - Update i to $j - \text{last}[c]$

Traditional string matching

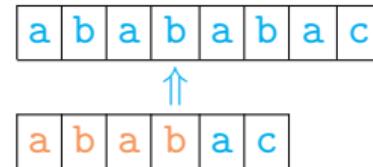
- Text t , pattern p of lengths n, m
- For each starting position i in t , compare $t[i:i+m]$ with p
 - Scan $t[i:i+m]$ right to left
- Can skip some positions i on mismatch [Boyer,Mooore]
 - Case 1: $t[i+j]$ does not appear in p
 - Update i to $i+j+1$
 - Case 2: $t[i+j]$ appears in p before $p[j]$
 - Precompute $\text{last}[c]$ for each c in p
 - Align $p[\text{last}[c]]$ with $t[i+j]$
 - Update i to $j - \text{last}[c]$
- Worst case remains $O(nm)$: $t = \text{aaa...a}$, $p = \text{baaaa}$

Remembering previous matches

- Can we intelligently re-use partial matches

Remembering previous matches

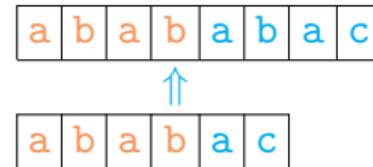
- Can we intelligently re-use partial matches



- Keep track of longest prefix of $p[:k]$ that we have matched

Remembering previous matches

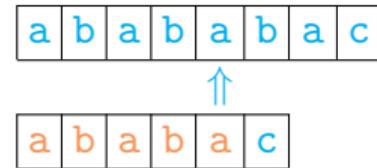
- Can we intelligently re-use partial matches



- Keep track of longest **prefix** of $p[:k]$ that we have matched
- This prefix is a **suffix** of $t[:i+j]$

Remembering previous matches

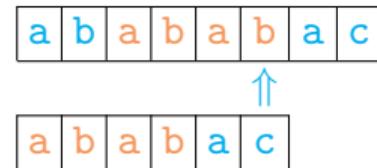
- Can we intelligently re-use partial matches



- Keep track of longest **prefix** of $p[:k]$ that we have matched
- This prefix is a **suffix** of $t[:i+j]$
- If $t[i+j] == p[k]$, extend the match

Remembering previous matches

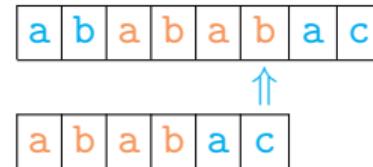
- Can we intelligently re-use partial matches



- Keep track of longest **prefix** of $p[:k]$ that we have matched
- This prefix is a **suffix** of $t[:i+j]$
- If $t[i+j] == p[k]$, extend the match
- If $t[i+j] != p[k]$, reset longest match for $t[:i+j+1]$

Remembering previous matches

- Can we intelligently re-use partial matches



- Keep track of longest **prefix** of $p[:k]$ that we have matched
- This prefix is a **suffix** of $t[:i+j]$
- If $t[i+j] == p[k]$, extend the match
- If $t[i+j] != p[k]$, reset longest match for $t[:i+j+1]$
- To reset, use precomputed values

Precomputing longest match as a graph

- Pattern p of length m

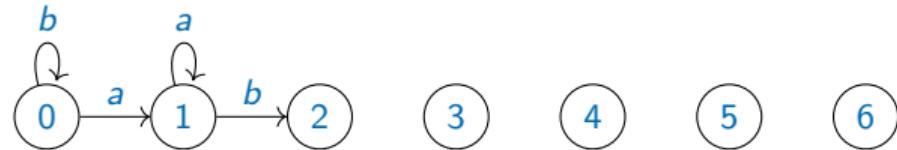
Precomputing longest match as a graph

- Pattern p of length m
- Graph with $m+1$ nodes, $\{0, 1, \dots, m\}$
 - Node i denotes match upto $p[:i]$



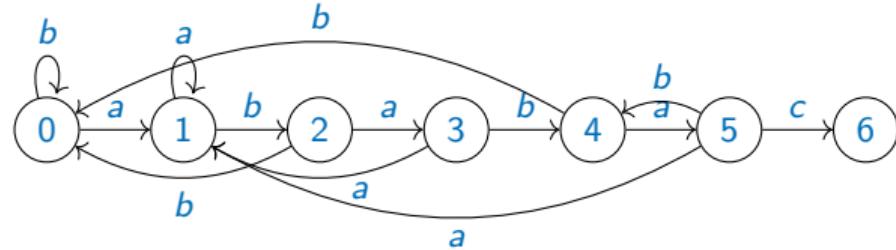
Precomputing longest match as a graph

- Pattern p of length m
- Graph with $m+1$ nodes, $\{0, 1, \dots, m\}$
 - Node i denotes match upto $p[:i]$
- Edges describe how to extend the match
- $t[j] = a$, $t[:j]$ matches $p[:i]$
 - Edge $i \xrightarrow{a} k$, $t[:j+1]$ matches $p[:k]$
 - If $t[j] = p[i]$, $k = i + 1$
 - Else find longest prefix of p that matches suffix of $t[:j+1]$



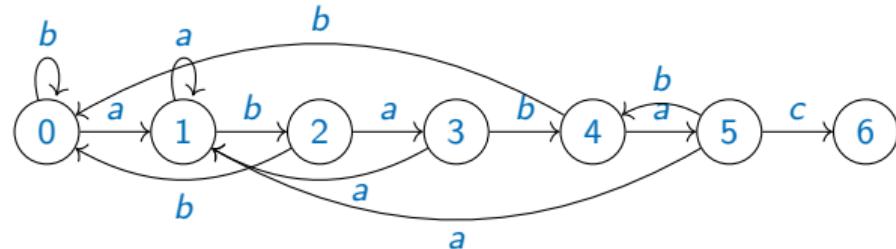
Precomputing longest match as a graph

- Pattern p of length m
- Graph with $m+1$ nodes, $\{0, 1, \dots, m\}$
 - Node i denotes match upto $p[:i]$
- Edges describe how to extend the match
- $t[j] = a$, $t[:j]$ matches $p[:i]$
 - Edge $i \xrightarrow{a} k$, $t[:j+1]$ matches $p[:k]$
 - If $t[j] = p[i]$, $k = i + 1$
 - Else find longest prefix of p that matches suffix of $t[:j+1]$



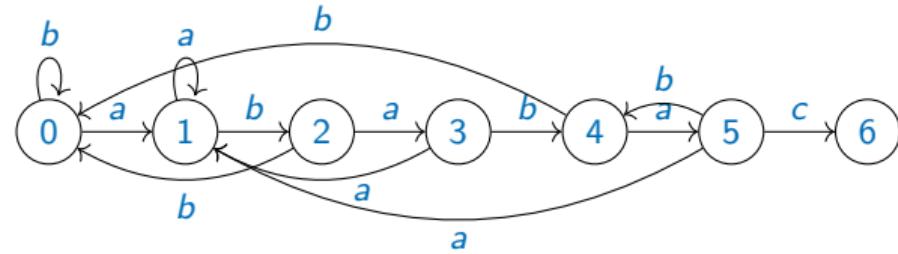
Precomputing longest match as a graph

- Pattern p of length m
- Graph with $m+1$ nodes, $\{0, 1, \dots, m\}$
 - Node i denotes match upto $p[:i]$
- Edges describe how to extend the match
- $t[j] = a$, $t[:j]$ matches $p[:i]$
 - Edge $i \xrightarrow{a} k$, $t[:j+1]$ matches $p[:k]$
 - If $t[j] = p[i]$, $k = i + 1$
 - Else find longest prefix of p that matches suffix of $t[:j+1]$
- Brute force, $O(m^2)$ per edge



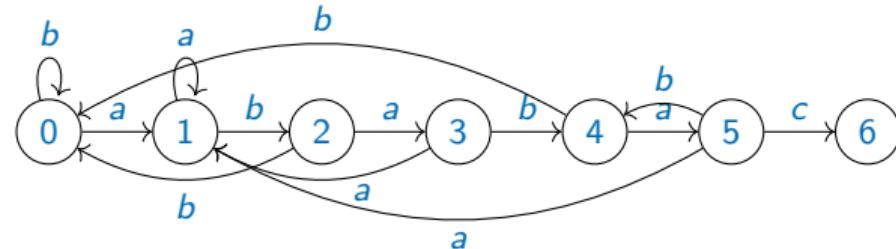
Pattern matching using automata

- The graph we have constructed is a finite state automaton
 - Nodes are states
 - Edges are transitions



Pattern matching using automata

- The graph we have constructed is a finite state automaton
 - Nodes are states
 - Edges are transitions
- Start scanning text in initial state 0

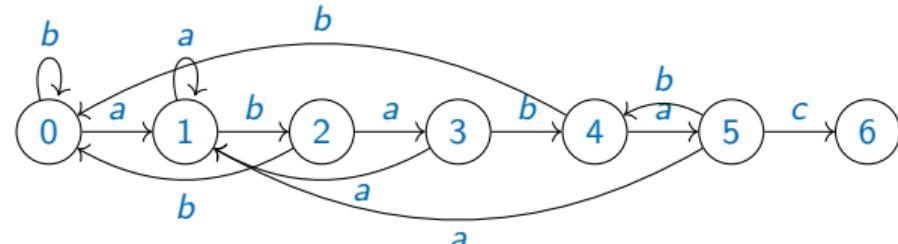


Processing *abababac*

0

Pattern matching using automata

- The graph we have constructed is a finite state automaton
 - Nodes are states
 - Edges are transitions
- Start scanning text in initial state 0
- In state i , read $t[j]$, take the transition labelled $t[j]$
- Updates the longest matching prefix



Processing *abababac*

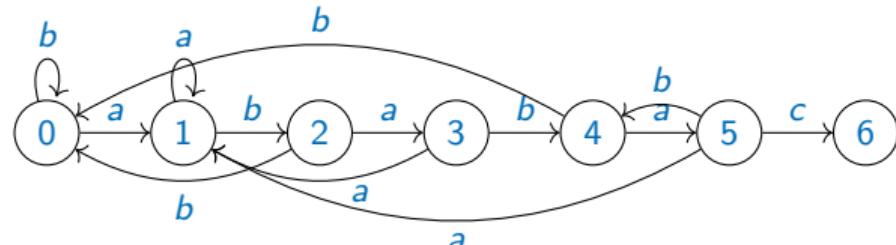
$$0 \xrightarrow{a} 1$$

Pattern matching using automata

- The graph we have constructed is a finite state automaton

- Nodes are states
- Edges are transitions

- Start scanning text in initial state 0
- In state i , read $t[j]$, take the transition labelled $t[j]$
- Updates the longest matching prefix

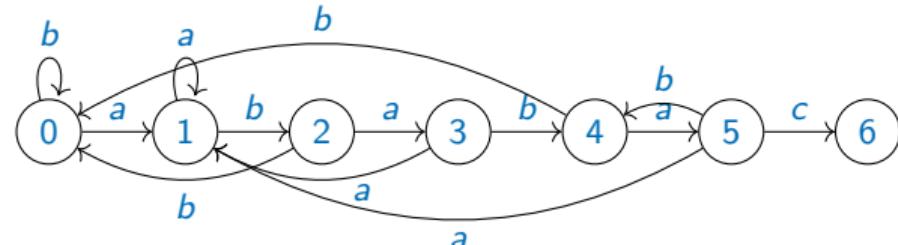


Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2$$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
 - Nodes are states
 - Edges are transitions
- Start scanning text in initial state 0
- In state i , read $t[j]$, take the transition labelled $t[j]$
- Updates the longest matching prefix

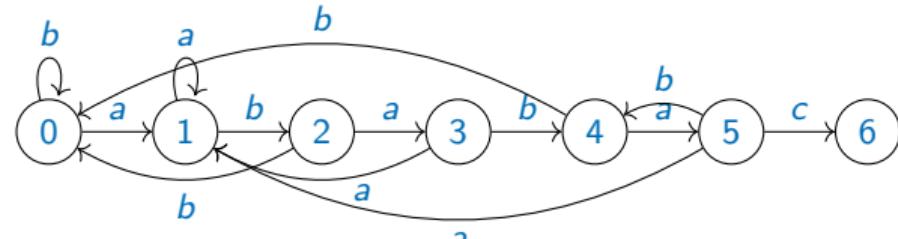


Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3$$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
 - Nodes are states
 - Edges are transitions
- Start scanning text in initial state 0
- In state i , read $t[j]$, take the transition labelled $t[j]$
- Updates the longest matching prefix

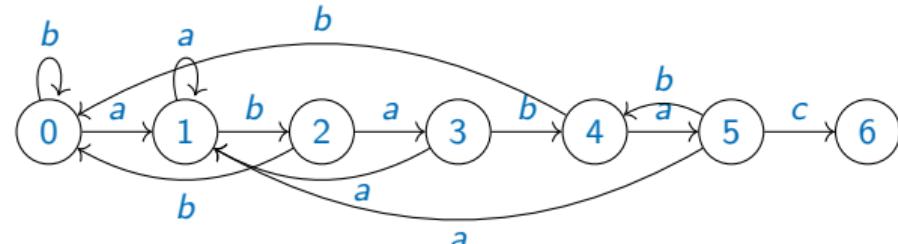


Processing *abababac*

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
 - Nodes are states
 - Edges are transitions
- Start scanning text in initial state 0
- In state i , read $t[j]$, take the transition labelled $t[j]$
- Updates the longest matching prefix

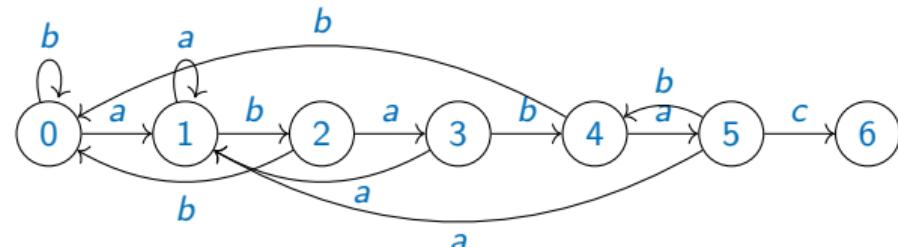


Processing *abababac*

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
 - Nodes are states
 - Edges are transitions
- Start scanning text in initial state 0
- In state i , read $t[j]$, take the transition labelled $t[j]$
- Updates the longest matching prefix

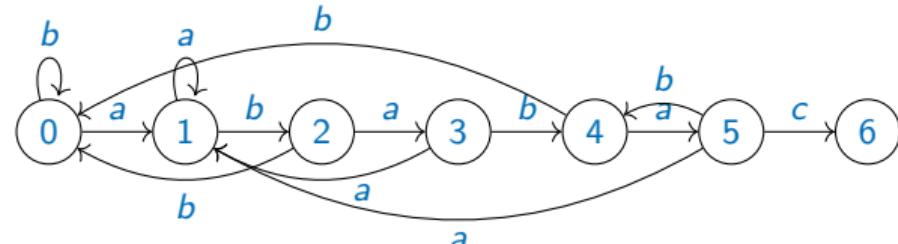


Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4$$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
 - Nodes are states
 - Edges are transitions
- Start scanning text in initial state 0
- In state i , read $t[j]$, take the transition labelled $t[j]$
- Updates the longest matching prefix

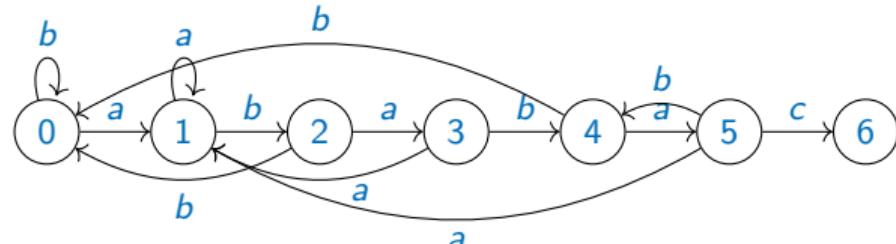


Processing *abababac*

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4 \xrightarrow{a} 5$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
 - Nodes are states
 - Edges are transitions
- Start scanning text in initial state 0
- In state i , read $t[j]$, take the transition labelled $t[j]$
- Updates the longest matching prefix
- If we reach the final state m , we have found a full match for p

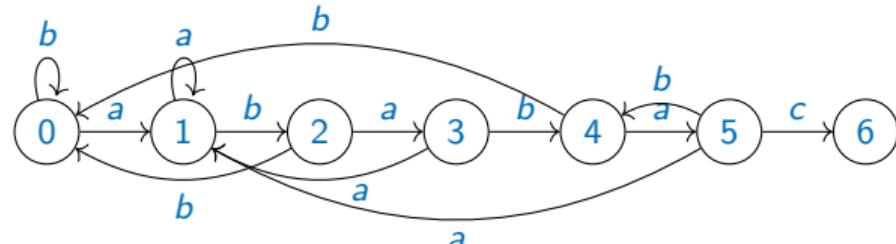


Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{c} 6$$

Pattern matching using automata

- The graph we have constructed is a finite state automaton
 - Nodes are states
 - Edges are transitions
- Start scanning text in initial state 0
- In state i , read $t[j]$, take the transition labelled $t[j]$
- Updates the longest matching prefix
- If we reach the final state m , we have found a full match for p
- Single scan of t suffices



Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{c} 6$$

Summary

- Build an automaton to keep track of longest matching prefix
- Using this automaton, we can do string matching in $O(n)$
 - The algorithm we described finds only the first match
 - Restart at the next position to find subsequent matches

Summary

- Build an automaton to keep track of longest matching prefix
- Using this automaton, we can do string matching in $O(n)$
 - The algorithm we described finds only the first match
 - Restart at the next position to find subsequent matches
- Bottleneck is precomputing the automaton
 - Computing each edge $i \xrightarrow{a} j$ took $O(m^2)$
 - Do this for each $i \in \{0, 1, \dots, m\}$ and each $a \in \Sigma$
 - Overall $O(m^3 \cdot |\Sigma|)$

Summary

- Build an automaton to keep track of longest matching prefix
- Using this automaton, we can do string matching in $O(n)$
 - The algorithm we described finds only the first match
 - Restart at the next position to find subsequent matches
- Bottleneck is precomputing the automaton
 - Computing each edge $i \xrightarrow{a} j$ took $O(m^2)$
 - Do this for each $i \in \{0, 1, \dots, m\}$ and each $a \in \Sigma$
 - Overall $O(m^3 \cdot |\Sigma|)$
- Do this in time $O(m)$ — [Knuth, Morris, Pratt]

String Matching: Knuth-Morris-Pratt algorithm

Madhavan Mukund

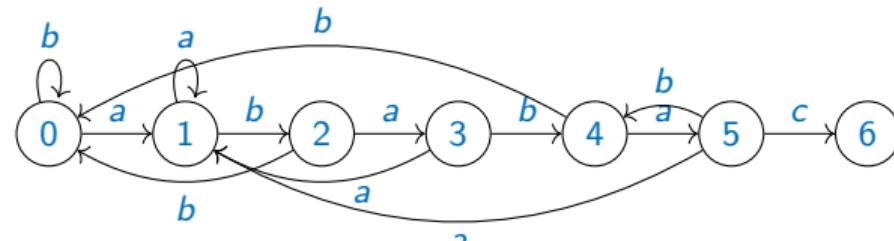
<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 10

Pattern matching using automata

- Finite state automaton for pattern p
 - States $\{0, 1, \dots, m\}$
 - State i denotes match upto $p[:i]$
 - Transition $i \xrightarrow{a} j$ describes how to update the match on reading a
- Start scanning text in **initial state 0**
- In state i , read $t[j]$, take the transition labelled $t[j]$
- If we reach the **final state m** , we have found a full match for p
- Single scan of t suffices



Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{c} 6$$

Knuth-Morris-Pratt algorithm

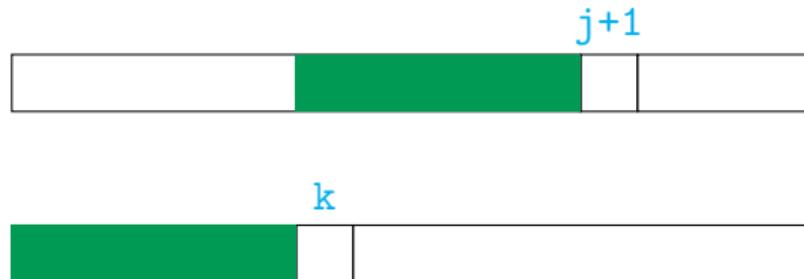
- Compute the automaton for p efficiently

Knuth-Morris-Pratt algorithm

- Compute the automaton for p efficiently
- Match p against itself
 - $\text{match}[j] = k$ if suffix of $p[:j+1]$ matches prefix $p[:k]$

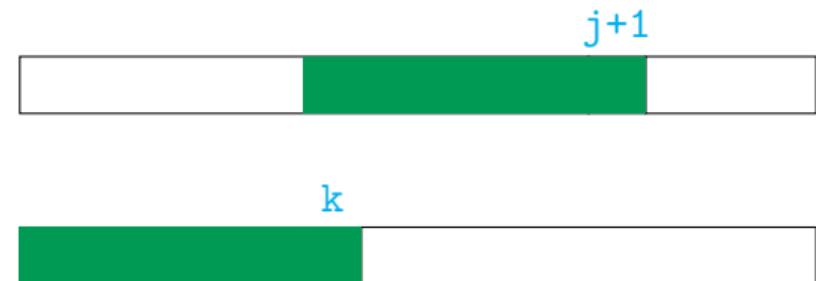
Knuth-Morris-Pratt algorithm

- Compute the automaton for p efficiently
- Match p against itself
 - $\text{match}[j] = k$ if suffix of $p[:j+1]$ matches prefix $p[:k]$
- Suppose suffix of $p[:j+1]$ matches prefix $p[:k]$



Knuth-Morris-Pratt algorithm

- Compute the automaton for p efficiently
- Match p against itself
 - $\text{match}[j] = k$ if suffix of $p[:j+1]$ matches prefix $p[:k]$
- Suppose suffix of $p[:j+1]$ matches prefix $p[:k]$
 - If $p[j+1] == p[k]$, extend the match



Knuth-Morris-Pratt algorithm

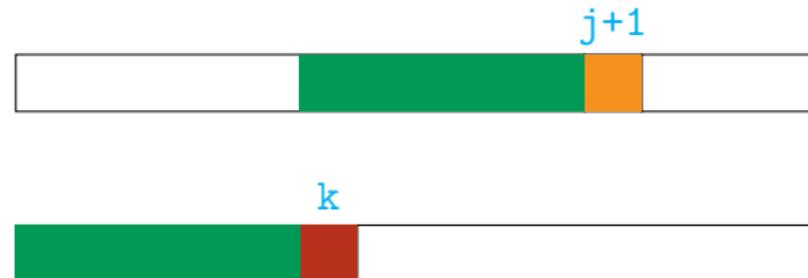
- Compute the automaton for p efficiently

- Match p against itself

- $\text{match}[j] = k$ if suffix of $p[:j+1]$ matches prefix $p[:k]$

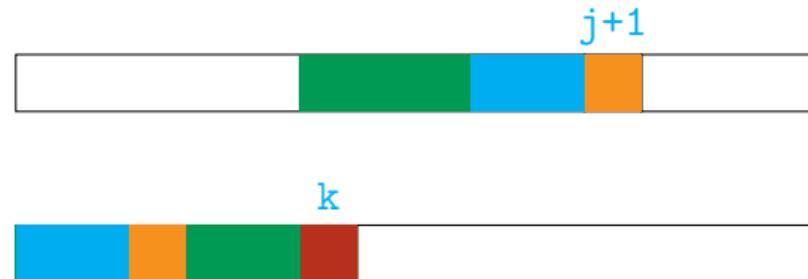
- Suppose suffix of $p[:j+1]$ matches prefix $p[:k]$

- If $p[j+1] == p[k]$, extend the match
 - Otherwise try to find a shorter prefix that can be extended by $p[j+1]$



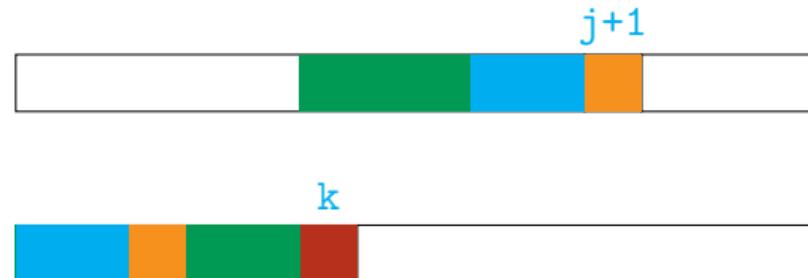
Knuth-Morris-Pratt algorithm

- Compute the automaton for p efficiently
- Match p against itself
 - $\text{match}[j] = k$ if suffix of $p[:j+1]$ matches prefix $p[:k]$
- Suppose suffix of $p[:j+1]$ matches prefix $p[:k]$
 - If $p[j+1] == p[k]$, extend the match
 - Otherwise try to find a shorter prefix that can be extended by $p[j+1]$



Knuth-Morris-Pratt algorithm

- Compute the automaton for p efficiently
- Match p against itself
 - $\text{match}[j] = k$ if suffix of $p[:j+1]$ matches prefix $p[:k]$
- Suppose suffix of $p[:j+1]$ matches prefix $p[:k]$
 - If $p[j+1] == p[k]$, extend the match
 - Otherwise try to find a shorter prefix that can be extended by $p[j+1]$
- Usually refer to `match` as failure function `fail`
 - Where to fall back if match fails



Computing the fail function

- Initialize `fail[j] = 0` for all `j`

```
def kmp_fail(p):  
    # Initialize  
    m = len(p)  
    fail = [0 for i in range(m)]  
  
    # Update  
    j,k = 1,0  
    while j < m:  
        if p[j] == p[k]: #k+1 chars match  
            fail[j] = k+1  
            j,k = j+1,k+1  
        elif k > 0:      #find shorter prefix  
            k = fail[k-1]  
        else:             #no match found at j  
            j = j+1  
    return(fail)
```

Computing the fail function

- Initialize `fail[j] = 0` for all `j`
- `k` keeps track of length of current match
- `j` is next position to update `fail`

```
def kmp_fail(p):  
    # Initialize  
    m = len(p)  
    fail = [0 for i in range(m)]  
  
    # Update  
    j,k = 1,0  
    while j < m:  
        if p[j] == p[k]: #k+1 chars match  
            fail[j] = k+1  
            j,k = j+1,k+1  
        elif k > 0:      #find shorter prefix  
            k = fail[k-1]  
        else:             #no match found at j  
            j = j+1  
    return(fail)
```

Computing the fail function

- Initialize `fail[j] = 0` for all `j`
- `k` keeps track of length of current match
- `j` is next position to update `fail`
- If `p[j] == p[k]` extend the match, set `fail[j] = k+1`

```
def kmp_fail(p):  
    # Initialize  
    m = len(p)  
    fail = [0 for i in range(m)]  
  
    # Update  
    j,k = 1,0  
    while j < m:  
        if p[j] == p[k]: #k+1 chars match  
            fail[j] = k+1  
            j,k = j+1,k+1  
        elif k > 0:      #find shorter prefix  
            k = fail[k-1]  
        else:             #no match found at j  
            j = j+1  
    return(fail)
```

Computing the fail function

- Initialize `fail[j] = 0` for all `j`
- `k` keeps track of length of current match
- `j` is next position to update `fail`
- If `p[j] == p[k]` extend the match, set `fail[j] = k+1`
- If `p[j] != p[k]` find a shorter prefix that matches suffix of `p[:j]`
 - Step back to `fail[k-1]`

```
def kmp_fail(p):  
    # Initialize  
    m = len(p)  
    fail = [0 for i in range(m)]  
  
    # Update  
    j,k = 1,0  
    while j < m:  
        if p[j] == p[k]: #k+1 chars match  
            fail[j] = k+1  
            j,k = j+1,k+1  
        elif k > 0:      #find shorter prefix  
            k = fail[k-1]  
        else:             #no match found at j  
            j = j+1  
    return(fail)
```

Computing the fail function

- Initialize `fail[j] = 0` for all `j`
- `k` keeps track of length of current match
- `j` is next position to update `fail`
- If `p[j] == p[k]` extend the match, set `fail[j] = k+1`
- If `p[j] != p[k]` find a shorter prefix that matches suffix of `p[:j]`
 - Step back to `fail[k-1]`
- If we don't find a nontrivial prefix to extend, retain `fail[j] = 0`, move to next position

```
def kmp_fail(p):  
    # Initialize  
    m = len(p)  
    fail = [0 for i in range(m)]  
  
    # Update  
    j,k = 1,0  
    while j < m:  
        if p[j] == p[k]: #k+1 chars match  
            fail[j] = k+1  
            j,k = j+1,k+1  
        elif k > 0:      #find shorter prefix  
            k = fail[k-1]  
        else:             #no match found at j  
            j = j+1  
    return(fail)
```

Analysis of fail computation

- Want to show this takes time $O(m)$

```
def kmp_fail(p):  
    # Initialize  
    m = len(p)  
    fail = [0 for i in range(m)]  
  
    # Update  
    j,k = 1,0  
    while j < m:  
        if p[j] == p[k]: #k+1 chars match  
            fail[j] = k+1  
            j,k = j+1,k+1  
        elif k > 0:      #find shorter prefix  
            k = fail[k-1]  
        else:             #no match found at j  
            j = j+1  
    return(fail)
```

Analysis of fail computation

- Want to show this takes time $O(m)$
- j incremented $m-1$ times in while

```
def kmp_fail(p):  
    # Initialize  
    m = len(p)  
    fail = [0 for i in range(m)]  
  
    # Update  
    j,k = 1,0  
    while j < m:  
        if p[j] == p[k]: #k+1 chars match  
            fail[j] = k+1  
            j,k = j+1,k+1  
        elif k > 0:      #find shorter prefix  
            k = fail[k-1]  
        else:             #no match found at j  
            j = j+1  
    return(fail)
```

Analysis of fail computation

- Want to show this takes time $O(m)$
- j incremented $m-1$ times in `while`
- But we also have iterations where `k` decreases and `j` is unchanged

```
def kmp_fail(p):  
    # Initialize  
    m = len(p)  
    fail = [0 for i in range(m)]  
  
    # Update  
    j,k = 1,0  
    while j < m:  
        if p[j] == p[k]: #k+1 chars match  
            fail[j] = k+1  
            j,k = j+1,k+1  
        elif k > 0:      #find shorter prefix  
            k = fail[k-1]  
        else:             #no match found at j  
            j = j+1  
    return(fail)
```

Analysis of fail computation

- Want to show this takes time $O(m)$
- j incremented $m-1$ times in `while`
- But we also have iterations where k decreases and j is unchanged
- Total number of decreases to k cannot exceed total number of increments to k

```
def kmp_fail(p):  
    # Initialize  
    m = len(p)  
    fail = [0 for i in range(m)]  
  
    # Update  
    j,k = 1,0  
    while j < m:  
        if p[j] == p[k]: #k+1 chars match  
            fail[j] = k+1  
            j,k = j+1,k+1  
        elif k > 0:      #find shorter prefix  
            k = fail[k-1]  
        else:             #no match found at j  
            j = j+1  
    return(fail)
```

Analysis of fail computation

- Want to show this takes time $O(m)$
- j incremented $m-1$ times in `while`
- But we also have iterations where k decreases and j is unchanged
- Total number of decreases to k cannot exceed total number of increments to k
- Overall k is incremented at most $m-1$ times

```
def kmp_fail(p):  
    # Initialize  
    m = len(p)  
    fail = [0 for i in range(m)]  
  
    # Update  
    j,k = 1,0  
    while j < m:  
        if p[j] == p[k]: #k+1 chars match  
            fail[j] = k+1  
            j,k = j+1,k+1  
        elif k > 0:      #find shorter prefix  
            k = fail[k-1]  
        else:             #no match found at j  
            j = j+1  
    return(fail)
```

Analysis of fail computation

- Want to show this takes time $O(m)$
- j incremented $m-1$ times in `while`
- But we also have iterations where k decreases and j is unchanged
- Total number of decreases to k cannot exceed total number of increments to k
- Overall k is incremented at most $m-1$ times
- Hence overall complexity is $O(m)$

```
def kmp_fail(p):  
    # Initialize  
    m = len(p)  
    fail = [0 for i in range(m)]  
  
    # Update  
    j,k = 1,0  
    while j < m:  
        if p[j] == p[k]: #k+1 chars match  
            fail[j] = k+1  
            j,k = j+1,k+1  
        elif k > 0:      #find shorter prefix  
            k = fail[k-1]  
        else:             #no match found at j  
            j = j+1  
    return(fail)
```

Implementing string search using fail function

- Scan t from beginning

```
def find_kmp(t, p):  
    n,m = len(t),len(p)  
    if m == 0:  
        return 0 # pattern is empty  
    fail = kmp_fail(p) # preprocessing  
    j = 0 # index into text  
    k = 0 # index into pattern  
    while j < n:  
        if t[j] == p[k]: # matched p[0:k+1]  
            if k == m - 1: # match is complete  
                return(j - m + 1)  
            j,k = j+1,k+1 # extend match  
        elif k > 0:  
            k = fail[k-1] # use smaller prefix  
        else:  
            j = j+1  
    return(-1) # reached end without match
```

Implementing string search using fail function

- Scan t from beginning
- j is next position in t
- k is currently matched position in p

```
def find_kmp(t, p):  
    n,m = len(t),len(p)  
    if m == 0:  
        return 0 # pattern is empty  
    fail = kmp_fail(p) # preprocessing  
    j = 0 # index into text  
    k = 0 # index into pattern  
    while j < n:  
        if t[j] == p[k]: # matched p[0:k+1]  
            if k == m - 1: # match is complete  
                return(j - m + 1)  
            j,k = j+1,k+1 # extend match  
        elif k > 0:  
            k = fail[k-1] # use smaller prefix  
        else:  
            j = j+1  
    return(-1) # reached end without match
```

Implementing string search using fail function

- Scan t from beginning
- j is next position in t
- k is currently matched position in p
- If $t[j] == p[k]$ extend the match

```
def find_kmp(t, p):  
    n,m = len(t),len(p)  
    if m == 0:  
        return 0 # pattern is empty  
    fail = kmp_fail(p) # preprocessing  
    j = 0 # index into text  
    k = 0 # index into pattern  
    while j < n:  
        if t[j] == p[k]: # matched p[0:k+1]  
            if k == m - 1: # match is complete  
                return(j - m + 1)  
            j,k = j+1,k+1 # extend match  
        elif k > 0:  
            k = fail[k-1] # use smaller prefix  
        else:  
            j = j+1  
    return(-1) # reached end without match
```

Implementing string search using fail function

- Scan t from beginning
- j is next position in t
- k is currently matched position in p
- If $t[j] == p[k]$ extend the match
- If $t[j] != p[k]$, update match prefix

```
def find_kmp(t, p):  
    n,m = len(t),len(p)  
    if m == 0:  
        return 0 # pattern is empty  
    fail = kmp_fail(p) # preprocessing  
    j = 0 # index into text  
    k = 0 # index into pattern  
    while j < n:  
        if t[j] == p[k]: # matched p[0:k+1]  
            if k == m - 1: # match is complete  
                return(j - m + 1)  
            j,k = j+1,k+1 # extend match  
        elif k > 0:  
            k = fail[k-1] # use smaller prefix  
        else:  
            j = j+1  
    return(-1) # reached end without match
```

Implementing string search using fail function

- Scan t from beginning
- j is next position in t
- k is currently matched position in p
- If $t[j] == p[k]$ extend the match
- If $t[j] != p[k]$, update match prefix
- If we reach the end of the `while` loop, no match

```
def find_kmp(t, p):  
    n,m = len(t),len(p)  
    if m == 0:  
        return 0 # pattern is empty  
    fail = kmp_fail(p) # preprocessing  
    j = 0 # index into text  
    k = 0 # index into pattern  
    while j < n:  
        if t[j] == p[k]: # matched p[0:k+1]  
            if k == m - 1: # match is complete  
                return(j - m + 1)  
            j,k = j+1,k+1 # extend match  
        elif k > 0:  
            k = fail[k-1] # use smaller prefix  
        else:  
            j = j+1  
    return(-1) # reached end without match
```

Implementing string search using fail function

- Scan t from beginning
- j is next position in t
- k is currently matched position in p
- If $t[j] == p[k]$ extend the match
- If $t[j] != p[k]$, update match prefix
- If we reach the end of the `while` loop, no match
- Complexity is $O(n)$

```
def find_kmp(t, p):  
    n,m = len(t),len(p)  
    if m == 0:  
        return 0 # pattern is empty  
    fail = kmp_fail(p) # preprocessing  
    j = 0 # index into text  
    k = 0 # index into pattern  
    while j < n:  
        if t[j] == p[k]: # matched p[0:k+1]  
            if k == m - 1: # match is complete  
                return(j - m + 1)  
            j,k = j+1,k+1 # extend match  
        elif k > 0:  
            k = fail[k-1] # use smaller prefix  
        else:  
            j = j+1  
    return(-1) # reached end without match
```

Implementing string search using fail function

- Scan t from beginning
- j is next position in t
- k is currently matched position in p
- If $t[j] == p[k]$ extend the match
- If $t[j] != p[k]$, update match prefix
- If we reach the end of the `while` loop, no match
- Complexity is $O(n)$
- This finds first match, modify to find all matches

```
def find_kmp(t, p):  
    n,m = len(t),len(p)  
    if m == 0:  
        return 0 # pattern is empty  
    fail = kmp_fail(p) # preprocessing  
    j = 0 # index into text  
    k = 0 # index into pattern  
    while j < n:  
        if t[j] == p[k]: # matched p[0:k+1]  
            if k == m - 1: # match is complete  
                return(j - m + 1)  
            j,k = j+1,k+1 # extend match  
        elif k > 0:  
            k = fail[k-1] # use smaller prefix  
        else:  
            j = j+1  
    return(-1) # reached end without match
```

Summary

- The Knuth, Morris, Pratt algorithm efficiently computes the automaton describing prefix matches in the pattern p
- Complexity of preprocessing the `fail` function is $O(m)$
- After preprocessing, can check matches in the text t in $O(n)$
- Overall, KMP algorithm works in time $O(m + n)$
- However, the Boyer-Moore algorithm can be faster in practice, skipping many positions

String Matching: Tries

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 10

Searching a fixed text

- String matching often involves searching a large fixed body of text
 - Collected works of Shakespeare
 - Comprehensive set of reference manuals
 - Genetic data

Searching a fixed text

- String matching often involves searching a large fixed body of text
 - Collected works of Shakespeare
 - Comprehensive set of reference manuals
 - Genetic data
- Make multiple queries on this text
 - Find the source of a famous quotation
 - Search for information on a part or a procedure
 - Search for a given gene sequence

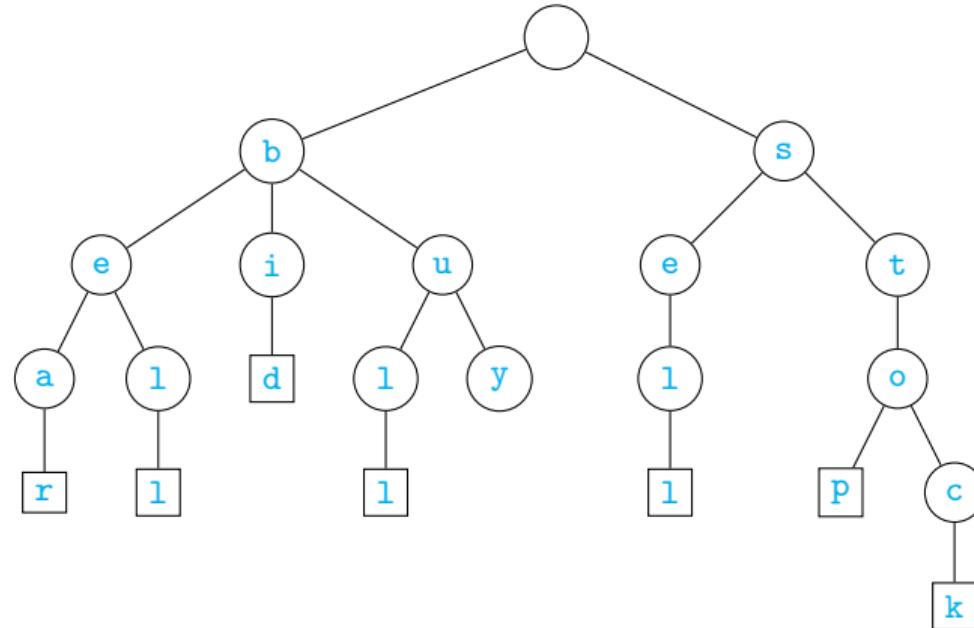
Searching a fixed text

- String matching often involves searching a large fixed body of text
 - Collected works of Shakespeare
 - Comprehensive set of reference manuals
 - Genetic data
- Make multiple queries on this text
 - Find the source of a famous quotation
 - Search for information on a part or a procedure
 - Search for a given gene sequence
- Preprocess the text to make the search efficient
 - Locate information about a pattern p of length m in time $O(m)$

Tries

- A **trie** is a special kind of tree
 - From “information retrieval”
 - Pronounced **try**, distinguish from tree

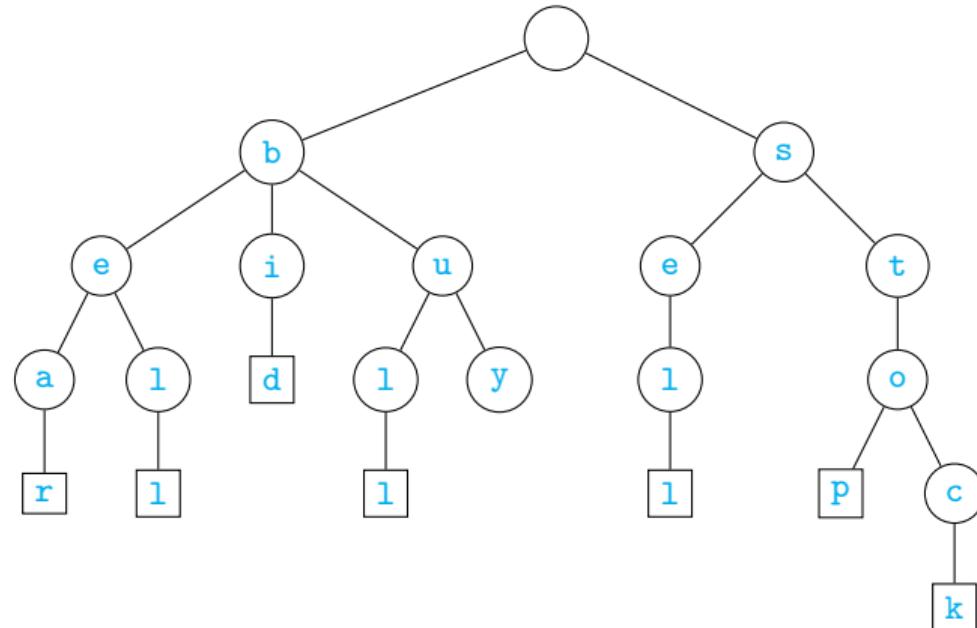
{bear,bell,bid,bull,buy,sell,stop,stock}



Tries

- A **trie** is a special kind of tree
 - From “information retrieval”
 - Pronounced **try**, distinguish from tree
- Rooted tree
 - Other than root, each node labelled by a letter from Σ
 - Children of a node have distinct labels

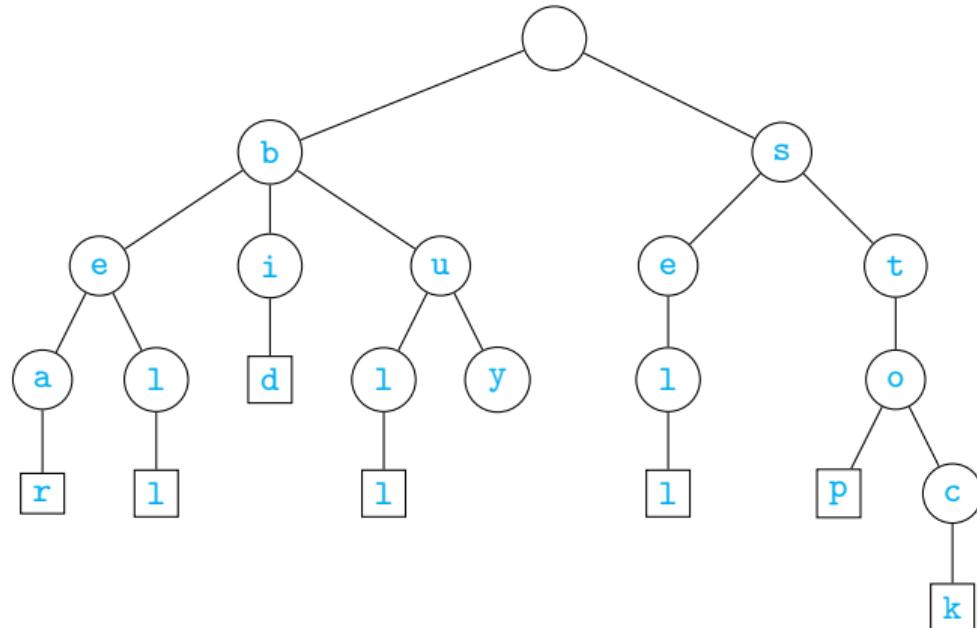
{bear,bell,bid,bull,buy,sell,stop,stock}



Tries

- A **trie** is a special kind of tree
 - From “information retrieval”
 - Pronounced **try**, distinguish from tree
- Rooted tree
 - Other than root, each node labelled by a letter from Σ
 - Children of a node have distinct labels
- Each maximal path is a word
 - One word should not be a prefix of another
 - Add special end of word symbol $\$$

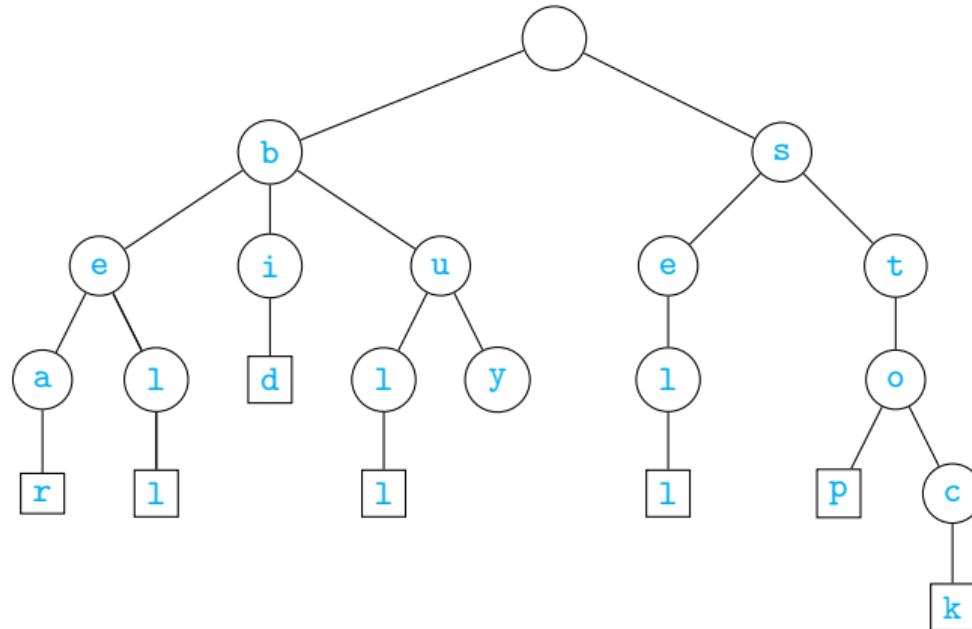
{bear,bell,bid,bull,buy,sell,stop,stock}



Tries

- Build a trie T from a set of words S with s words and n total symbols

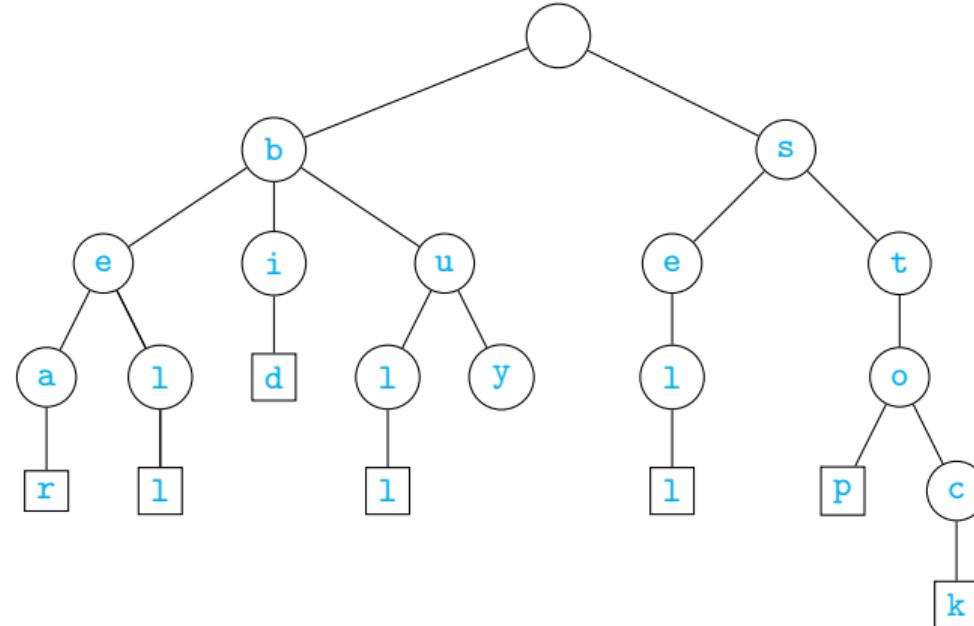
{bear,bell,bid,bull,buy,sell,stop,stock}



Tries

- Build a trie T from a set of words S with s words and n total symbols
- To search for a word w , follow its path

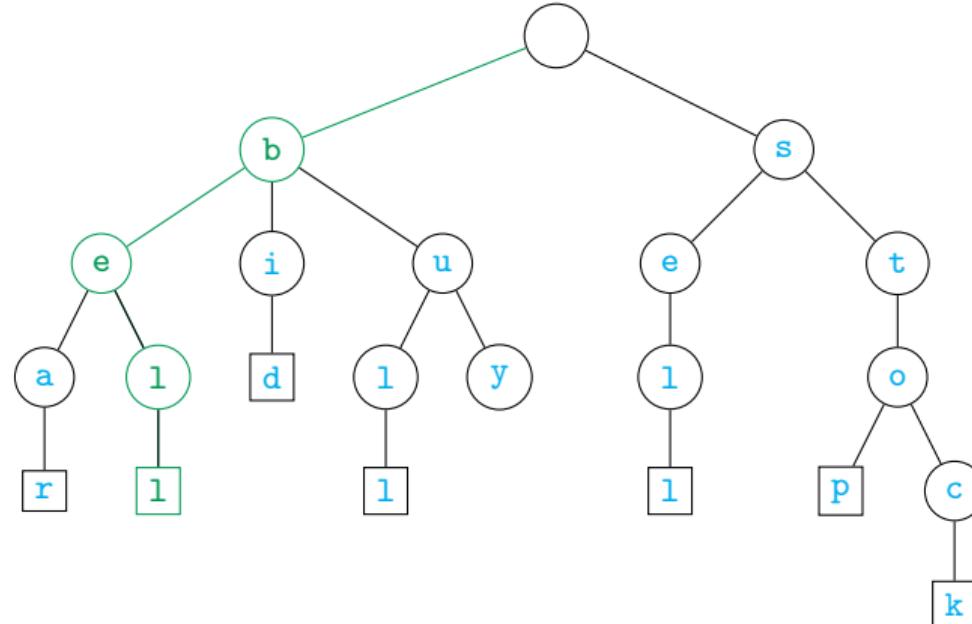
Search for `bell`



Tries

- Build a trie T from a set of words S with s words and n total symbols
- To search for a word w , follow its path
 - If the node we reach has $\$$ as a successor, $w \in S$

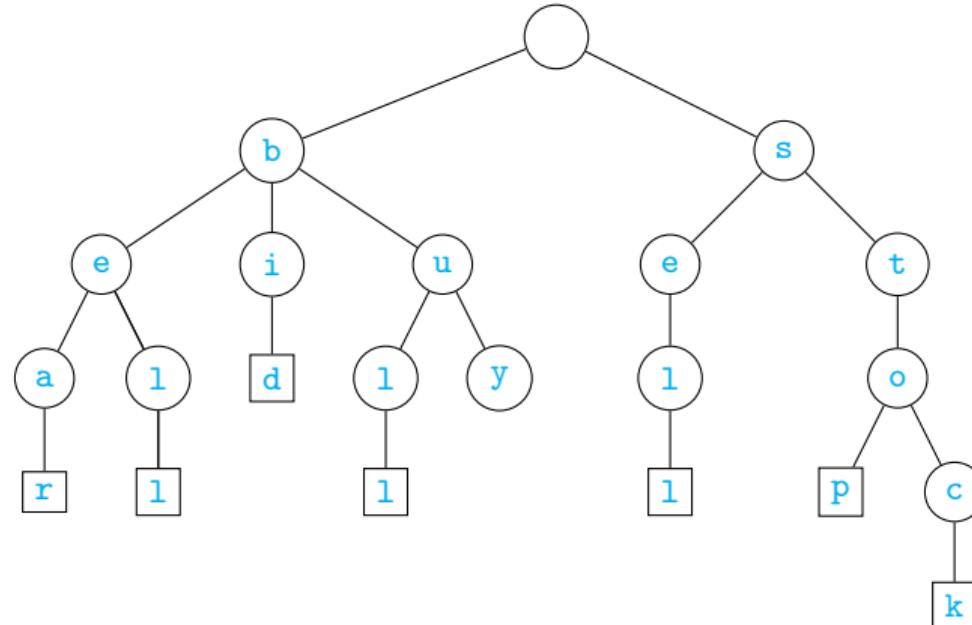
Search for `bell`



Tries

- Build a trie T from a set of words S with s words and n total symbols
- To search for a word w , follow its path
 - If the node we reach has $\$$ as a successor, $w \in S$
 - $w \notin S$ — path cannot be completed, or w is a prefix of some $w' \in S$

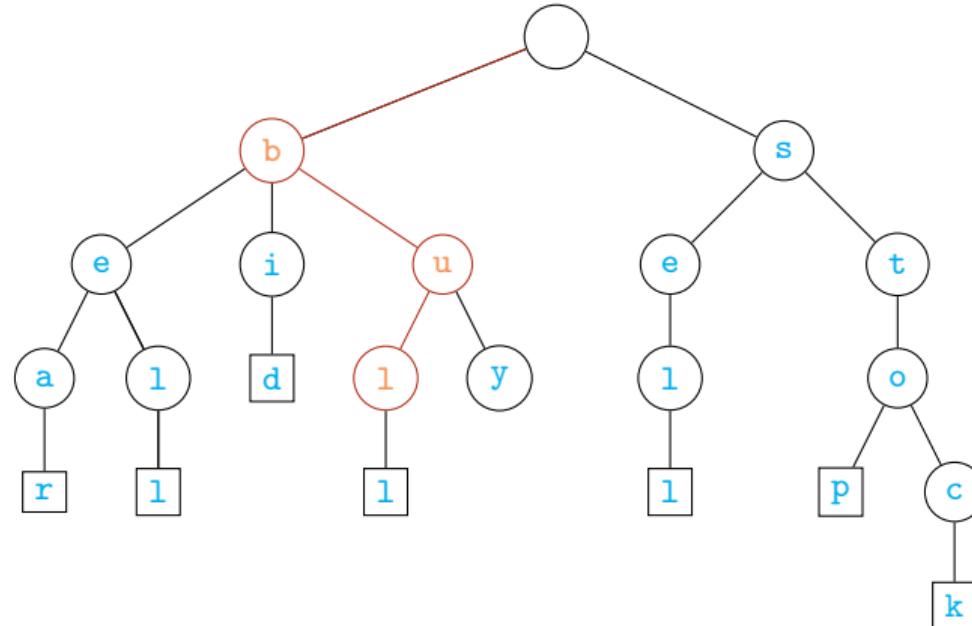
Search for `bulk`



Tries

- Build a trie T from a set of words S with s words and n total symbols
- To search for a word w , follow its path
 - If the node we reach has $\$$ as a successor, $w \in S$
 - $w \notin S$ — path cannot be completed, or w is a prefix of some $w' \in S$

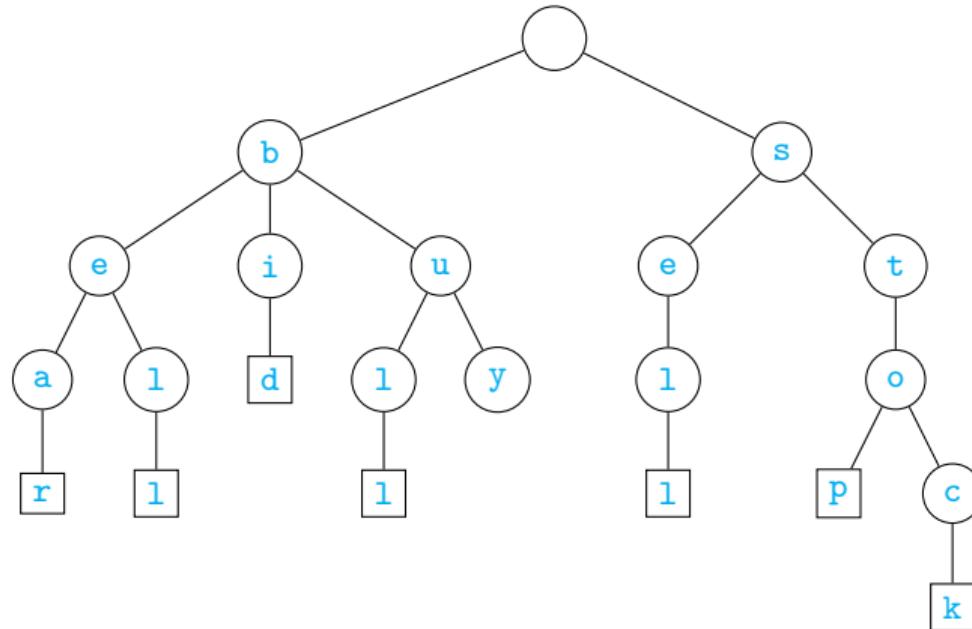
Search for `bulk`



Tries

- Build a trie T from a set of words S with s words and n total symbols

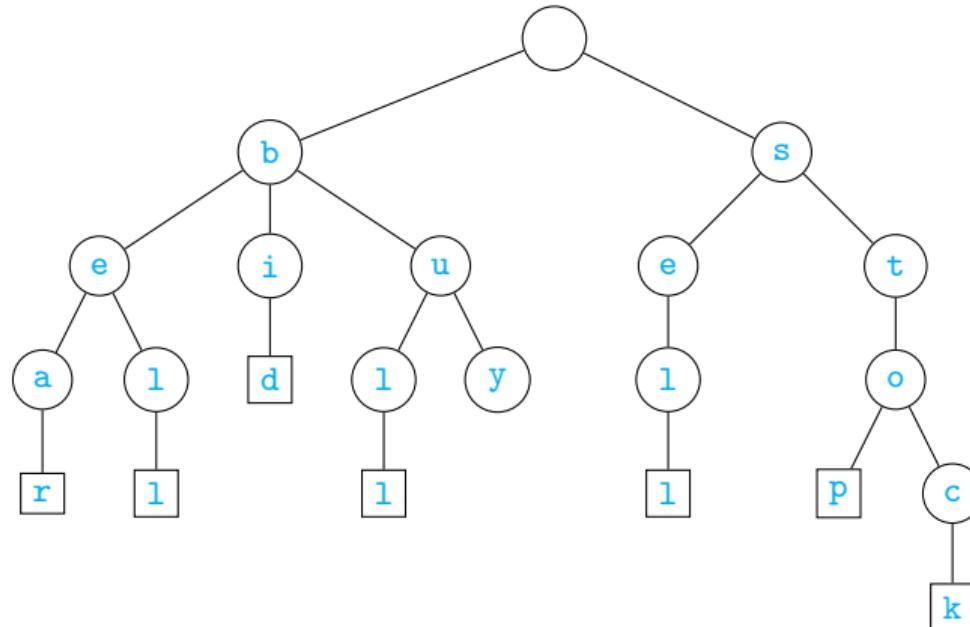
{bear,bell,bid,bull,buy,sell,stop,stock}



Tries

- Build a trie T from a set of words S with s words and n total symbols
- Basic properties for T built from S
 - Height of T is $\max_{w \in S} \text{len}(w)$
 - A node has at most $|\Sigma|$ children
 - The number of leaves in T is s
 - The number of nodes in T is $n + 1$, plus s nodes labelled $\$$

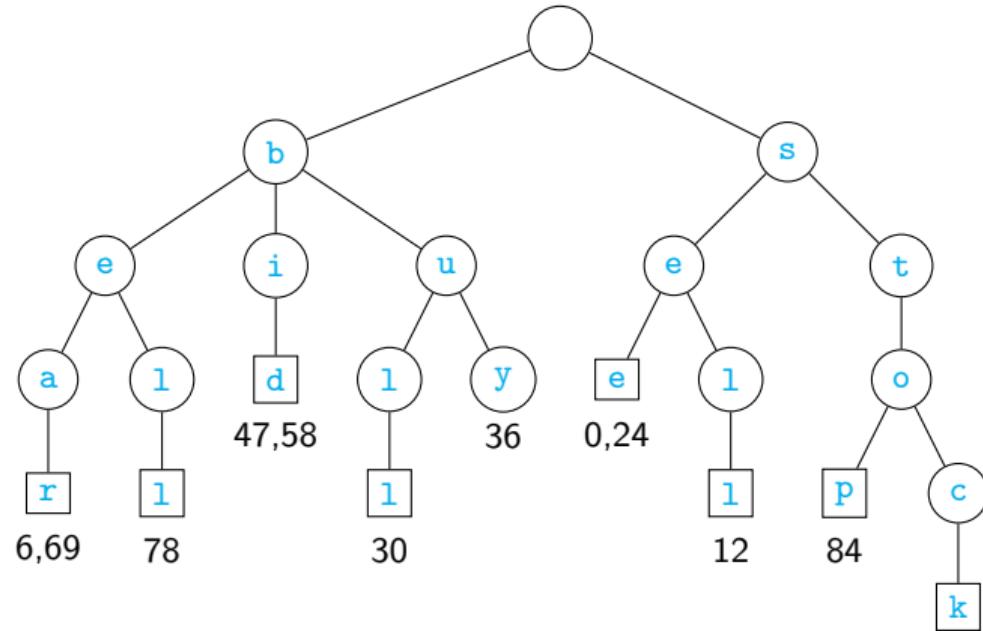
{bear,bell,bid,bull,buy,sell,stop,stock}



Auxiliary information

- Can maintain auxiliary information for each word
 - e.g., list of positions where the word occurs

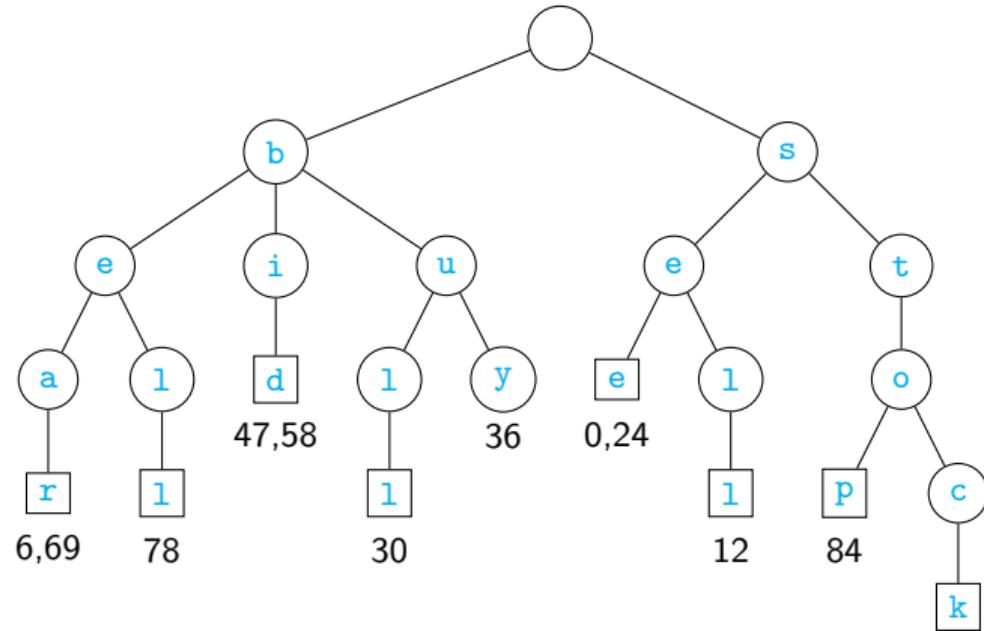
"see a bear? sell stock! see a bull? buy stock!
bid stock! bid stock! bear the bell? stop!"



Auxiliary information

- Can maintain auxiliary information for each word
 - e.g., list of positions where the word occurs
- Trie as a key-value map
 - Keys are words in S
 - Values are relevant information about the word

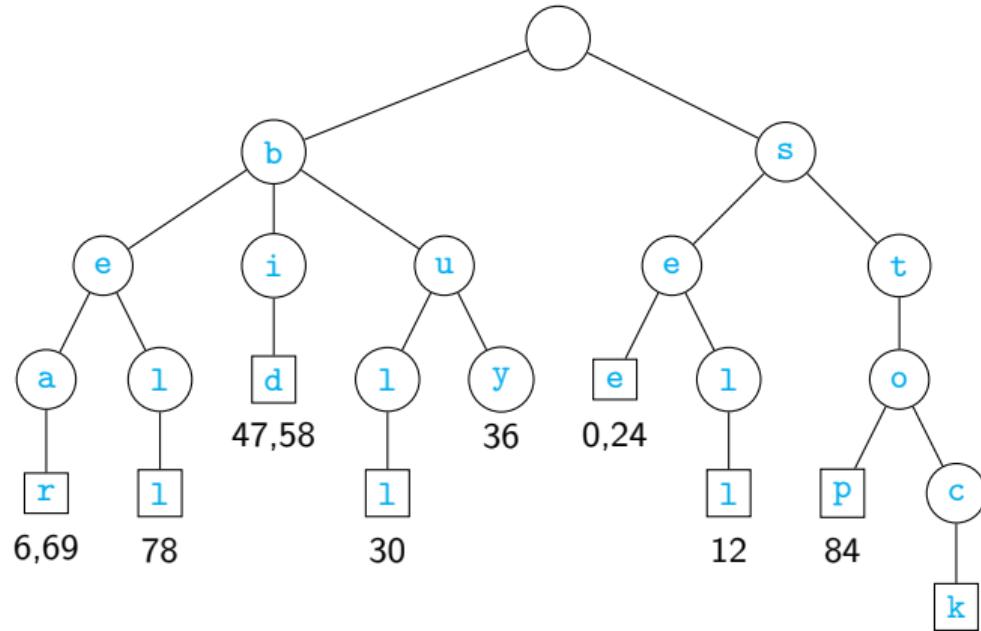
"see a bear? sell stock! see a bull? buy stock!
bid stock! bid stock! bear the bell? stop!"



Auxiliary information

- Can maintain auxiliary information for each word
 - e.g., list of positions where the word occurs
- Trie as a key-value map
 - Keys are words in *S*
 - Values are relevant information about the word
- Trie vs hash functions
 - Time to look up is proportional to length of key
 - No collisions in tries
 - Tries take up more space

"see a bear? sell stock! see a bull? buy stock!
bid stock! bid stock! bear the bell? stop!"



Trie: Implementation

- A Python `class` implementing tries

```
class Trie:  
  
    def __init__(self,S=[]):  
        self.root = {}  
        for s in S:  
            self.add(s)  
  
    def add(self,s):  
        curr = self.root  
        s = s + "$"  
        for c in s:  
            if c not in curr.keys():  
                curr[c] = {}  
            curr = curr[c]
```

Trie: Implementation

- A Python `class` implementing tries
- `add` inserts a new word into the trie

```
class Trie:  
  
    def __init__(self,S=[]):  
        self.root = {}  
        for s in S:  
            self.add(s)  
  
    def add(self,s):  
        curr = self.root  
        s = s + "$"  
        for c in s:  
            if c not in curr.keys():  
                curr[c] = {}  
            curr = curr[c]
```

Trie: Implementation

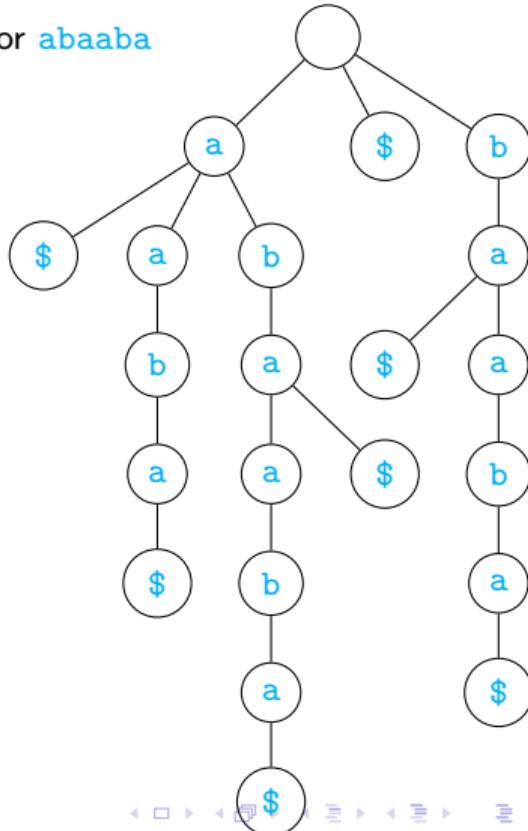
- A Python `class` implementing tries
- `add` inserts a new word into the trie
- `query` checks for a complete word
 - `True` — `s` is a complete word in `T`
 - `False` — `s` is not found in `T`
 - `None` — `s` is a prefix of some word in `T`

```
class Trie:  
  
    def __init__(self,S=[]):  
        self.root = {}  
        for s in S:  
            self.add(s)  
  
    def add(self,s):  
        ...  
  
    def query(self,s):  
        curr = self.root  
        for c in s:  
            if c not in curr.keys():  
                return(False)  
            curr = curr[c]  
        if "$" in curr.keys():  
            return(True)  
        else:  
            return(None)
```

Suffix tries

- Expand S to include all suffixes
 - For simplicity, assume $S = \{s\}$
 - $\text{suffix}(S) = \{w \mid \exists v, vw = s\}$

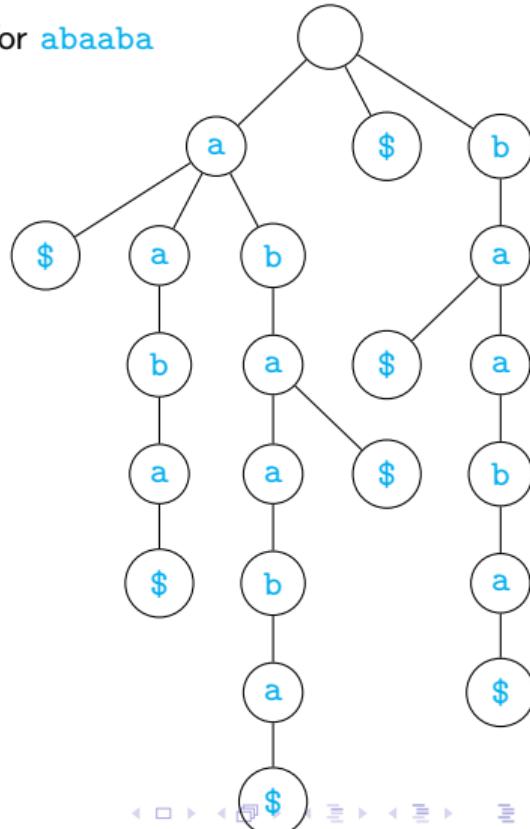
Suffix trie for abaaba



Suffix tries

- Expand S to include all suffixes
 - For simplicity, assume $S = \{s\}$
 - $\text{suffix}(S) = \{w \mid \exists v, vw = s\}$
- Build a trie for $\text{suffix}(S)$
 - Use $\$$ to mark end of word
 - Suffix trie for S

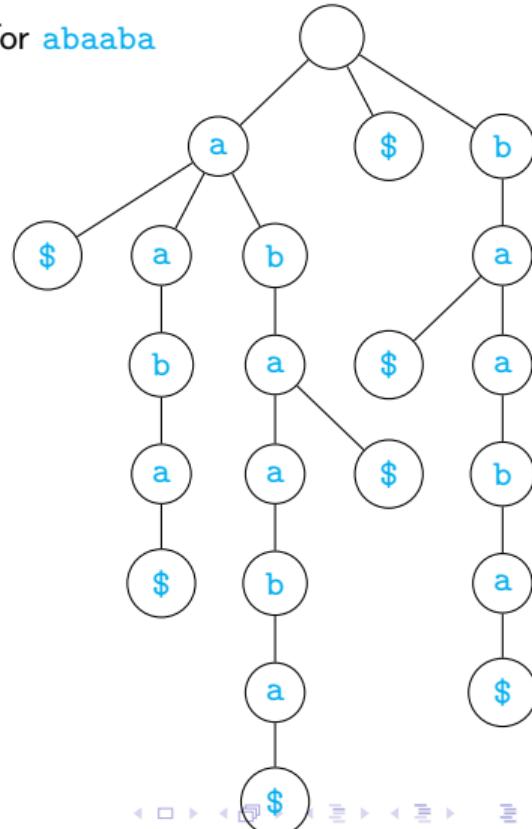
Suffix trie for abaaba



Suffix tries

- Expand S to include all suffixes
 - For simplicity, assume $S = \{s\}$
 - $\text{suffix}(S) = \{w \mid \exists v, vw = s\}$
- Build a trie for $\text{suffix}(S)$
 - Use $\$$ to mark end of word
 - Suffix trie for S
- Using a suffix trie we can answer the following
 - Is w a substring of s
 - How many times does w occur as a substring in s
 - What is the longest repeated substring in s

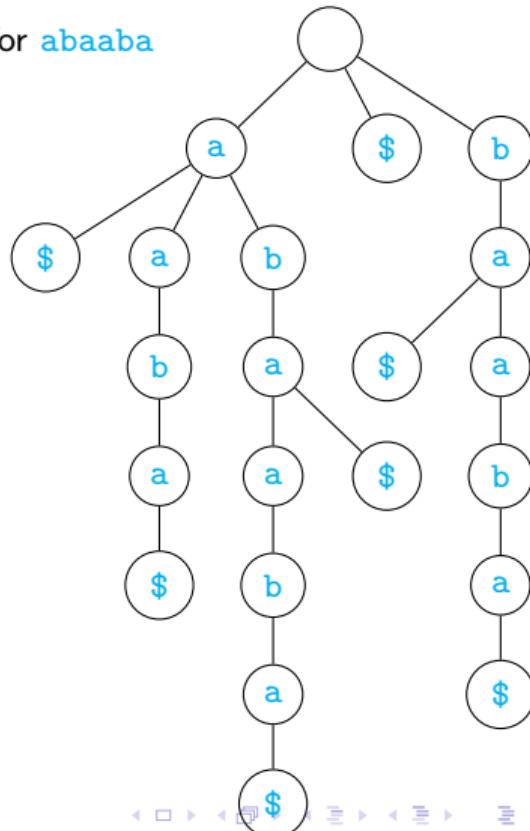
Suffix trie for abaaba



Using suffix tries

- Is w a substring of s ?
 - abaaba — yes, baabb — no

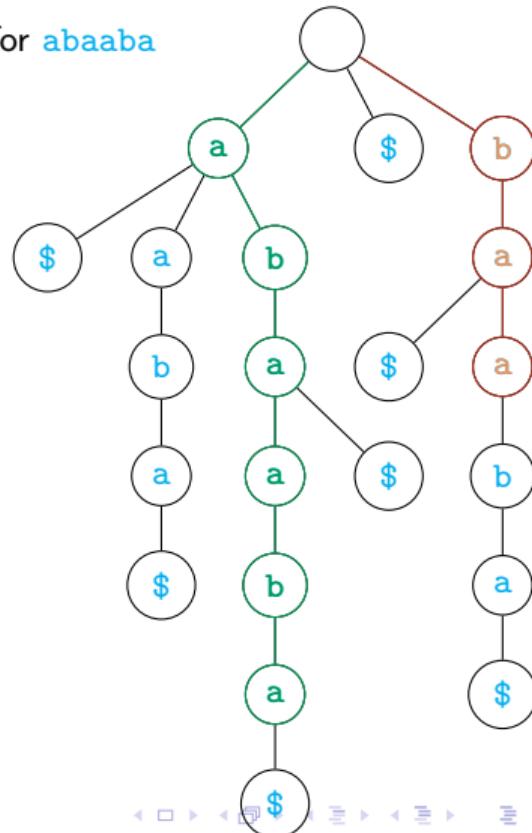
Suffix trie for abaaba



Using suffix tries

- Is *w* a substring of *s*?
 - abaaba — yes, baabb — no

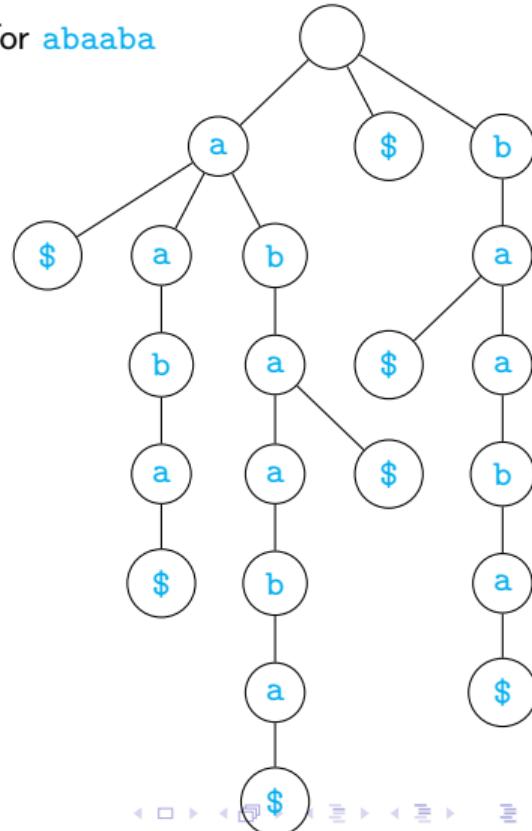
Suffix trie for abaaba



Using suffix tries

- Is w a substring of s ?
 - abaaba — yes, baabb — no
- Is w a suffix of s ?
 - baa — no, aba — yes

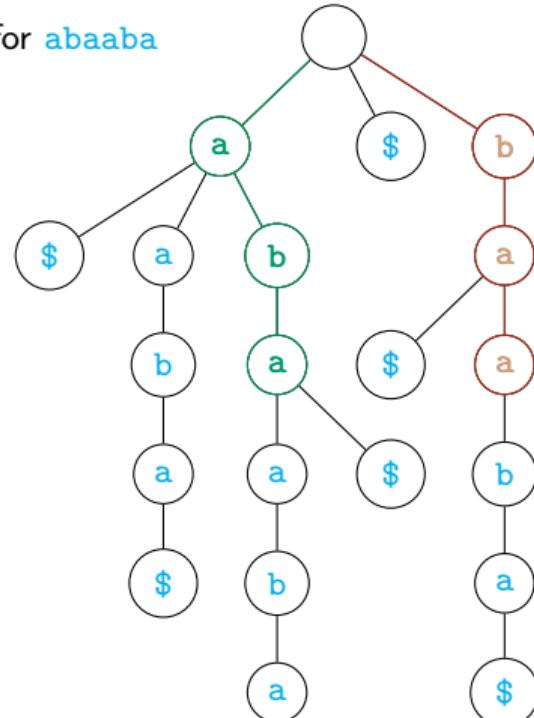
Suffix trie for abaaba



Using suffix tries

- Is w a substring of s ?
 - abaaba — yes, baabb — no
- Is w a suffix of s ?
 - baa — no, aba — yes

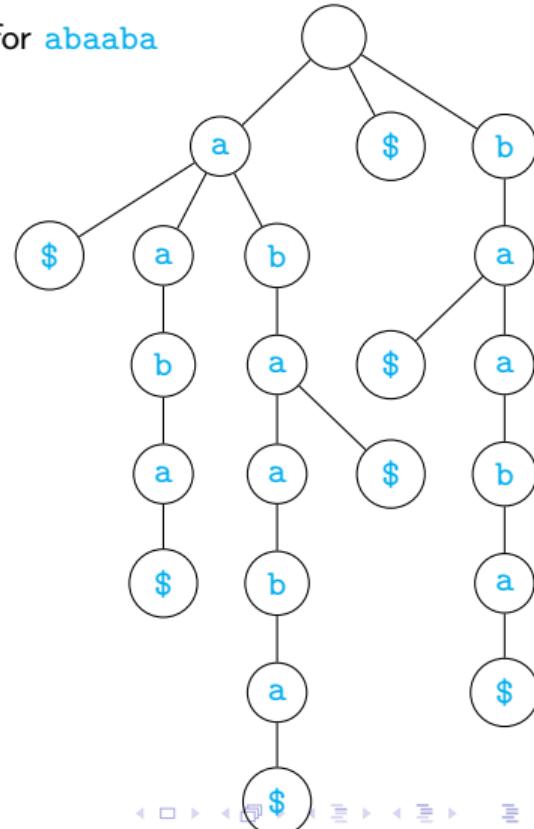
Suffix trie for abaaba



Using suffix tries

- Is w a substring of s ?
 - abaaba — yes, baabb — no
- Is w a suffix of s ?
 - baa — no, aba — yes
- Number of times w occurs as a substring of s
 - aba — 2 occurrences
 - Number of leaves below the node

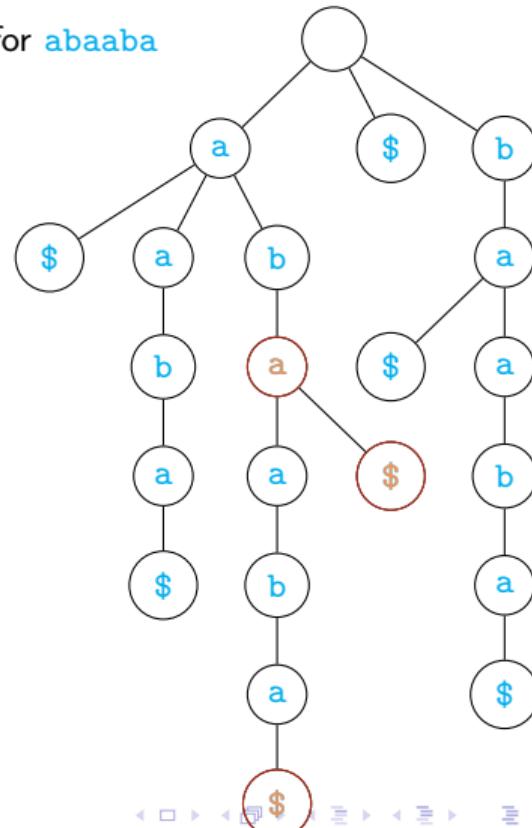
Suffix trie for abaaba



Using suffix tries

- Is w a substring of s ?
 - abaaba — yes, baabb — no
 - Is w a suffix of s ?
 - baa — no, aba — yes
 - Number of times w occurs as a substring of s
 - aba — 2 occurrences
 - Number of leaves below the node

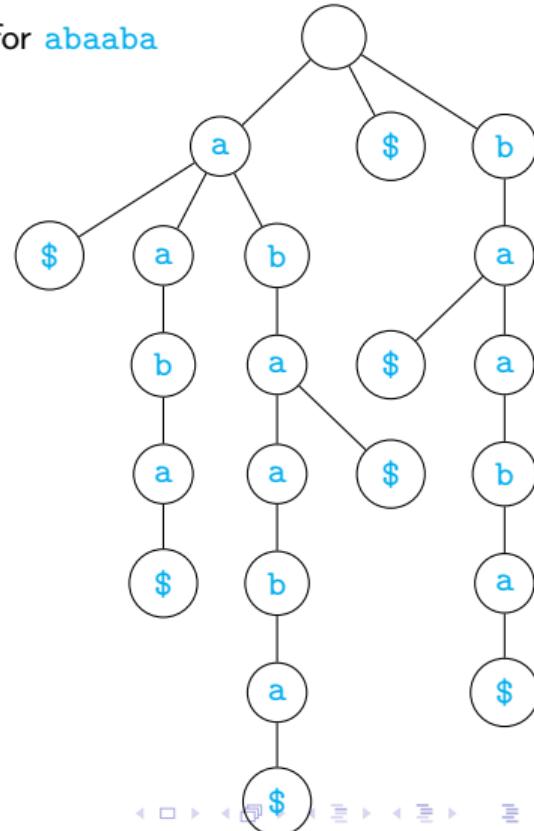
Suffix trie for abaaba



Using suffix tries

- Is w a substring of s ?
 - abaaba — yes, baabb — no
- Is w a suffix of s ?
 - baa — no, aba — yes
- Number of times w occurs as a substring of s
 - aba — 2 occurrences
 - Number of leaves below the node
- Longest repeated substring of s
 - aba — 2 occurrences
 - Deepest node with more than one child

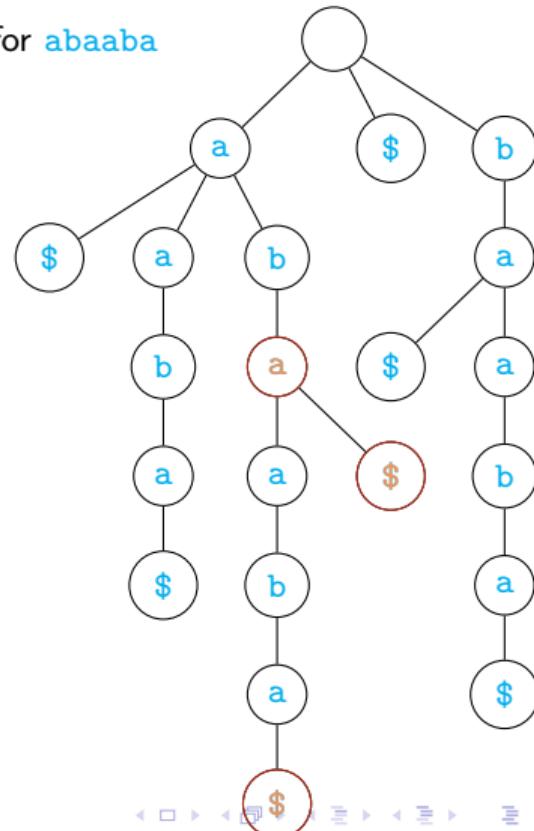
Suffix trie for abaaba



Using suffix tries

- Is w a substring of s ?
 - $abaaba$ — yes, $baabb$ — no
 - Is w a suffix of s ?
 - baa — no, aba — yes
 - Number of times w occurs as a substring of s
 - aba — 2 occurrences
 - Number of leaves below the node
 - Longest repeated substring of s
 - aba — 2 occurrences
 - Deepest node with more than one child

Suffix trie for abaaba



Suffix trie: Implementation

- Constructor builds a trie with every suffix of `s`

```
class SuffixTrie:  
  
    def __init__(self,s):  
        self.root = {}  
        s = s + "$"  
        for i in range(len(s)):  
            curr = self.root  
            for c in s[i:]:  
                if c not in curr.keys():  
                    curr[c] = {}  
                curr = curr[c]  
  
    def followPath(self,s):  
        curr = self.root  
        for c in s:  
            if c not in curr.keys():  
                return(None)  
            curr = curr[c]  
        return(curr)
```

Suffix trie: Implementation

- Constructor builds a trie with every suffix of `s`
- `followPath` follows the path dictated by `s`
 - Return `None` if path fails
 - Return last node in the path if it succeeds

```
class SuffixTrie:  
  
    def __init__(self,s):  
        self.root = {}  
        s = s + "$"  
        for i in range(len(s)):  
            curr = self.root  
            for c in s[i:]:  
                if c not in curr.keys():  
                    curr[c] = {}  
                curr = curr[c]  
  
    def followPath(self,s):  
        curr = self.root  
        for c in s:  
            if c not in curr.keys():  
                return(None)  
            curr = curr[c]  
        return(curr)
```

Suffix trie: Implementation

- Constructor builds a trie with every suffix of s
- `followPath` follows the path dictated by s
 - Return `None` if path fails
 - Return last node in the path if it succeeds
- If `followPath` finds a path, s is a valid substring

```
class SuffixTrie:  
  
    def __init__(self,s):  
        self.root = {}  
        s = s + "$"  
        for i in range(len(s)):  
            curr = self.root  
            for c in s[i:]:  
                if c not in curr.keys():  
                    curr[c] = {}  
                curr = curr[c]  
  
    def followPath(self,s):  
        ...  
  
    def hasSubstring(self,s):  
        return(self.followPath(s) is not None)
```

Suffix trie: Implementation

- Constructor builds a trie with every suffix of `s`
- `followPath` follows the path dictated by `s`
 - Return `None` if path fails
 - Return last node in the path if it succeeds
- If `followPath` finds a path, `s` is a valid substring
- If `followPath` ends in `$`, `s` is a suffix

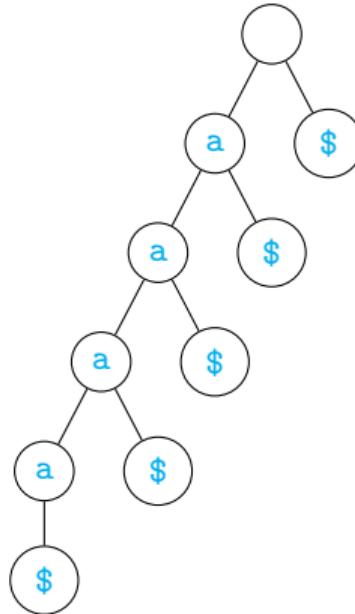
```
class SuffixTrie:  
  
    def __init__(self,s):  
        self.root = {}  
        s = s + "$"  
        for i in range(len(s)):  
            curr = self.root  
            for c in s[i:]:  
                if c not in curr.keys():  
                    curr[c] = {}  
                curr = curr[c]  
  
    def followPath(self,s):  
        ...  
  
    def hasSuffix(self,s):  
        node = self.followPath(s)  
        return(node is not None and  
              "$" in node.keys())
```

Suffix trie: size

- How big can a suffix trie be for s of length n ?

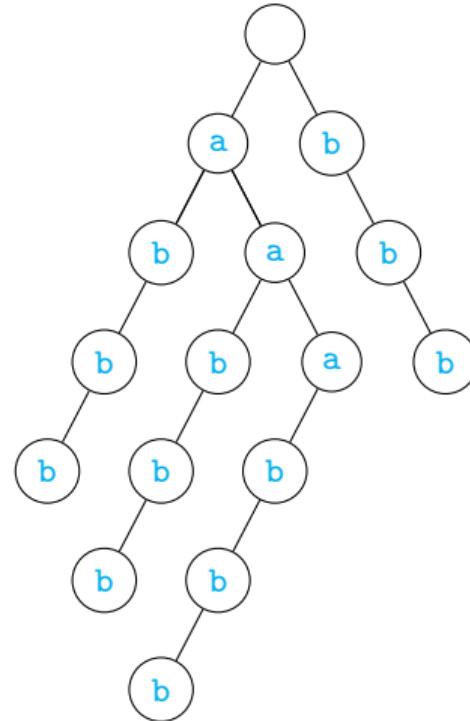
Suffix trie: size

- How big can a suffix trie be for s of length n ?
- Number of nodes proportional to n ?
 - Yes, a^n



Suffix trie: size

- How big can a suffix trie be for s of length n ?
- Number of nodes proportional to n ?
 - Yes, a^n
- Number of nodes proportional to n^2 ?
 - Yes, $a^n b^n$
 - \$ nodes not shown



Summary

- Tries are useful to preprocess fixed text for multiple searches
- Searching for p is proportional to length of p
- Suffix tries allow us to make more expressive searches
- Main drawback of a trie is size

String Matching: Regular Expressions

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 10

Searching for patterns

- So far, we have done string matching for a fixed pattern p in a string t

Searching for patterns

- So far, we have done string matching for a fixed pattern p in a string t
- What is we want to look for a pattern?

Searching for patterns

- So far, we have done string matching for a fixed pattern `p` in a string `t`
- What is we want to look for a pattern?
- Unsure of spelling — look for `Srivatsan` or `Srivathsan`

Searching for patterns

- So far, we have done string matching for a fixed pattern `p` in a string `t`
- What is we want to look for a pattern?
- Unsure of spelling — look for `Srivatsan` or `Srivathsan`
- Check for a sequence — look for a word that has `sub` followed by `tion`
 - `substitution`, `subtraction`

Searching for patterns

- So far, we have done string matching for a fixed pattern `p` in a string `t`
- What is we want to look for a pattern?
- Unsure of spelling — look for `Srivatsan` or `Srivathsan`
- Check for a sequence — look for a word that has `sub` followed by `tion`
 - `substitution`, `subtraction`
- Repetition — one or more copy of `na`
 - `pennant`, `banana`

Describing patterns

- Finite alphabet $\Sigma = \{a, b, c, \dots\}$

Describing patterns

- Finite alphabet $\Sigma = \{a, b, c, \dots\}$
- A pattern matches a set of words over Σ

Describing patterns

- Finite alphabet $\Sigma = \{a, b, c, \dots\}$
- A pattern matches a set of words over Σ
- Building patterns

Describing patterns

- Finite alphabet $\Sigma = \{a, b, c, \dots\}$
- A pattern matches a set of words over Σ
- Building patterns
- Each $a \in \Sigma$ is a pattern — matches $\{a\}$

Describing patterns

- Finite alphabet $\Sigma = \{a, b, c, \dots\}$
- A pattern matches a set of words over Σ
- Building patterns
- Each $a \in \Sigma$ is a pattern — matches $\{a\}$
- If p is a pattern matching S_p and q is a pattern matching S_q , then $p + q$ matches $S_p \cup S_q$

Describing patterns

- Finite alphabet $\Sigma = \{a, b, c, \dots\}$
- A pattern matches a set of words over Σ
- Building patterns
- Each $a \in \Sigma$ is a pattern — matches $\{a\}$
- If p is a pattern matching S_p and q is a pattern matching S_q , then $p + q$ matches $S_p \cup S_q$
- If p is a pattern matching S_p and q is a pattern matching S_q , then pq matches $S_p \cdot S_q = \{uv \mid u \in S_p, v \in S_q\}$

Describing patterns

- Finite alphabet $\Sigma = \{a, b, c, \dots\}$
- A pattern matches a set of words over Σ
- Building patterns
- Each $a \in \Sigma$ is a pattern — matches $\{a\}$
- If p is a pattern matching S_p and q is a pattern matching S_q , then $p + q$ matches $S_p \cup S_q$
- If p is a pattern matching S_p and q is a pattern matching S_q , then pq matches $S_p \cdot S_q = \{uv \mid u \in S_p, v \in S_q\}$
- If p is a pattern matching S_p , then p^+ matches any word w that can be decomposed as $w_1 w_2 \dots w_k$ where each $w_j \in S_p$
 - p^+ is 1 or more repetitions of p , p^* is 0 or more repetitions

Examples

- Pattern $a + b$ matches $\{a, b\}$
 - a matches $\{a\}$, b matches $\{b\}$, take the union

Examples

- Pattern $a + b$ matches $\{a, b\}$
 - a matches $\{a\}$, b matches $\{b\}$, take the union
- Likewise, pattern $c + d$ matches $\{c, d\}$

Examples

- Pattern $a + b$ matches $\{a, b\}$
 - a matches $\{a\}$, b matches $\{b\}$, take the union
- Likewise, pattern $c + d$ matches $\{c, d\}$
- Pattern $(a + b)(c + d)$ matches $\{ac, bc, ad, bd\}$
 - $a + b$ matches $\{a, b\}$, $c + d$ matches $\{c, d\}$, construct all words with first part from $\{a, b\}$ followed by second part from $\{c, d\}$

Examples

- Pattern $a + b$ matches $\{a, b\}$
 - a matches $\{a\}$, b matches $\{b\}$, take the union
- Likewise, pattern $c + d$ matches $\{c, d\}$
- Pattern $(a + b)(c + d)$ matches $\{ac, bc, ad, bd\}$
 - $a + b$ matches $\{a, b\}$, $c + d$ matches $\{c, d\}$, construct all words with first part from $\{a, b\}$ followed by second part from $\{c, d\}$
- Pattern $[(a + b)(c + d)]^+$ matches $\{acac, acbc, acad, acbd, bcac, acacac, \dots\}$
 - Match any word that can be decomposed into words from $\{ac, bc, ad, bd\}$

Shortcuts

- Want to match a followed by b
 - Any symbol from Σ can come in between
 - $\Sigma = \{a_1, a_2, \dots, a_k\}$
 - $a_1 + a_2 + \dots + a_k$ matches any symbol in Σ
 - Use Σ itself as an abbreviation for this special pattern
 - The pattern we want is $a\Sigma^+b$
 - If we allow no gap between a and b , $a\Sigma^*b$

Shortcuts

- Want to match a followed by b
 - Any symbol from Σ can come in between
 - $\Sigma = \{a_1, a_2, \dots, a_k\}$
 - $a_1 + a_2 + \dots + a_k$ matches any symbol in Σ
 - Use Σ itself as an abbreviation for this special pattern
 - The pattern we want is $a\Sigma^+b$
 - If we allow no gap between a and b , $a\Sigma^*b$
- Looking for *Srivatsan* or *Srivathsan*
 - Pattern is $(\Sigma^* Srivatsan \Sigma^*) + (\Sigma^* Srivathsan \Sigma^*)$
 - By convention, can drop initial and final Σ^* — $(Srivatsan + Srivathsan)$ matches anywhere in the text
 - More compactly $Srivat(h + \varepsilon)san$
 - ε matches the empty string, with no characters

Anchoring patterns

- Our convention is now that p matches anywhere in a string

Anchoring patterns

- Our convention is now that p matches anywhere in a string
- To match the start of the string, write p

Anchoring patterns

- Our convention is now that p matches anywhere in a string
- To match the start of the string, write p
- To match the end of the string, write $p$$

Anchoring patterns

- Our convention is now that p matches anywhere in a string
- To match the start of the string, write p
- To match the end of the string, write $p$$
- ba and $na$$ both match *banana*

Anchoring patterns

- Our convention is now that p matches anywhere in a string
- To match the start of the string, write p
- To match the end of the string, write $p$$
- ba and $na$$ both match *banana*
- p$ will match if entire word matches p

Anchoring patterns

- Our convention is now that p matches anywhere in a string
- To match the start of the string, write p
- To match the end of the string, write $p$$
- ba and $na$$ both match *banana*
- p$ will match if entire word matches p
- bana$ does not match *banana*, but $^ba(na)^+$$ does

Regular expressions

- Recall that the Knuth-Morris-Pratt algorithm built a finite state automaton for the longest prefix matched by a pattern

Regular expressions

- Recall that the Knuth-Morris-Pratt algorithm built a finite state automaton for the longest prefix matched by a pattern
- The automaton we built had a linear structure, with a single path from start to finish

Regular expressions

- Recall that the Knuth-Morris-Pratt algorithm built a finite state automaton for the longest prefix matched by a pattern
- The automaton we built had a linear structure, with a single path from start to finish
- In general, automata can follow multiple paths, accept multiple words

Regular expressions

- Recall that the Knuth-Morris-Pratt algorithm built a finite state automaton for the longest prefix matched by a pattern
- The automaton we built had a linear structure, with a single path from start to finish
- In general, automata can follow multiple paths, accept multiple words
- The set of words that automata can accept are called regular sets

Regular expressions

- Recall that the Knuth-Morris-Pratt algorithm built a finite state automaton for the longest prefix matched by a pattern
- The automaton we built had a linear structure, with a single path from start to finish
- In general, automata can follow multiple paths, accept multiple words
- The set of words that automata can accept are called regular sets
- Each pattern p describes a set of words, those that it matches

Regular expressions

- Recall that the Knuth-Morris-Pratt algorithm built a finite state automaton for the longest prefix matched by a pattern
- The automaton we built had a linear structure, with a single path from start to finish
- In general, automata can follow multiple paths, accept multiple words
- The set of words that automata can accept are called regular sets
- Each pattern p describes a set of words, those that it matches
- The sets we can describe using patterns are exactly the same as those that can be accepted by automata

Regular expressions

- Recall that the Knuth-Morris-Pratt algorithm built a finite state automaton for the longest prefix matched by a pattern
- The automaton we built had a linear structure, with a single path from start to finish
- In general, automata can follow multiple paths, accept multiple words
- The set of words that automata can accept are called regular sets
- Each pattern p describes a set of words, those that it matches
- The sets we can describe using patterns are exactly the same as those that can be accepted by automata
- Our patterns are called **regular expressions**

Regular expressions

- For every automaton, we can construct a pattern p that matches exactly the words that the automaton accepts

Regular expressions

- For every automaton, we can construct a pattern p that matches exactly the words that the automaton accepts
- For every pattern p , we can construct an automaton that accepts all words that match p

Regular expressions

- For every automaton, we can construct a pattern p that matches exactly the words that the automaton accepts
- For every pattern p , we can construct an automaton that accepts all words that match p
- We can extend string matching to pattern matching by building an automaton for a pattern p and processing the text through this automaton, like Knuth-Morris-Pratt for fixed strings

Regular expressions

- For every automaton, we can construct a pattern p that matches exactly the words that the automaton accepts
- For every pattern p , we can construct an automaton that accepts all words that match p
- We can extend string matching to pattern matching by building an automaton for a pattern p and processing the text through this automaton, like Knuth-Morris-Pratt for fixed strings
- Python provides a library for matching regular expressions

Linear Programming

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 11

Optimization problems

- Many computational tasks involve optimization
 - Shortest path
 - Minimum cost spanning tree
 - Longest common subsequence

Optimization problems

- Many computational tasks involve optimization
 - Shortest path
 - Minimum cost spanning tree
 - Longest common subsequence
- ...subject to constraints
 - Shortest path follows edges in the graph
 - Spanning tree is a subset of the given edges
 - Subsequence letters are from the given words

Optimization problems

- Many computational tasks involve optimization
 - Shortest path
 - Minimum cost spanning tree
 - Longest common subsequence
- ... subject to constraints
 - Shortest path follows edges in the graph
 - Spanning tree is a subset of the given edges
 - Subsequence letters are from the given words

Linear programming

- Constraints and objective to be optimized are linear functions
 - **Constraints:** $a_1x_1 + a_2x_2 + \dots + a_mx_m \leq K$, $b_1x_1 + b_2x_2 + \dots + b_mx_m \geq L$, ...
 - **Objective:** $c_1x_1 + c_2x_2 + \dots + c_mx_m$

Example: Maximize profits

Grandiose Sweets sells cashew barfis and dry fruit halwa.

Example: Maximize profits

Grandiose Sweets sells cashew barfis and dry fruit halwa.

- Profit for each box of barfis is Rs 100
- Profit for each box of halwa is Rs 600

Example: Maximize profits

Grandiose Sweets sells cashew barfis and dry fruit halwa.

- Profit for each box of barfis is Rs 100
- Profit for each box of halwa is Rs 600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes

Example: Maximize profits

Grandiose Sweets sells cashew barfis and dry fruit halwa.

- Profit for each box of barfis is Rs 100
- Profit for each box of halwa is Rs 600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes
- Staff can produce 400 boxes a day, altogether

Example: Maximize profits

Grandiose Sweets sells cashew barfis and dry fruit halwa.

- Profit for each box of barfis is Rs 100
- Profit for each box of halwa is Rs 600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes
- Staff can produce 400 boxes a day, altogether
- What is the most profitable mix of barfis and halwa to produce?

Example: Maximize profits

Grandiose Sweets sells cashew barfis and dry fruit halwa.

Linear programming model

- Profit for each box of barfis is Rs 100
- Profit for each box of halwa is Rs 600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes
- Staff can produce 400 boxes a day, altogether
- What is the most profitable mix of barfis and halwa to produce?

Example: Maximize profits

Grandiose Sweets sells cashew barfis and dry fruit halwa.

- Profit for each box of barfis is Rs 100
- Profit for each box of halwa is Rs 600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes
- Staff can produce 400 boxes a day, altogether
- What is the most profitable mix of barfis and halwa to produce?

Linear programming model

- b boxes of barfi to produce per day
- h boxes of halwa to produce per day

Example: Maximize profits

Grandiose Sweets sells cashew barfis and dry fruit halwa.

- Profit for each box of barfis is Rs 100
- Profit for each box of halwa is Rs 600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes
- Staff can produce 400 boxes a day, altogether
- What is the most profitable mix of barfis and halwa to produce?

Linear programming model

- b boxes of barfi to produce per day
- h boxes of halwa to produce per day
- Profit: $100b + 600h$

Example: Maximize profits

Grandiose Sweets sells cashew barfis and dry fruit halwa.

- Profit for each box of barfis is Rs 100
- Profit for each box of halwa is Rs 600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes
- Staff can produce 400 boxes a day, altogether
- What is the most profitable mix of barfis and halwa to produce?

Linear programming model

- b boxes of barfi to produce per day
- h boxes of halwa to produce per day
- Profit: $100b + 600h$
- Demand constraints:
 - $b \leq 200$
 - $h \leq 300$

Example: Maximize profits

Grandiose Sweets sells cashew barfis and dry fruit halwa.

- Profit for each box of barfis is Rs 100
- Profit for each box of halwa is Rs 600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes
- Staff can produce 400 boxes a day, altogether
- What is the most profitable mix of barfis and halwa to produce?

Linear programming model

- b boxes of barfi to produce per day
- h boxes of halwa to produce per day
- Profit: $100b + 600h$
- Demand constraints:
 - $b \leq 200$
 - $h \leq 300$
- Production constraint: $b + h \leq 400$

Example: Maximize profits

Grandiose Sweets sells cashew barfis and dry fruit halwa.

- Profit for each box of barfis is Rs 100
- Profit for each box of halwa is Rs 600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes
- Staff can produce 400 boxes a day, altogether
- What is the most profitable mix of barfis and halwa to produce?

Linear programming model

- b boxes of barfi to produce per day
- h boxes of halwa to produce per day
- Profit: $100b + 600h$
- Demand constraints:
 - $b \leq 200$
 - $h \leq 300$
- Production constraint: $b + h \leq 400$
- Implicit constraints:
 - $b \geq 0$
 - $h \geq 0$

Linear program

Objective

- Maximize $100b + 600h$

Linear programming model

- b boxes of barfi to produce per day
- h boxes of halwa to produce per day
- Profit: $100b + 600h$
- Demand constraints:
 - $b \leq 200$
 - $h \leq 300$
- Production constraint: $b + h \leq 400$
- Implicit constraints:
 - $b \geq 0$
 - $h \geq 0$

Linear program

Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Linear programming model

- b boxes of barfi to produce per day
- h boxes of halwa to produce per day
- Profit: $100b + 600h$
- Demand constraints:
 - $b \leq 200$
 - $h \leq 300$
- Production constraint: $b + h \leq 400$
- Implicit constraints:
 - $b \geq 0$
 - $h \geq 0$

Linear program

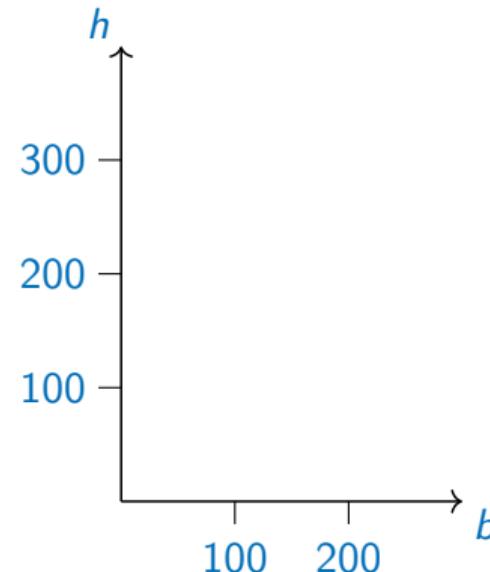
Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Pictorially



Linear program

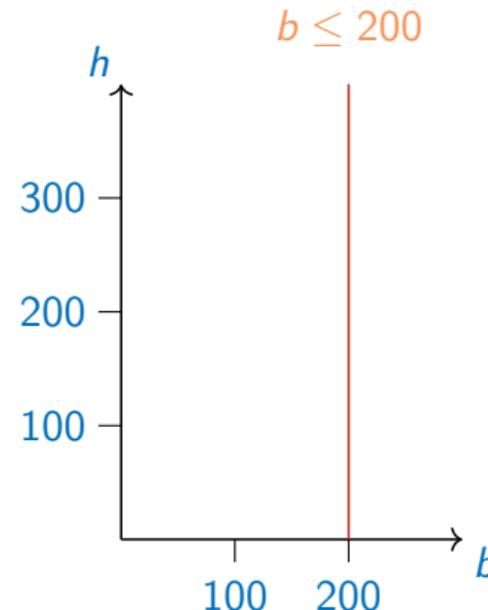
Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Pictorially



Linear program

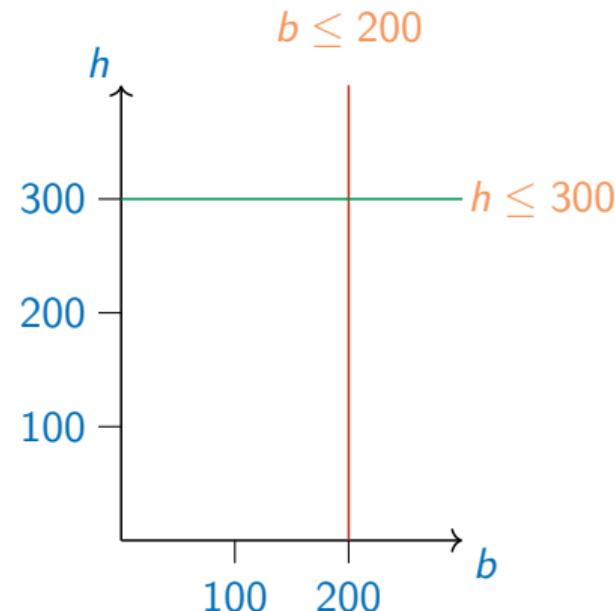
Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Pictorially



Linear program

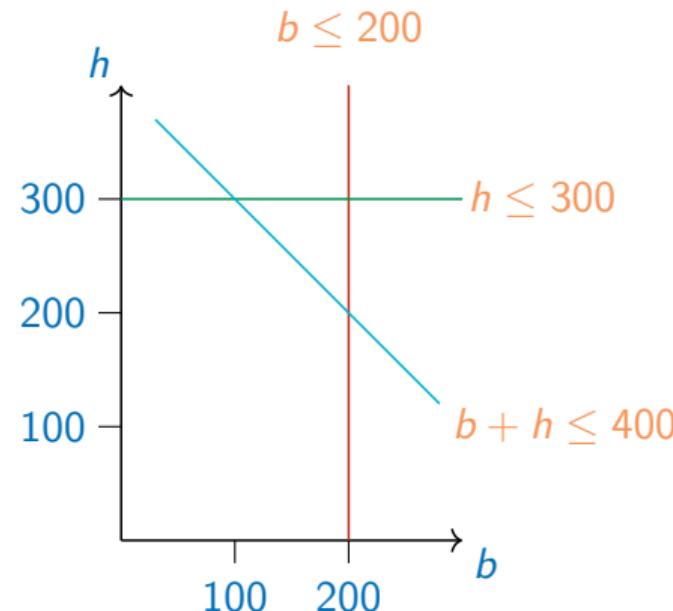
Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Pictorially



Linear program

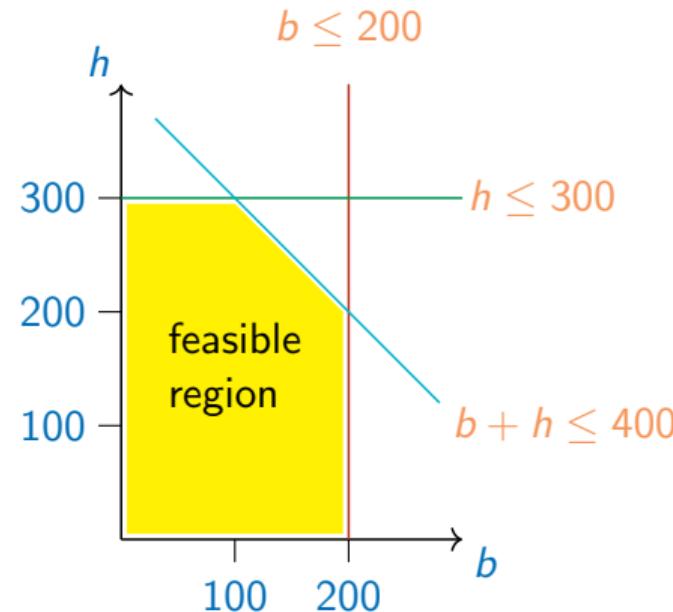
Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Pictorially



Linear program

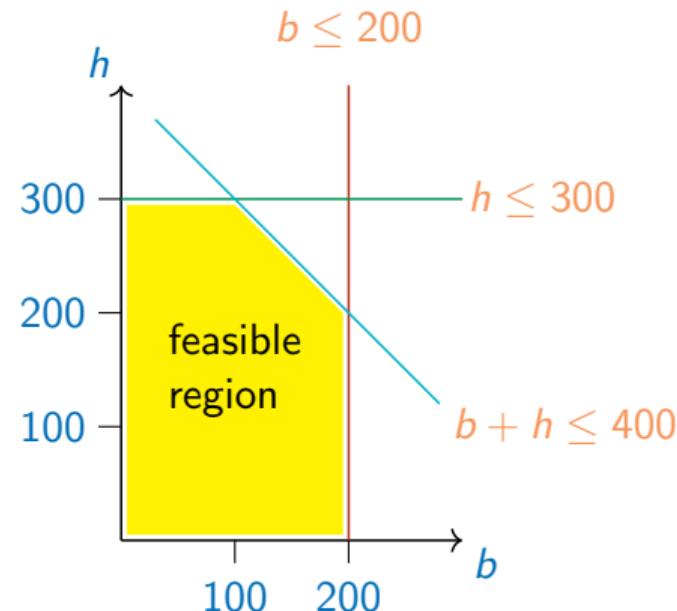
Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Pictorially



Objective: $c = 100b + 600h$

Linear program

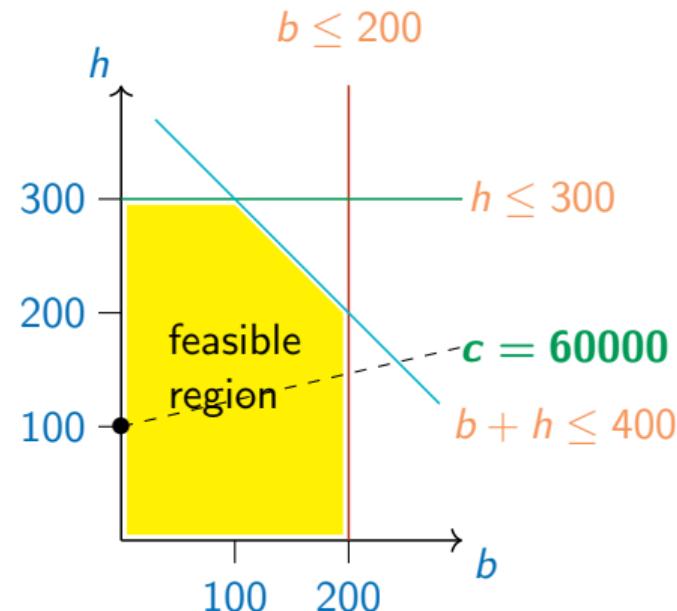
Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Pictorially



Objective: $c = 100b + 600h$

Linear program

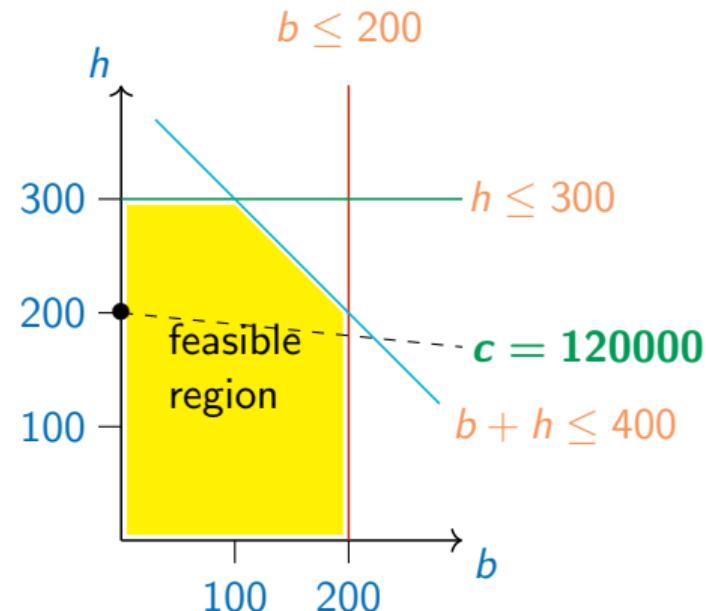
Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Pictorially



Objective: $c = 100b + 600h$

Linear program

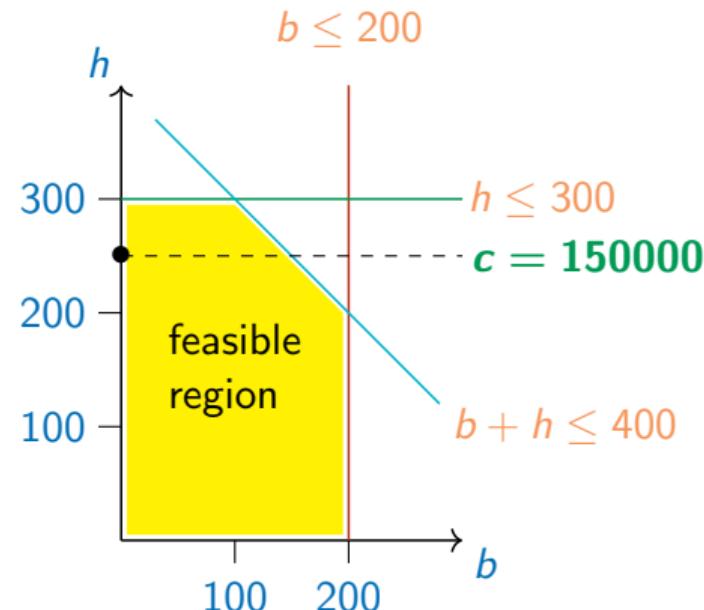
Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Pictorially



Objective: $c = 100b + 600h$

Linear program

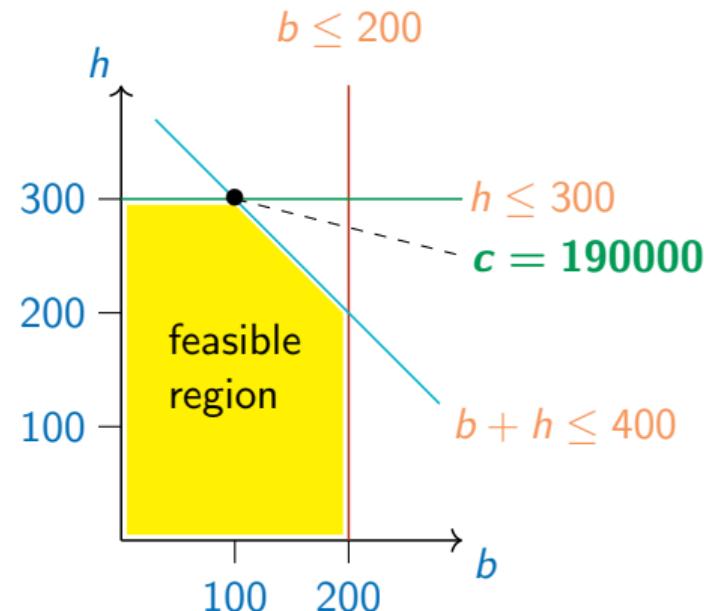
Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Pictorially



Objective: $c = 100b + 600h$

Linear program

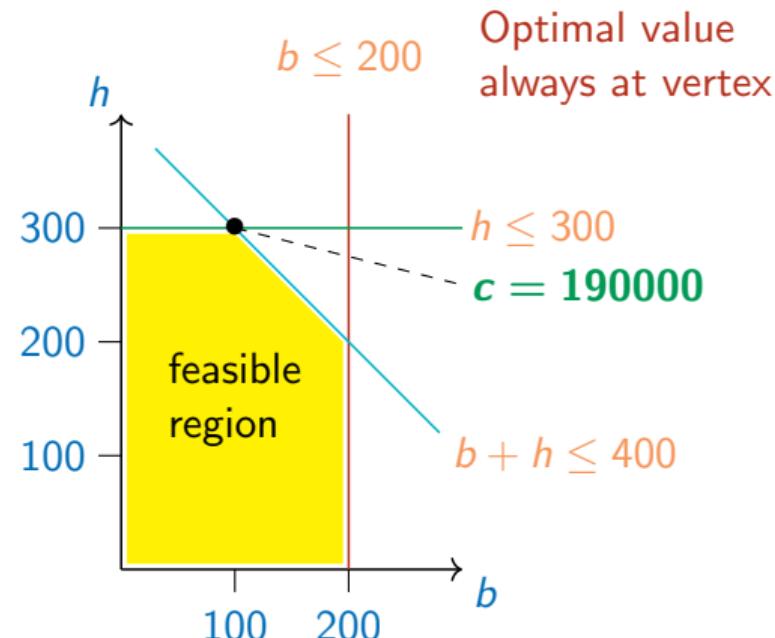
Objective

- Maximize $100b + 600h$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$

Pictorially



Objective: $c = 100b + 600h$



Solving linear programs

Simplex algorithm

- Start at any vertex, evaluate objective

Solving linear programs

Simplex algorithm

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move

Solving linear programs

Simplex algorithm

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbours, stop

Solving linear programs

Simplex algorithm

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbours, stop
- Can be exponential, but efficient in practice

Solving linear programs

Simplex algorithm

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbours, stop
- Can be exponential, but efficient in practice
- Theoretically efficient algorithms exist

Solving linear programs

Simplex algorithm

Existence of solutions

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbours, stop
- Can be exponential, but efficient in practice
- Theoretically efficient algorithms exist

Solving linear programs

Simplex algorithm

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbours, stop
- Can be exponential, but efficient in practice
- Theoretically efficient algorithms exist

Existence of solutions

- Feasible region is **convex**

Solving linear programs

Simplex algorithm

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbours, stop
- Can be exponential, but efficient in practice
- Theoretically efficient algorithms exist

Existence of solutions

- Feasible region is **convex**
- May be empty — constraints are unsatisfiable, no solutions

Solving linear programs

Simplex algorithm

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbours, stop
- Can be exponential, but efficient in practice
- Theoretically efficient algorithms exist

Existence of solutions

- Feasible region is **convex**
- May be empty — constraints are unsatisfiable, no solutions
- May be unbounded — no upper/lower limit on objective

Example, extended

Grandiose Sweets adds almond rasmalai

Example, extended

Grandiose Sweets adds almond rasmalai

- Profit per box: barfis – Rs 100, halwa – Rs 600, rasmalai – Rs 1300

Example, extended

Grandiose Sweets adds almond rasmalai

- Profit per box: barfis – Rs 100, halwa – Rs 600, rasmalai – Rs 1300
- Daily demand, in boxes: barfis – 200, halwa – 300, rasmalai – unlimited

Example, extended

Grandiose Sweets adds almond rasmalai

- Profit per box: barfis – Rs 100, halwa – Rs 600, rasmalai – Rs 1300
- Daily demand, in boxes: barfis – 200, halwa – 300, rasmalai – unlimited
- Production capacity: 400 boxes a day, altogether

Example, extended

Grandiose Sweets adds almond rasmalai

- Profit per box: barfis – Rs 100, halwa – Rs 600, rasmalai – Rs 1300
- Daily demand, in boxes: barfis – 200, halwa – 300, rasmalai – unlimited
- Production capacity: 400 boxes a day, altogether
- Milk supply is limited
 - 600 boxes halwa or 200 boxes rasmalai
 - Or any combination (rasmalai needs 3 times as much milk)

Example, extended

Grandiose Sweets adds almond rasmalai

- Profit per box: barfis – Rs 100, halwa – Rs 600, rasmalai – Rs 1300
- Daily demand, in boxes: barfis – 200, halwa – 300, rasmalai – unlimited
- Production capacity: 400 boxes a day, altogether
- Milk supply is limited
 - 600 boxes halwa or 200 boxes rasmalai
 - Or any combination (rasmalai needs 3 times as much milk)
- Most profitable mix to produce?

Example, extended

Grandiose Sweets adds almond rasmalai

New linear program

- Profit per box: barfis – Rs 100, halwa – Rs 600, rasmalai – Rs 1300
- Daily demand, in boxes: barfis – 200, halwa – 300, rasmalai – unlimited
- Production capacity: 400 boxes a day, altogether
- Milk supply is limited
 - 600 boxes halwa or 200 boxes rasmalai
 - Or any combination (rasmalai needs 3 times as much milk)
- Most profitable mix to produce?

Example, extended

Grandiose Sweets adds almond rasmalai

New linear program

- Profit per box: barfis – Rs 100, halwa – Rs 600, rasmalai – Rs 1300
- Daily demand, in boxes: barfis – 200, halwa – 300, rasmalai – unlimited
- Production capacity: 400 boxes a day, altogether
- Milk supply is limited
 - 600 boxes halwa or 200 boxes rasmalai
 - Or any combination (rasmalai needs 3 times as much milk)
- Most profitable mix to produce?

Objective

- Maximize $100b + 600h + 1300r$

Example, extended

Grandiose Sweets adds almond rasmalai

- Profit per box: barfis – Rs 100, halwa – Rs 600, rasmalai – Rs 1300
- Daily demand, in boxes: barfis – 200, halwa – 300, rasmalai – unlimited
- Production capacity: 400 boxes a day, altogether
- Milk supply is limited
 - 600 boxes halwa or 200 boxes rasmalai
 - Or any combination (rasmalai needs 3 times as much milk)
- Most profitable mix to produce?

New linear program

Objective

- Maximize $100b + 600h + 1300r$

Constraints

- $b \leq 200$
- $h \leq 300$

Example, extended

Grandiose Sweets adds almond rasmalai

- Profit per box: barfis – Rs 100, halwa – Rs 600, rasmalai – Rs 1300
- Daily demand, in boxes: barfis – 200, halwa – 300, rasmalai – unlimited
- Production capacity: 400 boxes a day, altogether
- Milk supply is limited
 - 600 boxes halwa or 200 boxes rasmalai
 - Or any combination (rasmalai needs 3 times as much milk)
- Most profitable mix to produce?

New linear program

Objective

- Maximize $100b + 600h + 1300r$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h + r \leq 400$

Example, extended

Grandiose Sweets adds almond rasmalai

- Profit per box: barfis – Rs 100, halwa – Rs 600, rasmalai – Rs 1300
- Daily demand, in boxes: barfis – 200, halwa – 300, rasmalai – unlimited
- Production capacity: 400 boxes a day, altogether
- Milk supply is limited
 - 600 boxes halwa or 200 boxes rasmalai
 - Or any combination (rasmalai needs 3 times as much milk)
- Most profitable mix to produce?

New linear program

Objective

- Maximize $100b + 600h + 1300r$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h + r \leq 400$
- $h + 3r \leq 600$

Example, extended

Grandiose Sweets adds almond rasmalai

- Profit per box: barfis – Rs 100, halwa – Rs 600, rasmalai – Rs 1300
- Daily demand, in boxes: barfis – 200, halwa – 300, rasmalai – unlimited
- Production capacity: 400 boxes a day, altogether
- Milk supply is limited
 - 600 boxes halwa or 200 boxes rasmalai
 - Or any combination (rasmalai needs 3 times as much milk)
- Most profitable mix to produce?

New linear program

Objective

- Maximize $100b + 600h + 1300r$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h + r \leq 400$
- $h + 3r \leq 600$
- $b \geq 0, h \geq 0, r \geq 0$

Example, extended

New linear program

Objective

- Maximize $100b + 600h + 1300r$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h + r \leq 400$
- $h + 3r \leq 600$
- $b \geq 0, h \geq 0, r \geq 0$

Example, extended

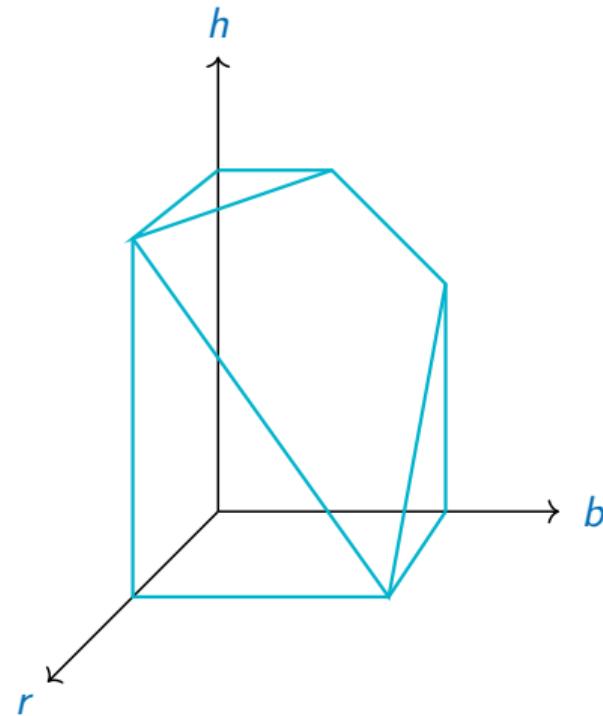
New linear program

Objective

- Maximize $100b + 600h + 1300r$

Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h + r \leq 400$
- $h + 3r \leq 600$
- $b \geq 0, h \geq 0, r \geq 0$



Example, extended

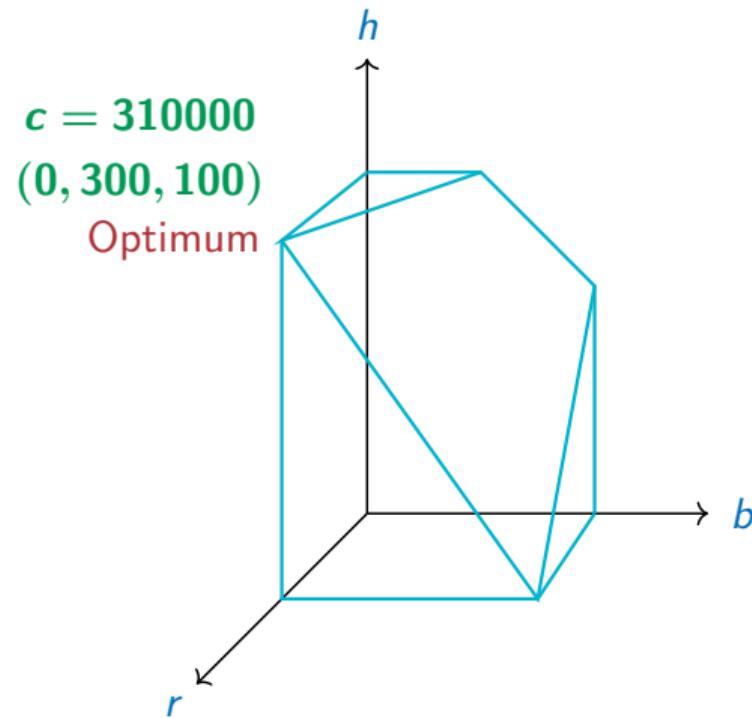
New linear program

Objective

- Maximize $100b + 600h + 1300r$

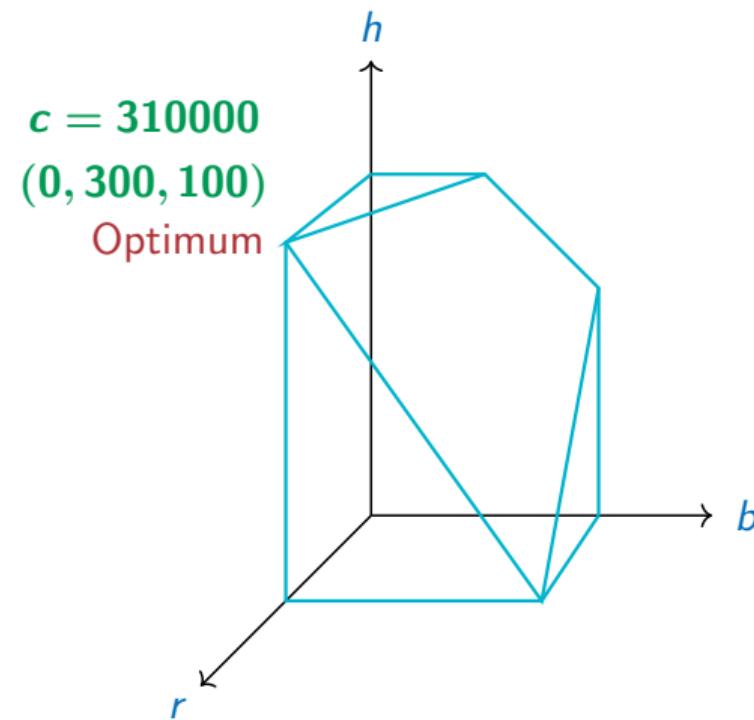
Constraints

- $b \leq 200$
- $h \leq 300$
- $b + h + r \leq 400$
- $h + 3r \leq 600$
- $b \geq 0, h \geq 0, r \geq 0$



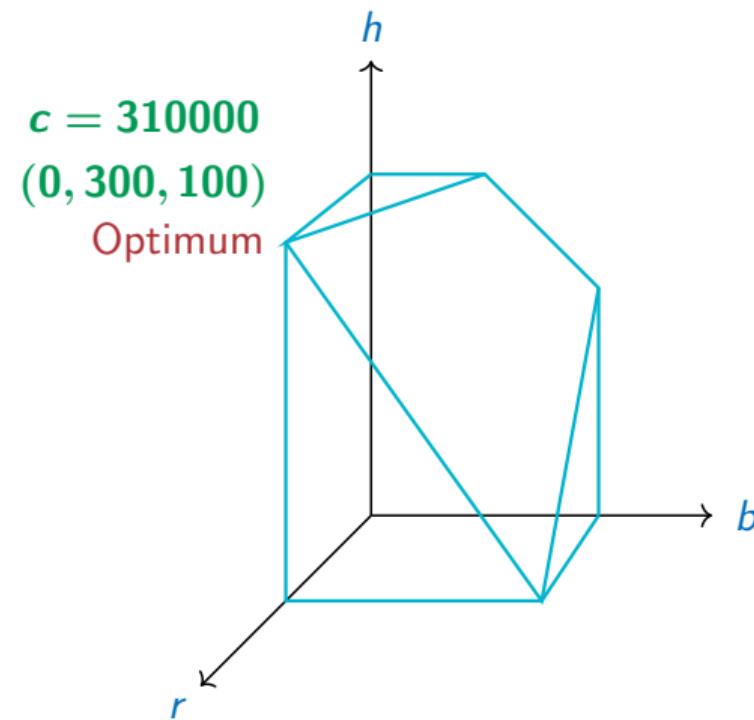
Example, extended

- Why is $(0, 300, 100)$ optimal?



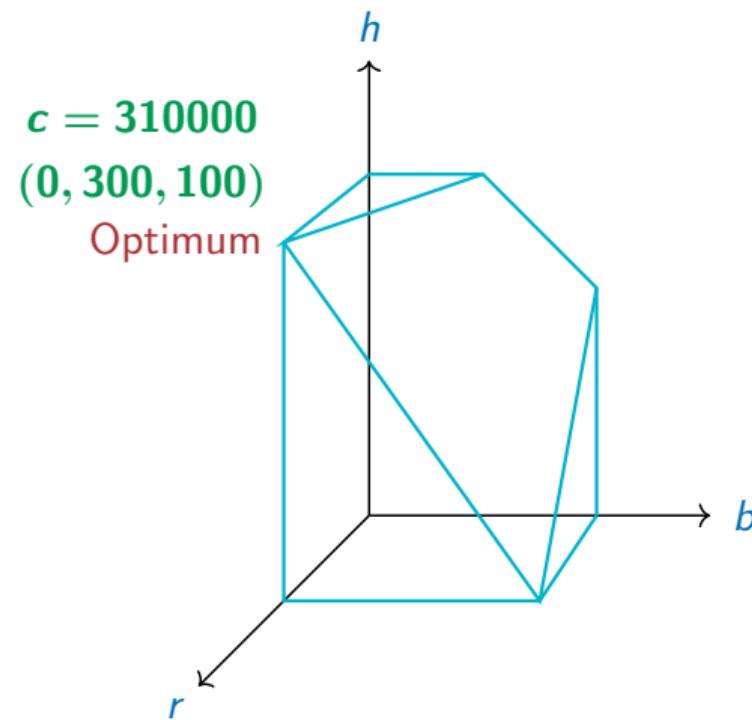
Example, extended

- Why is $(0, 300, 100)$ optimal?
- Profit is $100b + 600h + 1300r$



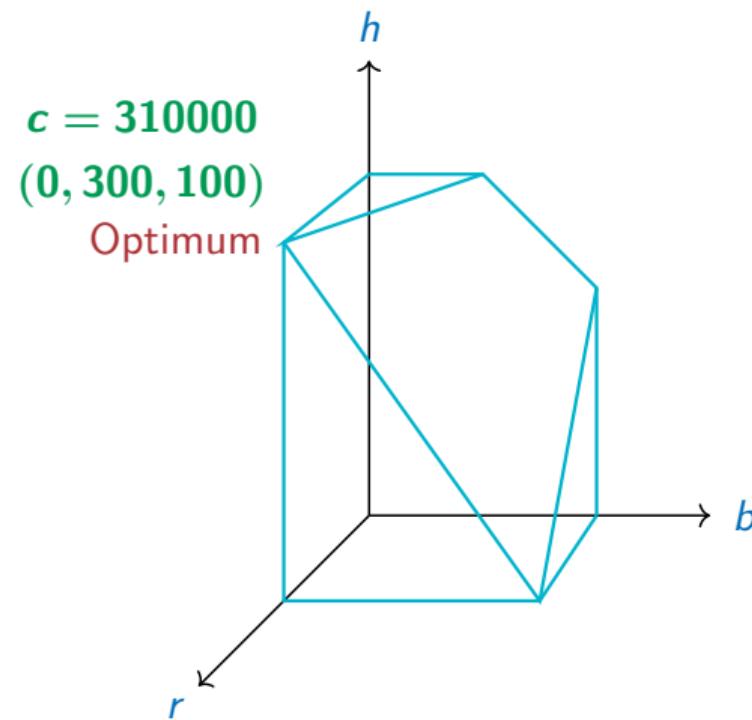
Example, extended

- Why is $(0, 300, 100)$ optimal?
- Profit is $100b + 600h + 1300r$
- Consider the following constraints
 - (A) $h \leq 300$
 - (B) $b + h + r \leq 400$
 - (C) $h + 3r \leq 600$



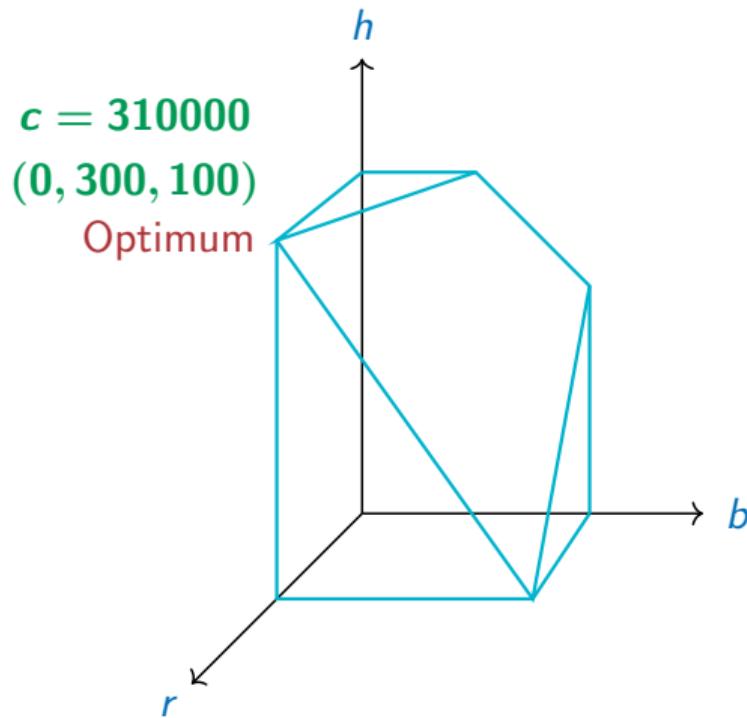
Example, extended

- Why is $(0, 300, 100)$ optimal?
- Profit is $100b + 600h + 1300r$
- Consider the following constraints
 - (A) $h \leq 300$
 - (B) $b + h + r \leq 400$
 - (C) $h + 3r \leq 600$
- Combine as
 $100 \cdot (A) + 100 \cdot (B) + 400 \cdot (C)$



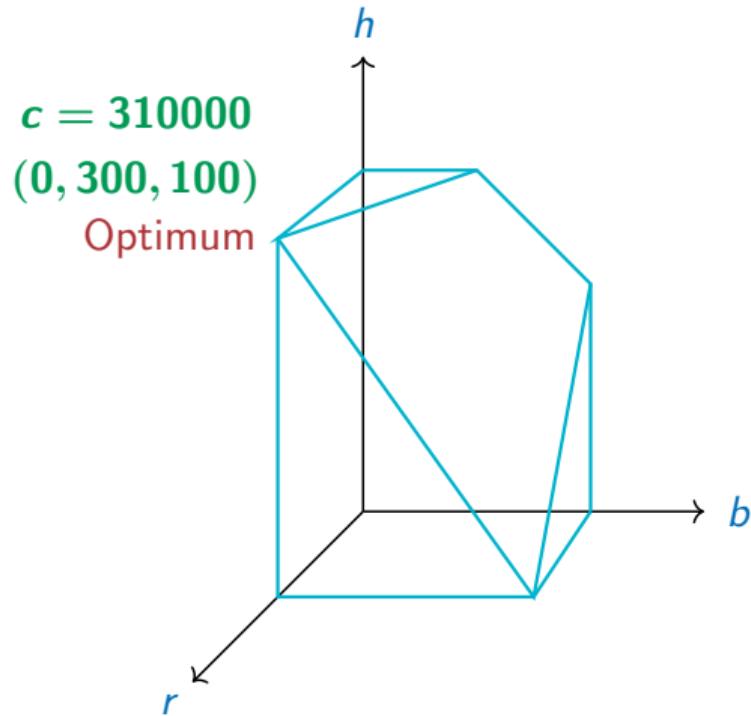
Example, extended

- Why is $(0, 300, 100)$ optimal?
- Profit is $100b + 600h + 1300r$
- Consider the following constraints
 - (A) $h \leq 300$
 - (B) $b + h + r \leq 400$
 - (C) $h + 3r \leq 600$
- Combine as
 $100 \cdot (A) + 100 \cdot (B) + 400 \cdot (C)$
- Result is
 $100b + 600h + 1300r \leq 310000$



Example, extended

- Why is $(0, 300, 100)$ optimal?
- Profit is $100b + 600h + 1300r$
- Consider the following constraints
 - (A) $h \leq 300$
 - (B) $b + h + r \leq 400$
 - (C) $h + 3r \leq 600$
- Combine as
 $100 \cdot (A) + 100 \cdot (B) + 400 \cdot (C)$
- Result is
 $100b + 600h + 1300r \leq 310000$
- LHS is profit, so value at $(0, 300, 100)$ matches upper bound on profit



LP Duality

- We derived an upper bound on the objective through a linear combination of constraints
- Why is $(0, 300, 100)$ optimal?
- Profit is $100b + 600h + 1300r$
- Consider the following constraints
 - (A) $h \leq 300$
 - (B) $b + h + r \leq 400$
 - (C) $h + 3r \leq 600$
- Combine as
 $100 \cdot (A) + 100 \cdot (B) + 400 \cdot (C)$
- Result is
 $100b + 600h + 1300r \leq 310000$
- LHS is profit, so value at $(0, 300, 100)$ matches upper bound on profit

LP Duality

- We derived an upper bound on the objective through a linear combination of constraints
- This is **always** possible!
- Why is $(0, 300, 100)$ optimal?
- Profit is $100b + 600h + 1300r$
- Consider the following constraints
 - (A) $h \leq 300$
 - (B) $b + h + r \leq 400$
 - (C) $h + 3r \leq 600$
- Combine as
 $100 \cdot (A) + 100 \cdot (B) + 400 \cdot (C)$
- Result is
 $100b + 600h + 1300r \leq 310000$
- LHS is profit, so value at $(0, 300, 100)$ matches upper bound on profit

LP Duality

- We derived an upper bound on the objective through a linear combination of constraints
- This is **always** possible!
- Dual LP problem
 - Minimize linear combination of constraints
 - Variables are multipliers for the linear combination
 - Implicit constraint: multipliers are non-negative
 - Optimum solution solves both the original (primal) and the dual LP
- Why is $(0, 300, 100)$ optimal?
- Profit is $100b + 600h + 1300r$
- Consider the following constraints
 - (A) $h \leq 300$
 - (B) $b + h + r \leq 400$
 - (C) $h + 3r \leq 600$
- Combine as
 $100 \cdot (A) + 100 \cdot (B) + 400 \cdot (C)$
- Result is
 $100b + 600h + 1300r \leq 310000$
- LHS is profit, so value at $(0, 300, 100)$ matches upper bound on profit

Linear Programming: Production Planning

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 11

Linear programming

- Constraints and objective to be optimized are **linear** functions
 - **Constraints:** $a_1x_1 + a_2x_2 + \dots + a_mx_m \leq K$, $b_1x_1 + b_2x_2 + \dots + b_mx_m \geq L$, ...
 - **Objective:** $c_1x_1 + c_2x_2 + \dots + c_mx_m$
- Defines a convex feasible region

Linear programming

- Constraints and objective to be optimized are **linear** functions
 - **Constraints:** $a_1x_1 + a_2x_2 + \dots + a_mx_m \leq K$, $b_1x_1 + b_2x_2 + \dots + b_mx_m \geq L$, ...
 - **Objective:** $c_1x_1 + c_2x_2 + \dots + c_mx_m$
- Defines a convex feasible region

Simplex algorithm

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbours, stop

Linear programming

- Constraints and objective to be optimized are **linear** functions
 - **Constraints:** $a_1x_1 + a_2x_2 + \dots + a_mx_m \leq K$, $b_1x_1 + b_2x_2 + \dots + b_mx_m \geq L$, ...
 - **Objective:** $c_1x_1 + c_2x_2 + \dots + c_mx_m$
- Defines a convex feasible region

Simplex algorithm

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbours, stop
- Can be exponential, but efficient in practice
- Theoretically efficient algorithms exist

- Can **always** construct a linear combination of constraints that tightly captures upper bound on objective function
- Dual LP problem
 - Minimize linear combination of constraints
 - Variables are multipliers for the linear combination
 - Implicit constraint: multipliers are non-negative
 - Optimum solution solves both the original (primal) and the dual LP

Production planning

Handwoven carpets

Production planning

Handwoven carpets

- 30 employees,
 - Each produces 20 carpets a month
 - Salary Rs 20,000
 - Labour cost: Rs 1000 per carpet

Production planning

Handwoven carpets

- 30 employees,
 - Each produces 20 carpets a month
 - Salary Rs 20,000
 - Labour cost: Rs 1000 per carpet
- Seasonal monthly demand
 - d_1, d_2, \dots, d_{12} , demand from January to December

Production planning

Handwoven carpets

Coping with varying demand

- 30 employees,
 - Each produces 20 carpets a month
 - Salary Rs 20,000
 - Labour cost: Rs 1000 per carpet
- Seasonal monthly demand
 - d_1, d_2, \dots, d_{12} , demand from January to December

Production planning

Handwoven carpets

- 30 employees,
 - Each produces 20 carpets a month
 - Salary Rs 20,000
 - Labour cost: Rs 1000 per carpet
- Seasonal monthly demand
 - d_1, d_2, \dots, d_{12} , demand from January to December

Coping with varying demand

- Overtime
 - Pay 80% extra
 - Overtime limit is 30% per worker

Production planning

Handwoven carpets

- 30 employees,
 - Each produces 20 carpets a month
 - Salary Rs 20,000
 - Labour cost: Rs 1000 per carpet
- Seasonal monthly demand
 - d_1, d_2, \dots, d_{12} , demand from January to December

Coping with varying demand

- Overtime
 - Pay 80% extra
 - Overtime limit is 30% per worker
- Hiring and firing
 - Hiring costs Rs 3200 per worker
 - Firing costs Rs 4000 per worker

Production planning

Handwoven carpets

- 30 employees,
 - Each produces 20 carpets a month
 - Salary Rs 20,000
 - Labour cost: Rs 1000 per carpet
- Seasonal monthly demand
 - d_1, d_2, \dots, d_{12} , demand from January to December

Coping with varying demand

- Overtime
 - Pay 80% extra
 - Overtime limit is 30% per worker
- Hiring and firing
 - Hiring costs Rs 3200 per worker
 - Firing costs Rs 4000 per worker
- Make surplus and store
 - Costs Rs 80 per carpet

Formulate a linear program

- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, d_2, \dots, d_{12}
- Overtime: pay 80% extra, overtime limit is 30% per worker
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

Formulate a linear program

- w_i : workers in month i , $w_0 = 30$
- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, d_2, \dots, d_{12}
- Overtime: pay 80% extra, overtime limit is 30% per worker
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

Formulate a linear program

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, d_2, \dots, d_{12}
- Overtime: pay 80% extra, overtime limit is 30% per worker
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

Formulate a linear program

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- o_i : carpets made in overtime, month i
- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, d_2, \dots, d_{12}
- Overtime: pay 80% extra, overtime limit is 30% per worker
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

Formulate a linear program

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- o_i : carpets made in overtime, month i
- h_i : workers hired at start of month i
- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, d_2, \dots, d_{12}
- Overtime: pay 80% extra, overtime limit is 30% per worker
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

Formulate a linear program

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- o_i : carpets made in overtime, month i
- h_i : workers hired at start of month i
- f_i : workers fired at start of month i
- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, d_2, \dots, d_{12}
- Overtime: pay 80% extra, overtime limit is 30% per worker
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

Formulate a linear program

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- o_i : carpets made in overtime, month i
- h_i : workers hired at start of month i
- f_i : workers fired at start of month i
- s_i : surplus carpets after month i
 - $s_0 = 0$
- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, d_2, \dots, d_{12}
- Overtime: pay 80% extra, overtime limit is 30% per worker
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

Formulate a linear program

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- o_i : carpets made in overtime, month i
- h_i : workers hired at start of month i
- f_i : workers fired at start of month i
- s_i : surplus carpets after month i
 - $s_0 = 0$
- 72 variables, plus w_0 , s_0
- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, d_2, \dots, d_{12}
- Overtime: pay 80% extra, overtime limit is 30% per worker
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

Formulate a linear program

- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, \dots, d_{12}
- Overtime: 80% extra, limit 30%
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- o_i : carpets made in overtime, month i
- h_i : workers hired at start of month i
- f_i : workers fired at start of month i
- s_i : surplus after month i , $s_0 = 0$

Formulate a linear program

- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, \dots, d_{12}
- Overtime: 80% extra, limit 30%
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- o_i : carpets made in overtime, month i
- h_i : workers hired at start of month i
- f_i : workers fired at start of month i
- s_i : surplus after month i , $s_0 = 0$

Constraints

- All variables are nonnegative
 - $w_i, x_i, o_i, h_i, f_i, s_i \geq 0$

Formulate a linear program

- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, \dots, d_{12}
- Overtime: 80% extra, limit 30%
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- o_i : carpets made in overtime, month i
- h_i : workers hired at start of month i
- f_i : workers fired at start of month i
- s_i : surplus after month i , $s_0 = 0$

Constraints

- All variables are nonnegative
 - $w_i, x_i, o_i, h_i, f_i, s_i \geq 0$
- Carpets made = regular + overtime
 - $x_i = 20w_i + o_i$

Formulate a linear program

- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, \dots, d_{12}
- Overtime: 80% extra, limit 30%
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- o_i : carpets made in overtime, month i
- h_i : workers hired at start of month i
- f_i : workers fired at start of month i
- s_i : surplus after month i , $s_0 = 0$

Constraints

- All variables are nonnegative
 - $w_i, x_i, o_i, h_i, f_i, s_i \geq 0$
- Carpets made = regular + overtime
 - $x_i = 20w_i + o_i$
- Number of workers match hiring/firing
 - $w_i = w_{i-1} + h_i - f_i$

Formulate a linear program

- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, \dots, d_{12}
- Overtime: 80% extra, limit 30%
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- o_i : carpets made in overtime, month i
- h_i : workers hired at start of month i
- f_i : workers fired at start of month i
- s_i : surplus after month i , $s_0 = 0$

Constraints

- All variables are nonnegative
 - $w_i, x_i, o_i, h_i, f_i, s_i \geq 0$
- Carpets made = regular + overtime
 - $x_i = 20w_i + o_i$
- Number of workers match hiring/firing
 - $w_i = w_{i-1} + h_i - f_i$
- Number of stored carpets connected to earlier stock, production, demand
 - $s_i = s_{i-1} + x_i - d_i$

Formulate a linear program

- 30 employees, each 20 carpets a month, salary Rs 20,000, Rs 1000 per carpet
- Monthly demand d_1, \dots, d_{12}
- Overtime: 80% extra, limit 30%
- Hiring cost Rs 3200, firing cost Rs 4000
- Surplus storage cost: Rs 80 per carpet

- w_i : workers in month i , $w_0 = 30$
- x_i : carpets made in month i
- o_i : carpets made in overtime, month i
- h_i : workers hired at start of month i
- f_i : workers fired at start of month i
- s_i : surplus after month i , $s_0 = 0$

Constraints

- All variables are nonnegative
 - $w_i, x_i, o_i, h_i, f_i, s_i \geq 0$
- Carpets made = regular + overtime
 - $x_i = 20w_i + o_i$
- Number of workers match hiring/firing
 - $w_i = w_{i-1} + h_i - f_i$
- Number of stored carpets connected to earlier stock, production, demand
 - $s_i = s_{i-1} + x_i - d_i$
- Overtime production at most 6 carpets per worker (30% of regular production)
 - $o_i \leq 6w_i$

Formulate a linear program

Constraints

- $w_0 = 30, s_0 = 0$

For each $i \in \{1, 2, \dots, 12\}$

- $w_i, x_i, o_i, h_i, f_i, s_i \geq 0$
- $x_i = 20w_i + o_i$
- $w_i = w_{i-1} + h_i - f_i$
- $s_i = s_{i-1} + x_i - d_i$
- $o_i \leq 6w_i$

Formulate a linear program

Constraints

- $w_0 = 30, s_0 = 0$

For each $i \in \{1, 2, \dots, 12\}$

- $w_i, x_i, o_i, h_i, f_i, s_i \geq 0$
- $x_i = 20w_i + o_i$
- $w_i = w_{i-1} + h_i - f_i$
- $s_i = s_{i-1} + x_i - d_i$
- $o_i \leq 6w_i$

Objective

- Minimize the cost

$$\begin{aligned} & 20000(w_1 + w_2 + \dots + w_{12}) + \\ & 3200(h_1 + h_2 + \dots + h_{12}) + \\ & 4000(f_1 + f_2 + \dots + f_{12}) + \\ & 80(s_1 + s_2 + \dots + s_{12}) + \\ & 1800(o_1 + o_2 + \dots + o_{12}) \end{aligned}$$

Solving the linear program

- Run Simplex and find a solution

Solving the linear program

- Run Simplex and find a solution
- Are we done?

Solving the linear program

- Run Simplex and find a solution
- Are we done?
- Optimum may have fractional values
 - Hire 10.6 workers in March

Solving the linear program

- Run Simplex and find a solution
- Are we done?
- Optimum may have fractional values
 - Hire 10.6 workers in March

Handling fractional solutions

Solving the linear program

- Run Simplex and find a solution
- Are we done?
- Optimum may have fractional values
 - Hire 10.6 workers in March

Handling fractional solutions

- Round off to 10 or 11 and recompute cost

Solving the linear program

- Run Simplex and find a solution
- Are we done?
- Optimum may have fractional values
 - Hire 10.6 workers in March

Handling fractional solutions

- Round off to 10 or 11 and recompute cost
- If values are “large”, rounding does not affect quality of solution much

Solving the linear program

- Run Simplex and find a solution
- Are we done?
- Optimum may have fractional values
 - Hire 10.6 workers in March

Handling fractional solutions

- Round off to 10 or 11 and recompute cost
- If values are “large”, rounding does not affect quality of solution much
- Values are “small”, need more care when rounding

Solving the linear program

- Run Simplex and find a solution
- Are we done?
- Optimum may have fractional values
 - Hire 10.6 workers in March

Handling fractional solutions

- Round off to 10 or 11 and recompute cost
- If values are “large”, rounding does not affect quality of solution much
- Values are “small”, need more care when rounding
- Insisting on integer solutions makes the problem computationally intractable

Integer Linear Programming

Linear Programming: Bandwidth Allocation

Madhavan Mukund

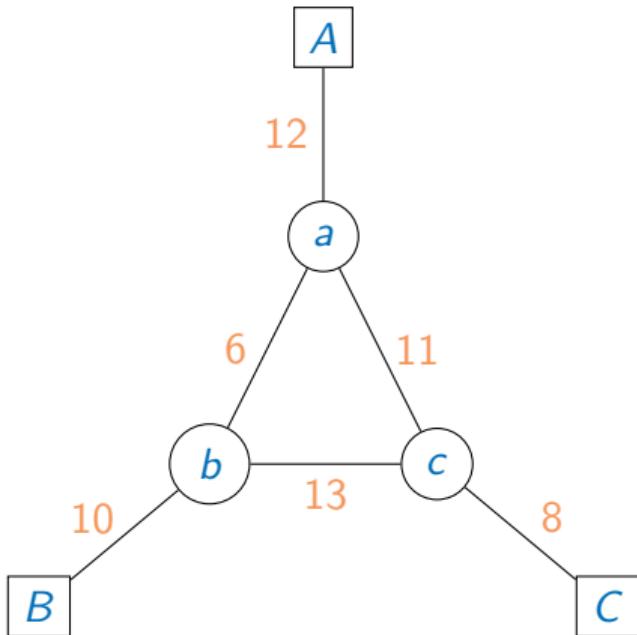
<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 11

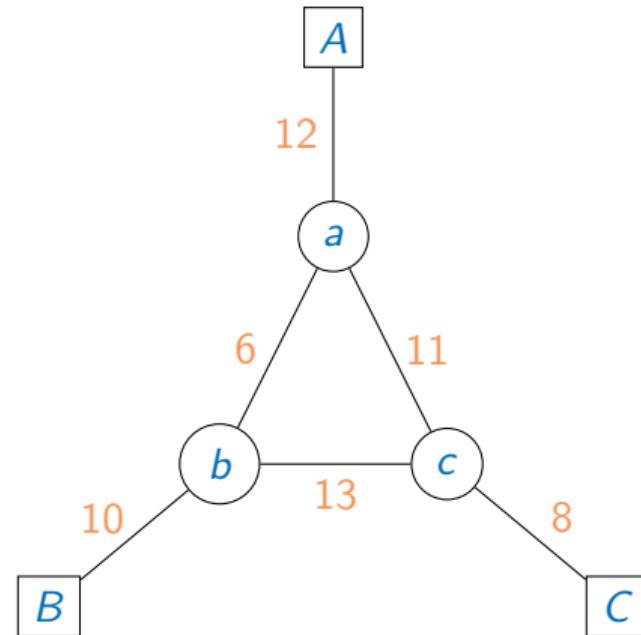
Network bandwidth

- 3 users, A , B , C to be connected to each other



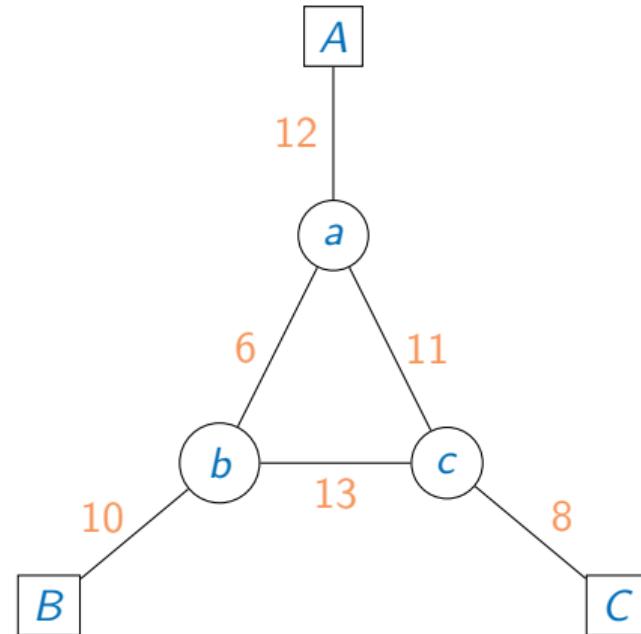
Network bandwidth

- 3 users, A , B , C to be connected to each other
- Link have capacity constraints (in Mbps)



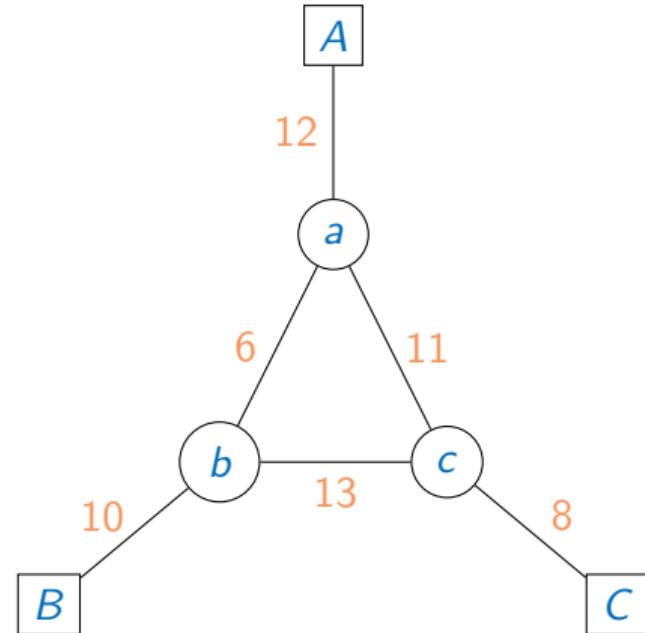
Network bandwidth

- 3 users, A , B , C to be connected to each other
- Link have capacity constraints (in Mbps)
- Each connection $A-B$, $B-C$, $A-C$ should have at least 2 Mbps of bandwidth



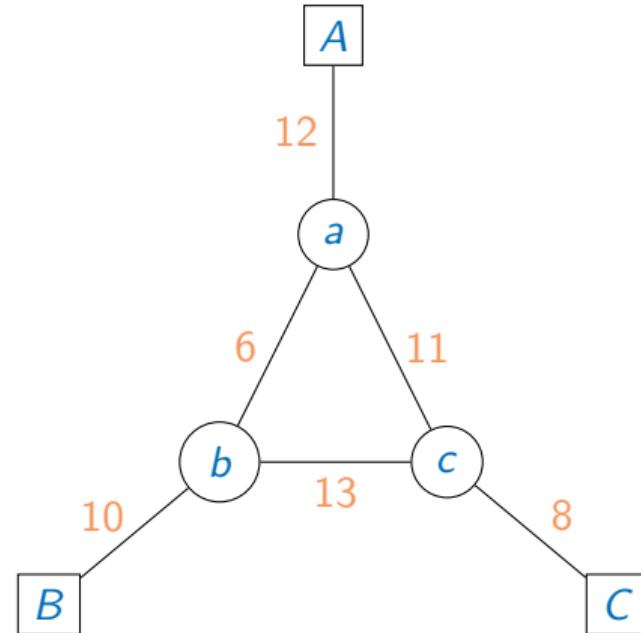
Network bandwidth

- 3 users, A , B , C to be connected to each other
- Link have capacity constraints (in Mbps)
- Each connection $A-B$, $B-C$, $A-C$ should have at least 2 Mbps of bandwidth
- Direct and/or indirect connections allowed:
 - $A-a-b-B$
 - $A-a-c-b-B$



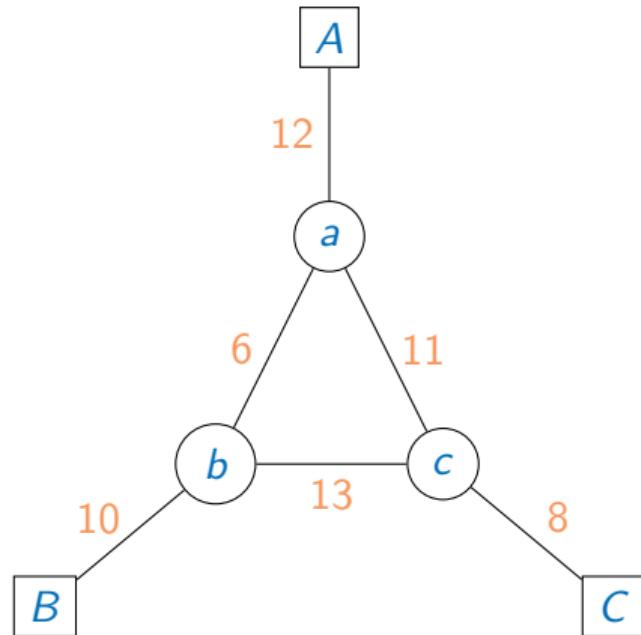
Network bandwidth

- 3 users, A , B , C to be connected to each other
- Link have capacity constraints (in Mbps)
- Each connection $A-B$, $B-C$, $A-C$ should have at least 2 Mbps of bandwidth
- Direct and/or indirect connections allowed:
 - $A-a-b-B$
 - $A-a-c-b-B$
- Each connection earns revenue, per Mbps
 - $A-B$, Rs 300/Mbps
 - $B-C$, Rs 200/Mbps
 - $A-C$, Rs 400/Mbps



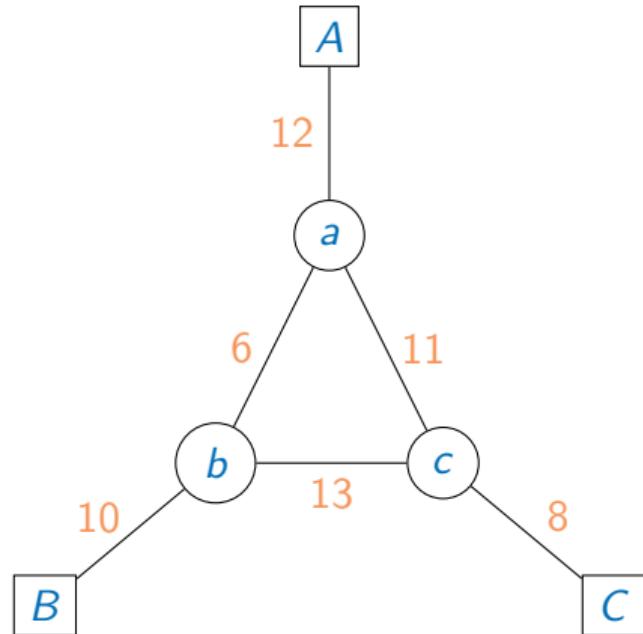
Network bandwidth

- 3 users, A , B , C to be connected to each other
- Link have capacity constraints (in Mbps)
- Each connection $A-B$, $B-C$, $A-C$ should have at least 2 Mbps of bandwidth
- Direct and/or indirect connections allowed:
 - $A-a-b-B$
 - $A-a-c-b-B$
- Each connection earns revenue, per Mbps
 - $A-B$, Rs 300/Mbps
 - $B-C$, Rs 200/Mbps
 - $A-C$, Rs 400/Mbps
- Allocate bandwidth to maximize revenue



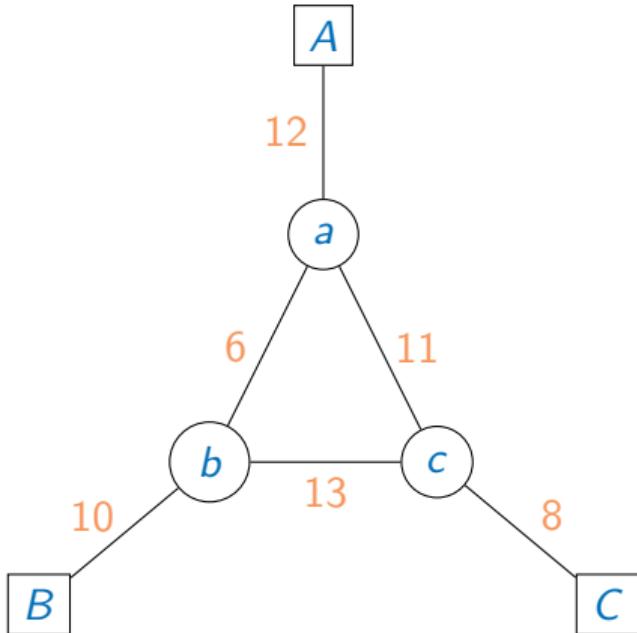
Linear program

- x_{AB} – bandwidth via short connection
 $A-a-b-B$,



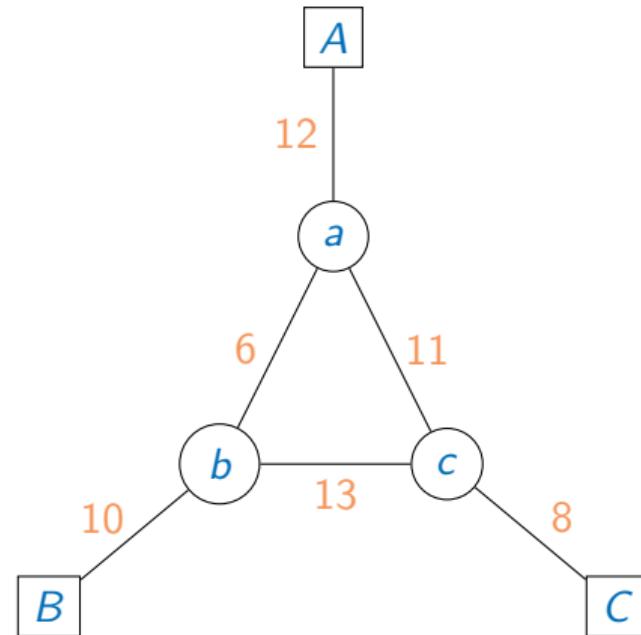
Linear program

- x_{AB} – bandwidth via short connection
 $A-a-b-B$,
- y_{AB} – bandwidth via long connection
 $A-a-c-b-B$



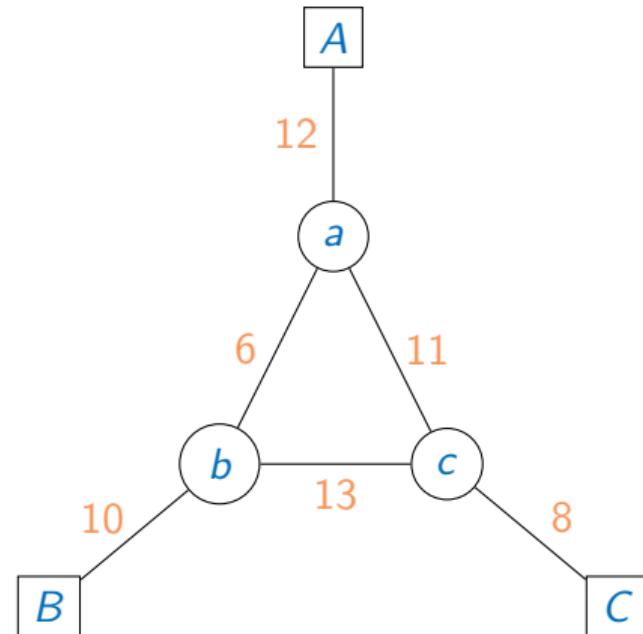
Linear program

- x_{AB} – bandwidth via short connection
 $A-a-b-B$,
- y_{AB} – bandwidth via long connection
 $A-a-c-b-B$
- Likewise, x_{AC} , y_{AC} , x_{BC} , y_{BC}



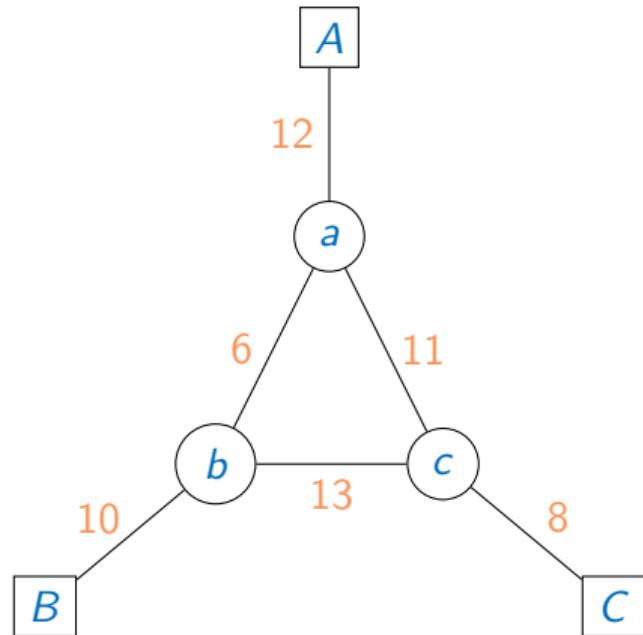
Linear program

- x_{AB} – bandwidth via short connection
 $A-a-b-B$,
- y_{AB} – bandwidth via long connection
 $A-a-c-b-B$
- Likewise, x_{AC} , y_{AC} , x_{BC} , y_{BC}
- x_{AB} , y_{AB} both flow via edge $b-B$, as do x_{BC} , y_{BC}
 - $x_{AB} + y_{AB} + x_{BC} + y_{BC} \leq 10$



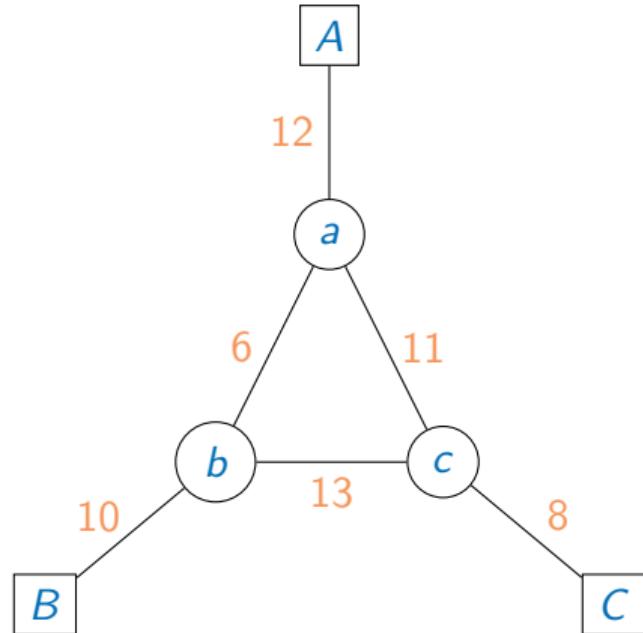
Linear program

- x_{AB} – bandwidth via short connection
 $A-a-b-B$,
- y_{AB} – bandwidth via long connection
 $A-a-c-b-B$
- Likewise, x_{AC} , y_{AC} , x_{BC} , y_{BC}
- x_{AB} , y_{AB} both flow via edge $b-B$, as do x_{BC} , y_{BC}
 - $x_{AB} + y_{AB} + x_{BC} + y_{BC} \leq 10$
- Likewise
 - $x_{AB} + y_{AB} + x_{AC} + y_{AC} \leq 12$
 - $x_{AC} + y_{AC} + x_{BC} + y_{BC} \leq 8$



Linear program

- x_{AB} , y_{AC} , y_{BC} all flow via edge $a-b$
 - $x_{AB} + y_{AC} + y_{BC} \leq 6$



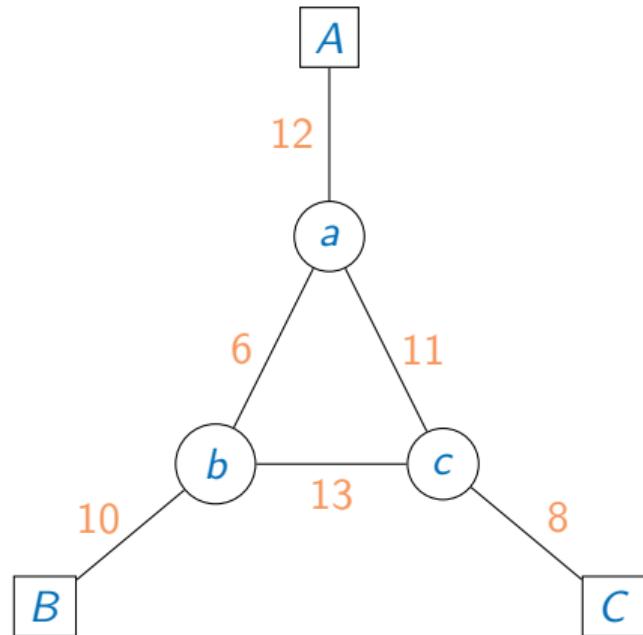
Linear program

- x_{AB} , y_{AC} , y_{BC} all flow via edge $a-b$

- $x_{AB} + y_{AC} + y_{BC} \leq 6$

- Likewise

- $y_{AB} + x_{BC} + y_{AC} \leq 13$
 - $y_{AB} + y_{BC} + x_{AC} \leq 11$



Linear program

- x_{AB} , y_{AC} , y_{BC} all flow via edge $a-b$

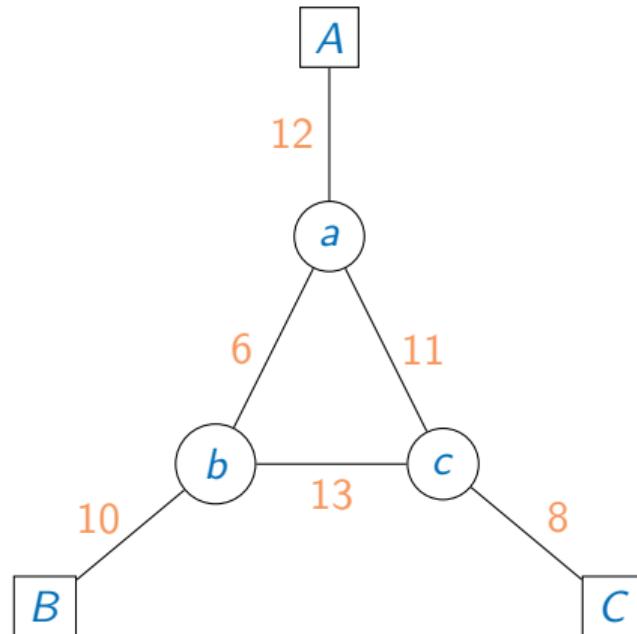
- $x_{AB} + y_{AC} + y_{BC} \leq 6$

- Likewise

- $y_{AB} + x_{BC} + y_{AC} \leq 13$
 - $y_{AB} + y_{BC} + x_{AC} \leq 11$

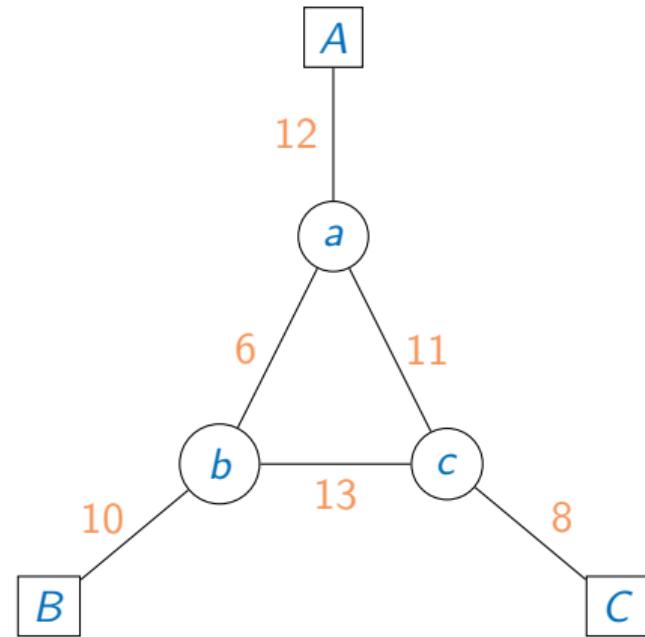
- Pairwise bandwidth at least 2 Mbps

- $x_{AB} + y_{AB} \geq 2$
 - $x_{BC} + y_{BC} \geq 2$
 - $x_{AC} + y_{AC} \geq 2$



Linear program

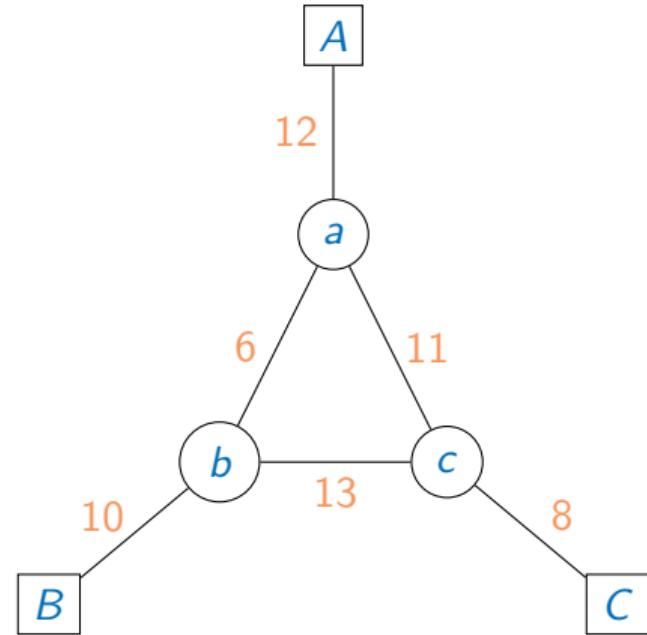
- x_{AB}, y_{AC}, y_{BC} all flow via edge $a-b$
 - $x_{AB} + y_{AC} + y_{BC} \leq 6$
- Likewise
 - $y_{AB} + x_{BC} + y_{AC} \leq 13$
 - $y_{AB} + y_{BC} + x_{AC} \leq 11$
- Pairwise bandwidth at least 2 Mbps
 - $x_{AB} + y_{AB} \geq 2$
 - $x_{BC} + y_{BC} \geq 2$
 - $x_{AC} + y_{AC} \geq 2$
- Traffic on all routes is nonnegative
 - $x_{AB}, y_{AB}, x_{AC}, y_{AC}, x_{BC}, y_{AC} \geq 0$



Objective

- Revenue

- $A-B$, Rs 300/Mbps
- $B-C$, Rs 200/Mbps
- $A-C$, Rs 400/Mbps



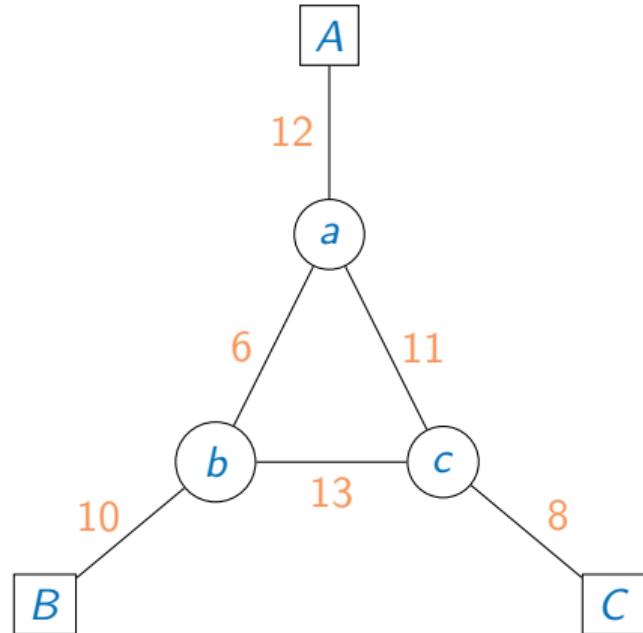
Objective

- Revenue

- $A-B$, Rs 300/Mbps
- $B-C$, Rs 200/Mbps
- $A-C$, Rs 400/Mbps

- Maximize

- $300(x_{AB} + y_{AB}) +$
 $200(x_{BC} + y_{BC})$
 $400(x_{AC} + y_{AC}) +$



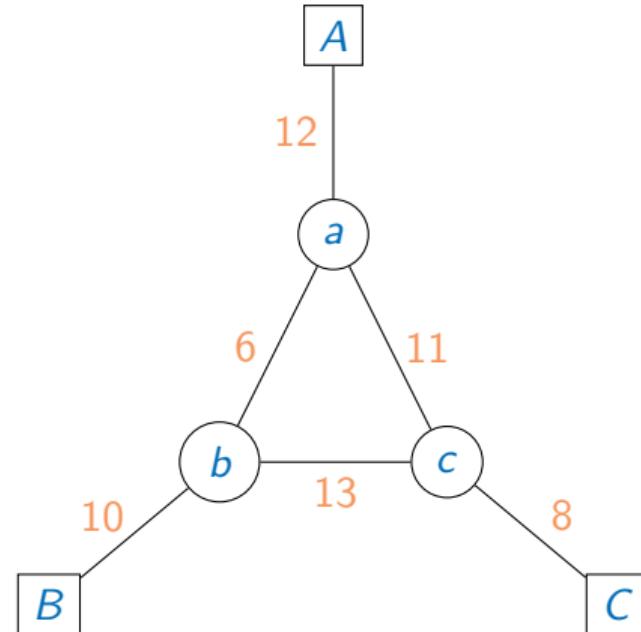
Solution

- Simplex yields

- $x_{AB} = 0, y_{AB} = 7$

- $x_{BC} = 1.5, y_{BC} = 1.5$

- $x_{AC} = 0.5, y_{AC} = 4.5$



Solution

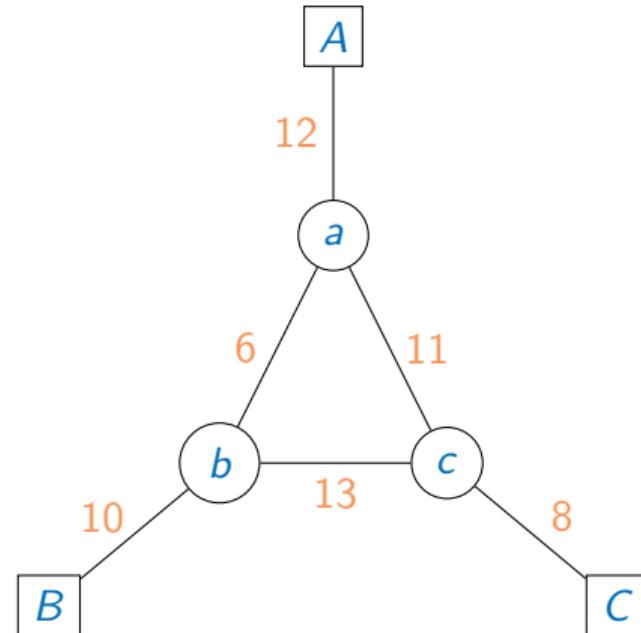
- Simplex yields

- $x_{AB} = 0, y_{AB} = 7$

- $x_{BC} = 1.5, y_{BC} = 1.5$

- $x_{AC} = 0.5, y_{AC} = 4.5$

- Fractional solutions are OK



Solution

- Simplex yields

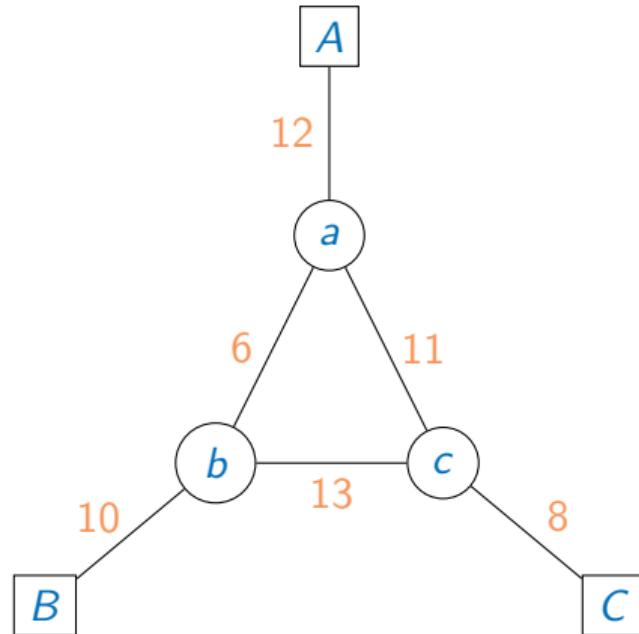
- $x_{AB} = 0, y_{AB} = 7$

- $x_{BC} = 1.5, y_{BC} = 1.5$

- $x_{AC} = 0.5, y_{AC} = 4.5$

- Fractional solutions are OK

- All edges full capacity except $a-c$



Solution

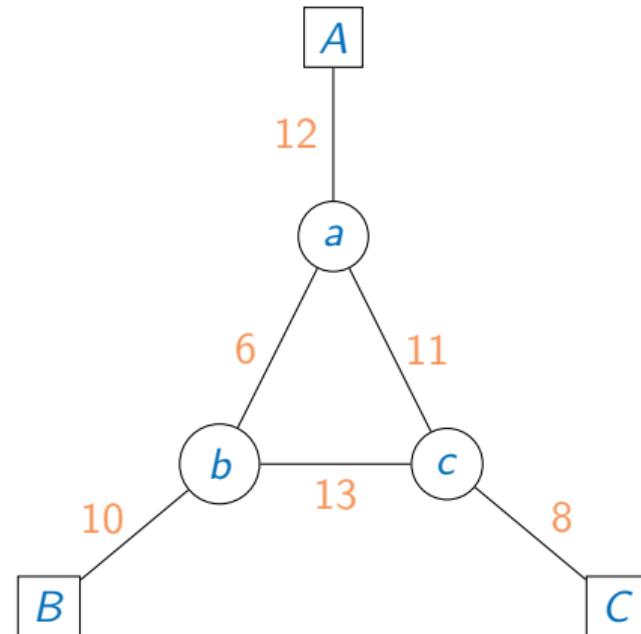
- Simplex yields

- $x_{AB} = 0, y_{AB} = 7$
 - $x_{BC} = 1.5, y_{BC} = 1.5$
 - $x_{AC} = 0.5, y_{AC} = 4.5$

- Fractional solutions are OK

- All edges full capacity except $a-c$

Note



Solution

- Simplex yields

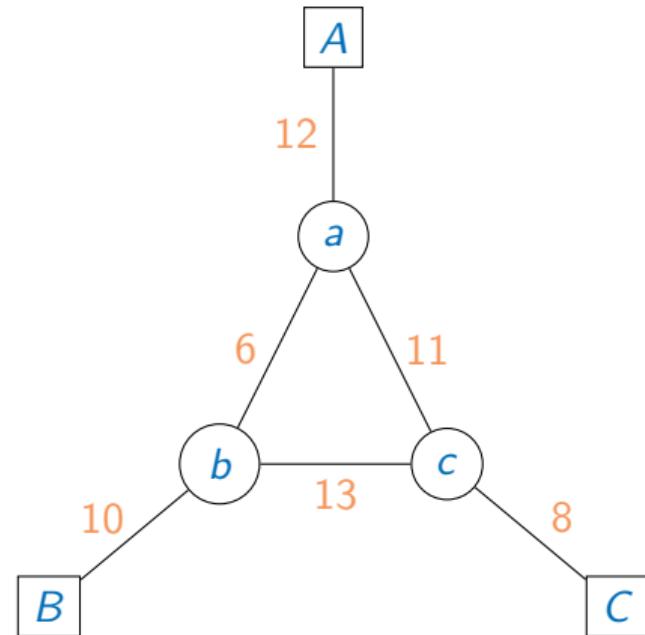
- $x_{AB} = 0, y_{AB} = 7$
 - $x_{BC} = 1.5, y_{BC} = 1.5$
 - $x_{AC} = 0.5, y_{AC} = 4.5$

- Fractional solutions are OK

- All edges full capacity except $a-c$

Note

- Modelling strategy does not scale well



Solution

- Simplex yields

- $x_{AB} = 0, y_{AB} = 7$

- $x_{BC} = 1.5, y_{BC} = 1.5$

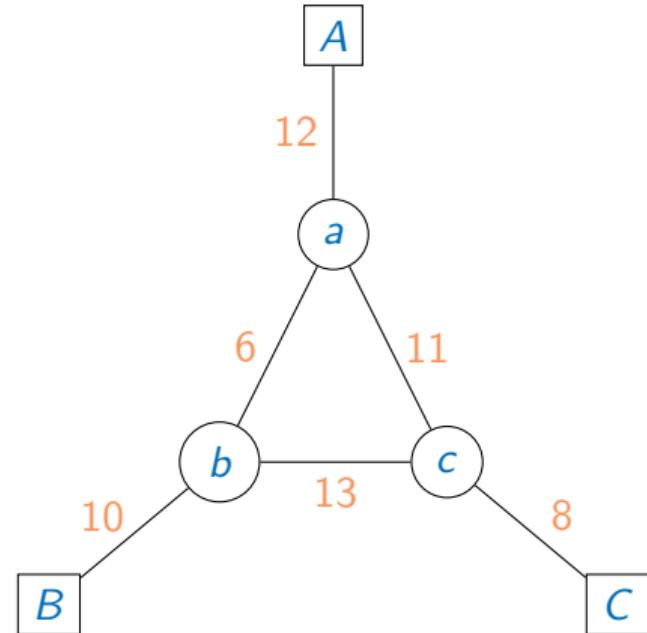
- $x_{AC} = 0.5, y_{AC} = 4.5$

- Fractional solutions are OK

- All edges full capacity except $a-c$

Note

- Modelling strategy does not scale well
- One variable per path — number of paths is exponential, in general

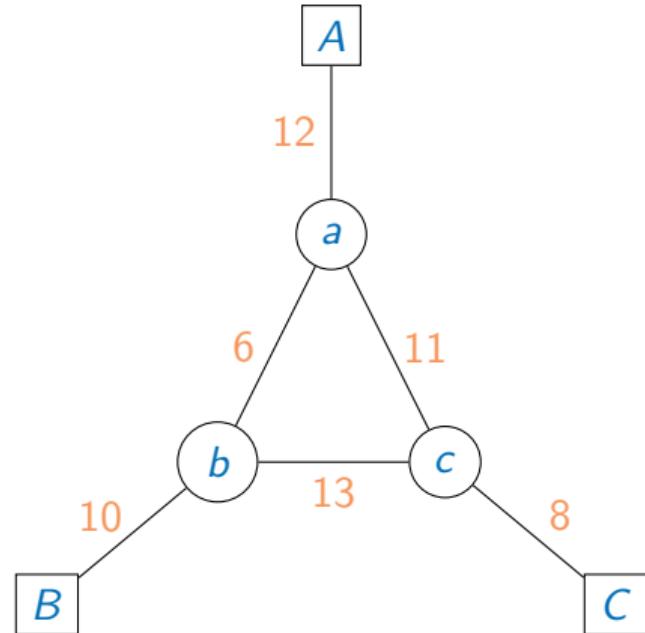


Solution

- Simplex yields
 - $x_{AB} = 0, y_{AB} = 7$
 - $x_{BC} = 1.5, y_{BC} = 1.5$
 - $x_{AC} = 0.5, y_{AC} = 4.5$
- Fractional solutions are OK
- All edges full capacity except $a-c$

Note

- Modelling strategy does not scale well
- One variable per path — number of paths is exponential, in general
- Will look at a better approach to analyze such network flows



Network Flows

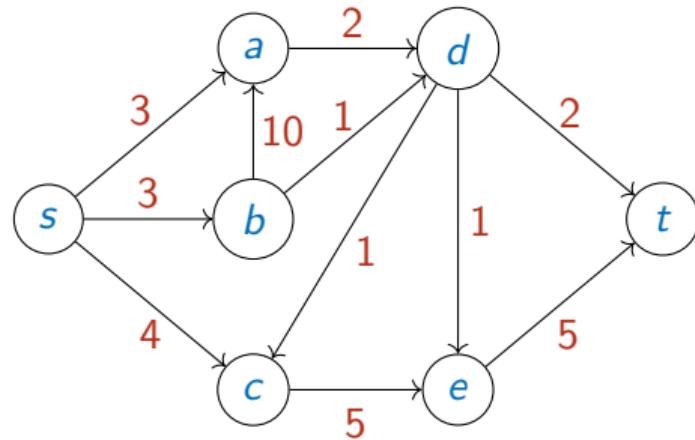
Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 11

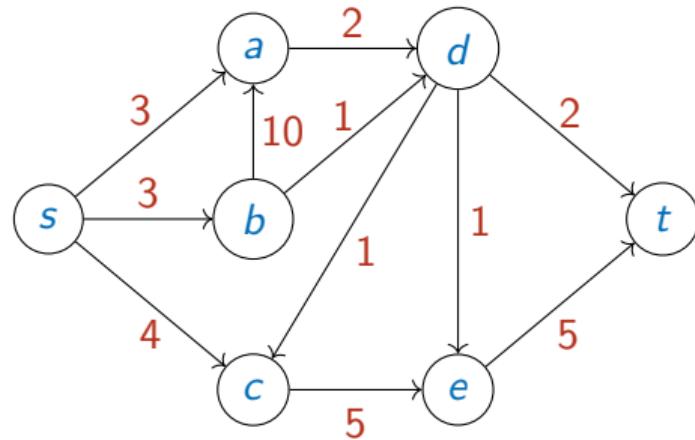
Oil network

- Network of pipelines



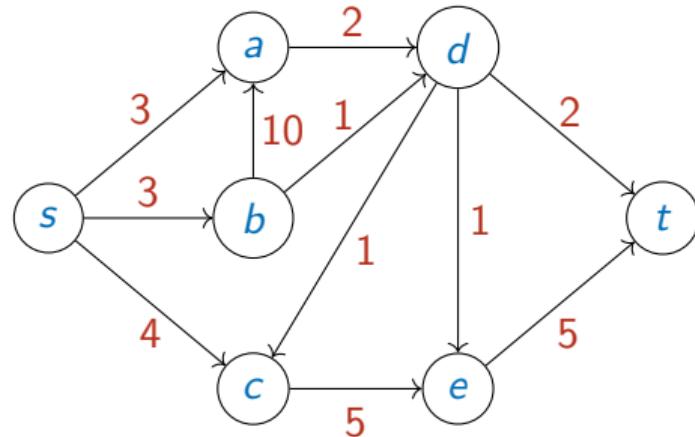
Oil network

- Network of pipelines
- Ship as much oil as possible from s to t



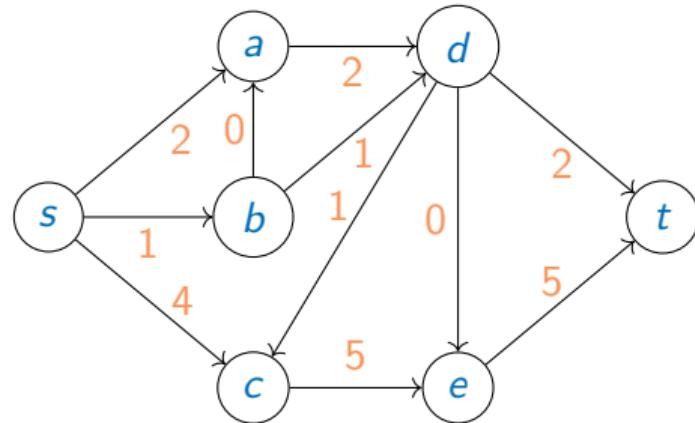
Oil network

- Network of pipelines
- Ship as much oil as possible from s to t
- No storage along the way



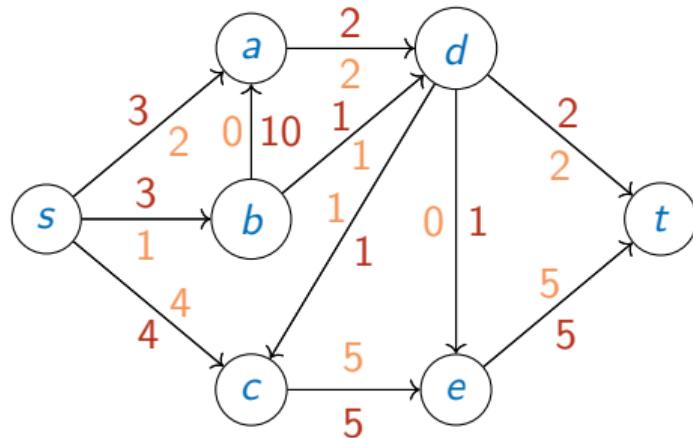
Oil network

- Network of pipelines
- Ship as much oil as possible from s to t
- No storage along the way
- A flow of 7 is possible



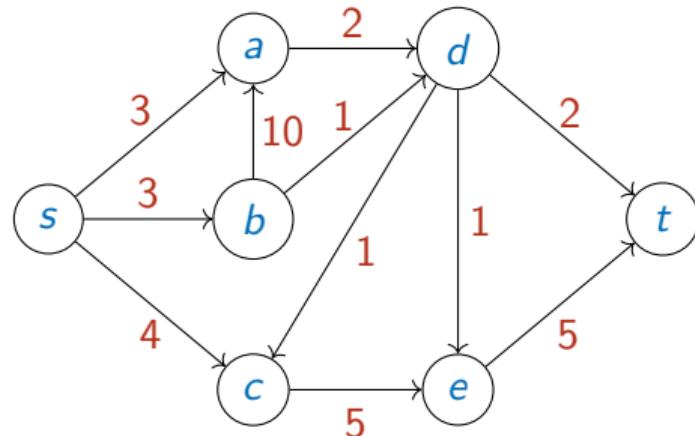
Oil network

- Network of pipelines
- Ship as much oil as possible from s to t
- No storage along the way
- A flow of 7 is possible
- Is this the maximum?



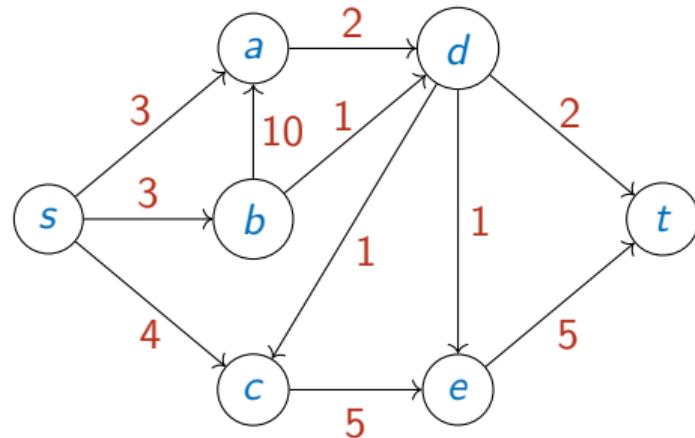
Oil network

- Network: graph $G = (V, E)$
- Special nodes: s (source), t (sink)



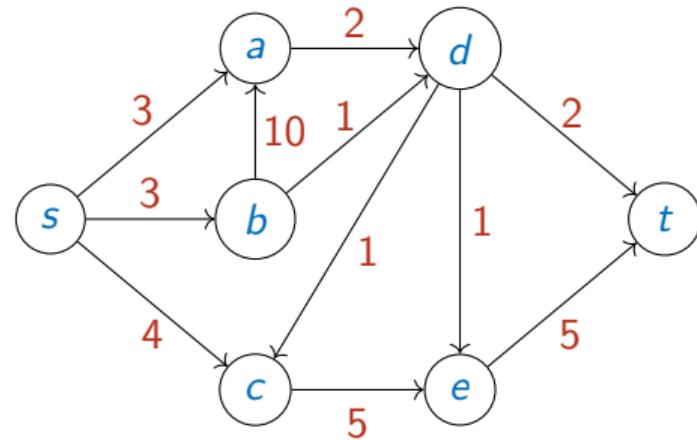
Oil network

- Network: graph $G = (V, E)$
- Special nodes: s (source), t (sink)
- Each edge e has capacity c_e



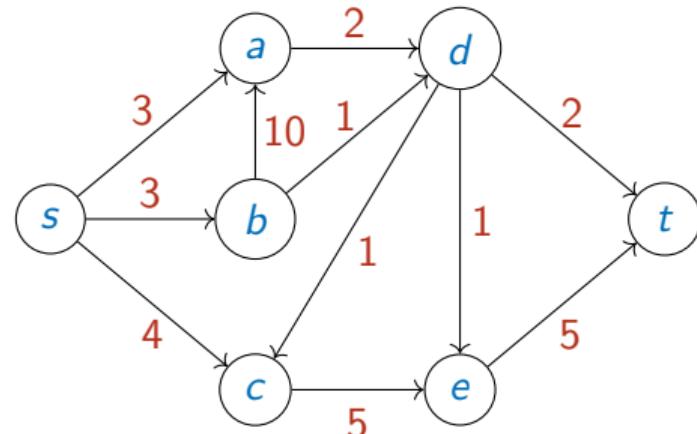
Oil network

- Network: graph $G = (V, E)$
- Special nodes: s (source), t (sink)
- Each edge e has capacity c_e
- Flow: f_e for each edge e
 - $f_e \leq c_e$
 - At each node, except s and t , sum of incoming flows equal sum of outgoing flows



Oil network

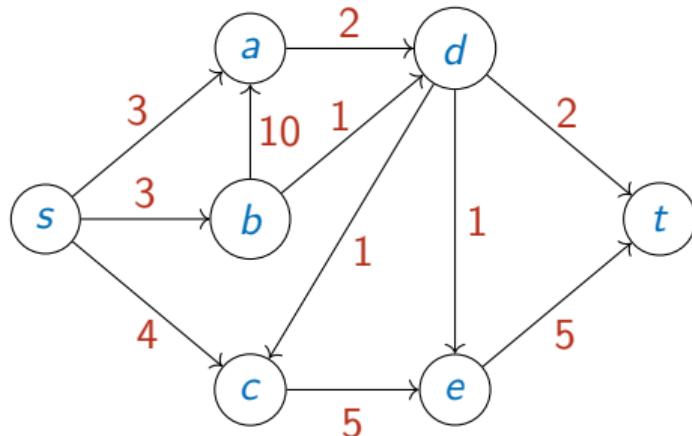
- Network: graph $G = (V, E)$
- Special nodes: s (source), t (sink)
- Each edge e has capacity c_e
- Flow: f_e for each edge e
 - $f_e \leq c_e$
 - At each node, except s and t , sum of incoming flows equal sum of outgoing flows
- Total volume of flow is sum of outgoing flow from s



LP formulation

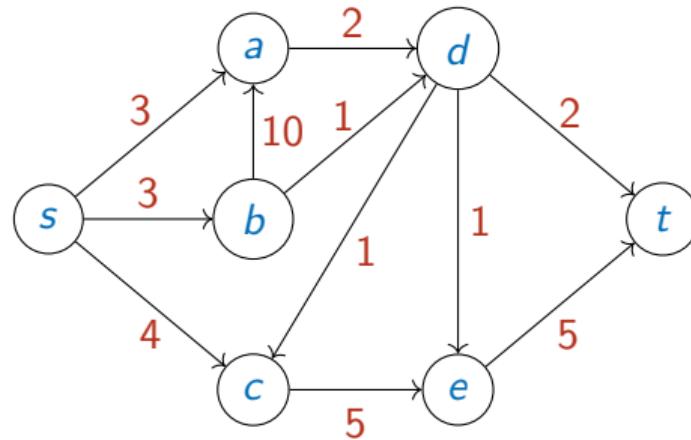
- Variable f_e for each edge e

- $f_{sa}, f_{bd}, f_{ce}, \dots$



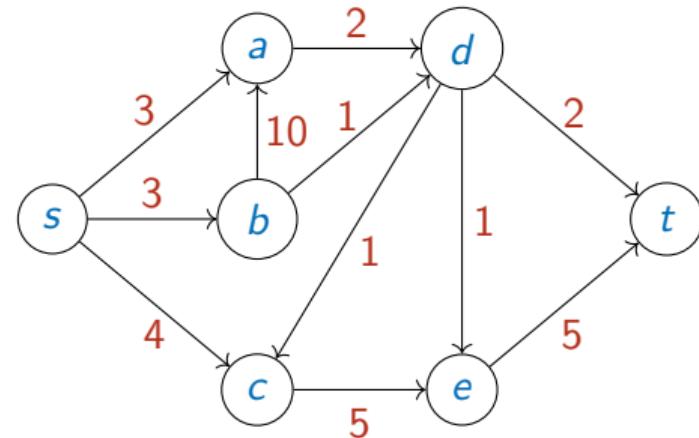
LP formulation

- Variable f_e for each edge e
 - $f_{sa}, f_{bd}, f_{ce}, \dots$
- Capacity constraints per edge
 - $f_{ba} \leq 10, \dots$



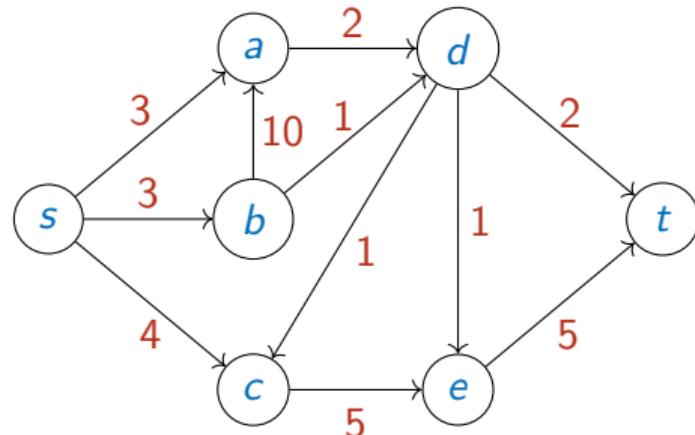
LP formulation

- Variable f_e for each edge e
 - $f_{sa}, f_{bd}, f_{ce}, \dots$
- Capacity constraints per edge
 - $f_{ba} \leq 10, \dots$
- Conservation of flow at each internal node
 - $f_{ad} + f_{bd} = f_{dc} + f_{de} + f_{dt}, \dots$



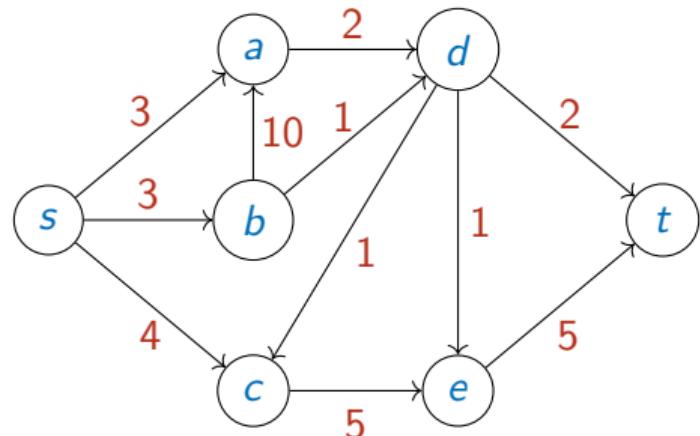
LP formulation

- Variable f_e for each edge e
 - $f_{sa}, f_{bd}, f_{ce}, \dots$
- Capacity constraints per edge
 - $f_{ba} \leq 10, \dots$
- Conservation of flow at each internal node
 - $f_{ad} + f_{bd} = f_{dc} + f_{de} + f_{dt}, \dots$
- Objective: maximize flow volume
 - Maximize $f_{sa} + f_{sb} + f_{sc}$



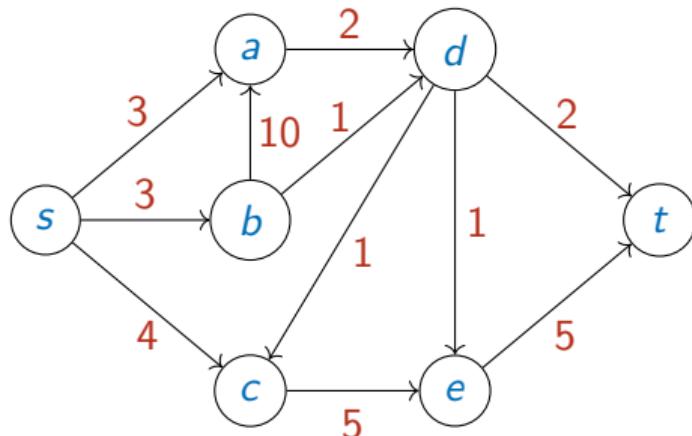
LP formulation

- Variable f_e for each edge e
 - $f_{sa}, f_{bd}, f_{ce}, \dots$
- Capacity constraints per edge
 - $f_{ba} \leq 10, \dots$
- Conservation of flow at each internal node
 - $f_{ad} + f_{bd} = f_{dc} + f_{de} + f_{dt}, \dots$
- Objective: maximize flow volume
 - Maximize $f_{sa} + f_{sb} + f_{sc}$
- Simplex explores vertices of feasible region to solve LP, find maximum flow



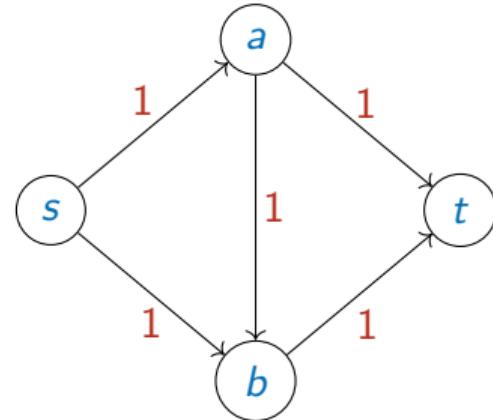
LP formulation

- Variable f_e for each edge e
 - $f_{sa}, f_{bd}, f_{ce}, \dots$
- Capacity constraints per edge
 - $f_{ba} \leq 10, \dots$
- Conservation of flow at each internal node
 - $f_{ad} + f_{bd} = f_{dc} + f_{de} + f_{dt}, \dots$
- Objective: maximize flow volume
 - Maximize $f_{sa} + f_{sb} + f_{sc}$
- Simplex explores vertices of feasible region to solve LP, find maximum flow
- Moving from vertex to vertex gives a more direct algorithm for maximum flow



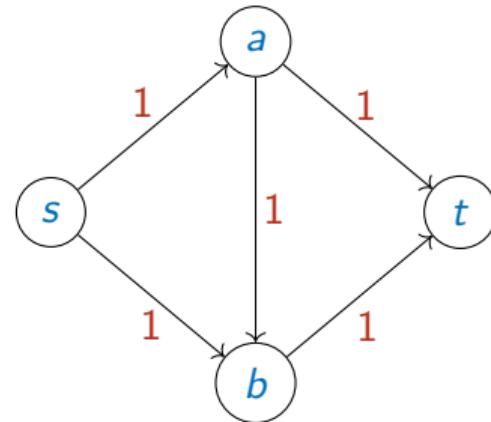
Ford-Fulkerson algorithm

- Start with zero flow



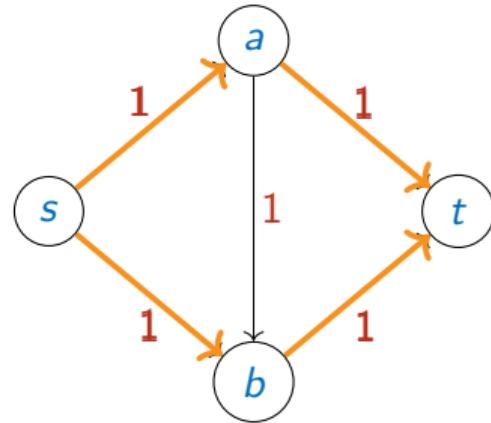
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible



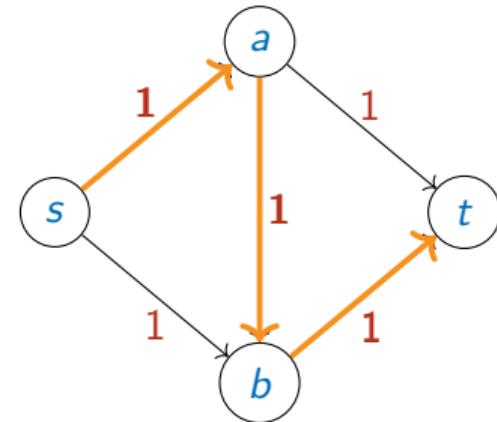
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Network on the right has max flow 2



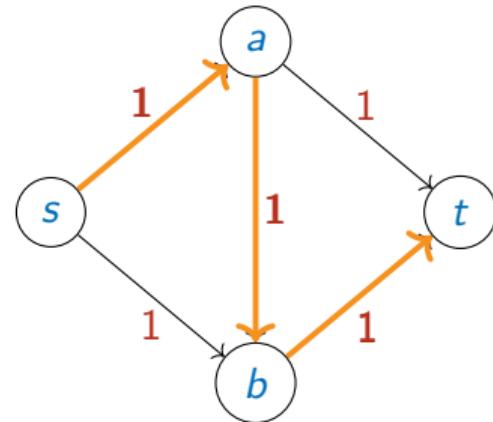
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Network on the right has max flow 2
- What if one chooses a bad flow to begin with?



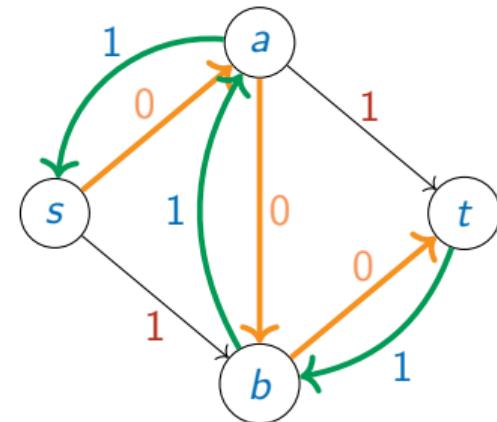
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Network on the right has max flow 2
- What if one chooses a bad flow to begin with?
- Add reverse edges to undo flow from previous steps



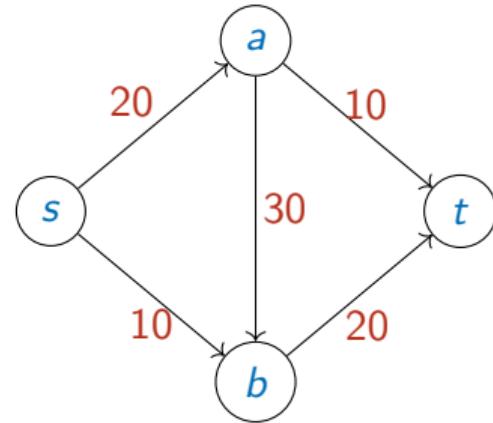
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Network on the right has max flow 2
- What if one chooses a bad flow to begin with?
- Add reverse edges to undo flow from previous steps
- **Residual graph:** for each edge e with capacity c_e and current flow f_e
 - Reduce capacity to $c_e - f_e$
 - Add reverse edge with capacity f_e



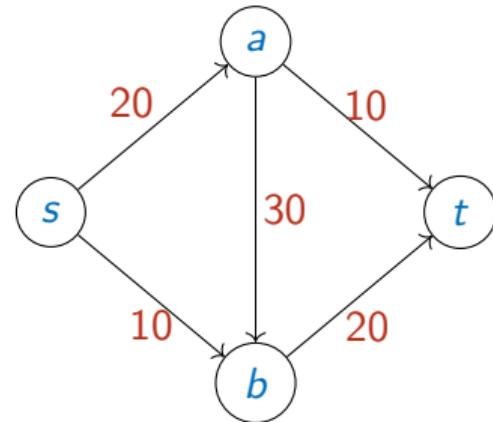
Ford-Fulkerson algorithm

- Start with zero flow



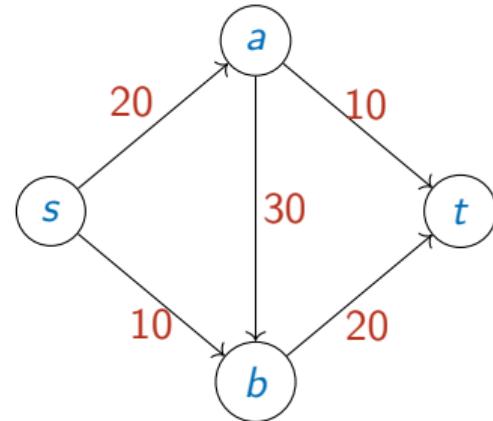
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible



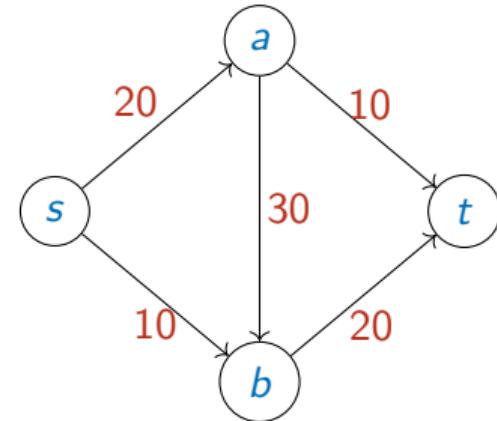
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Build residual graph



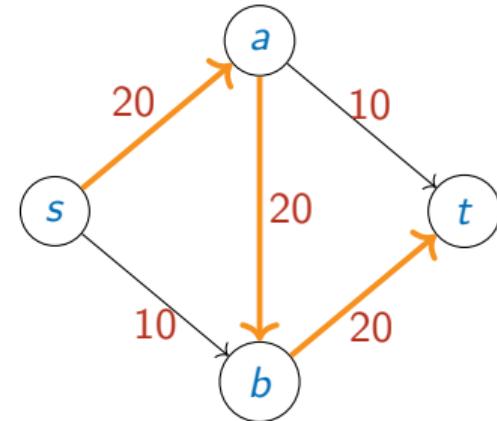
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Build residual graph
- Repeat the previous two steps till there is no feasible flow from s to t



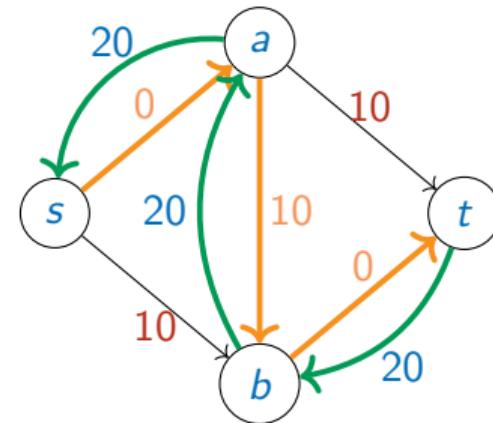
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Build residual graph
- Repeat the previous two steps till there is no feasible flow from s to t
- Flow 20, $s - a - b - t$,



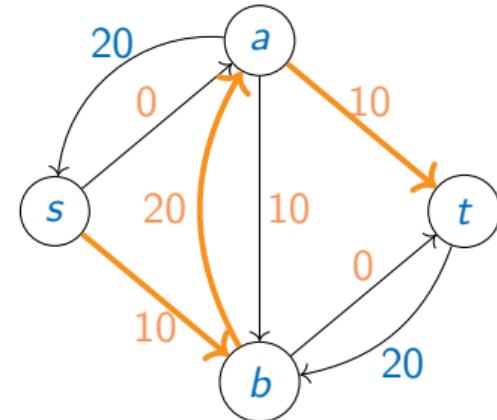
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Build residual graph
- Repeat the previous two steps till there is no feasible flow from s to t
- Flow 20, $s - a - b - t$, build residual graph



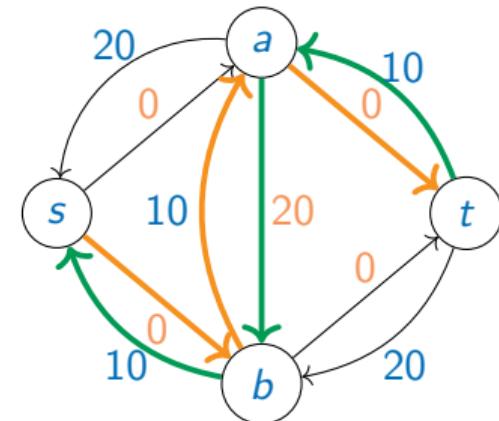
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Build residual graph
- Repeat the previous two steps till there is no feasible flow from s to t
- Flow 20, $s - a - b - t$, build residual graph
- Add flow 10, $s - b - a - t$,



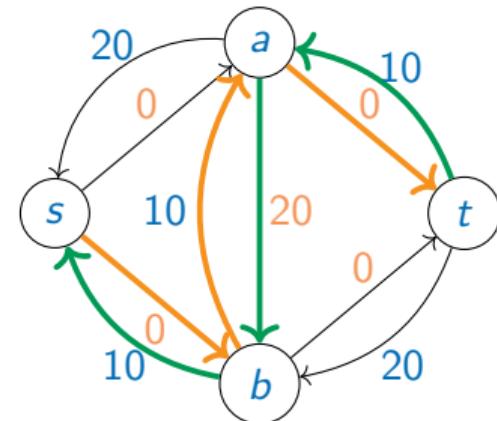
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Build residual graph
- Repeat the previous two steps till there is no feasible flow from s to t
- Flow 20, $s - a - b - t$, build residual graph
- Add flow 10, $s - b - a - t$, build residual graph



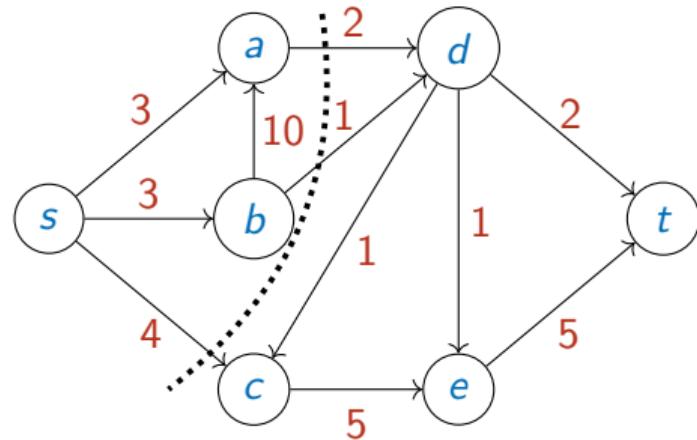
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Build residual graph
- Repeat the previous two steps till there is no feasible flow from s to t
- Flow 20, $s - a - b - t$, build residual graph
- Add flow 10, $s - b - a - t$, build residual graph
- No more feasible paths from s to t



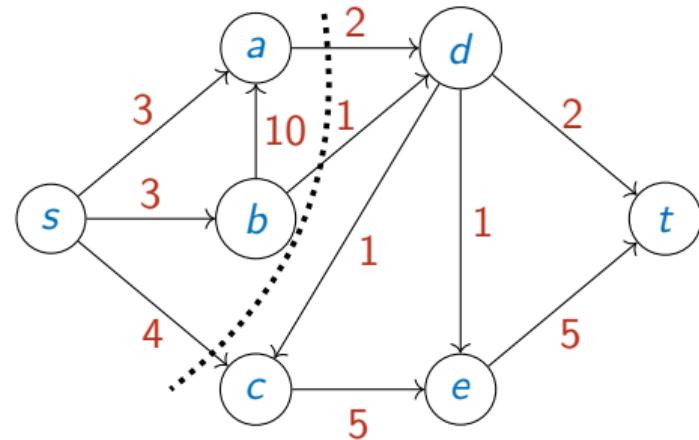
Certificate of optimality

- Edges $\{ad, bd, sc\}$ disconnect s and t
 - (s, t) -cut



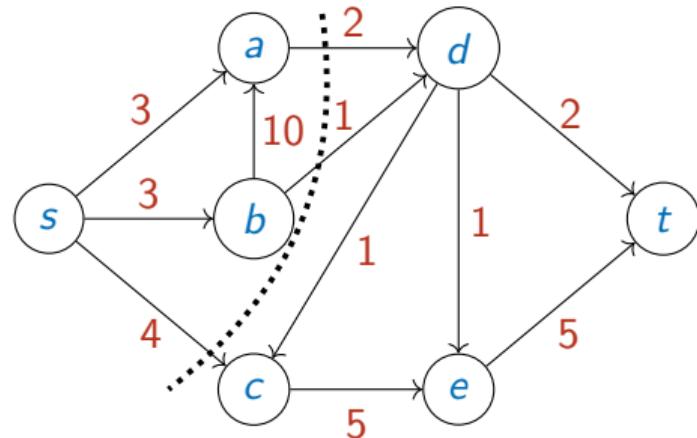
Certificate of optimality

- Edges $\{ad, bd, sc\}$ disconnect s and t
 - (s, t) -cut
- Flow from s to t must go through this cut



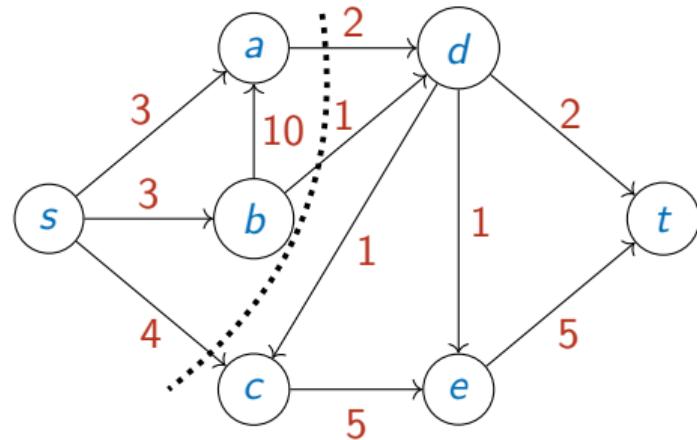
Certificate of optimality

- Edges $\{ad, bd, sc\}$ disconnect s and t
 - (s, t) -cut
- Flow from s to t must go through this cut
- Cannot exceed cut capacity, 7



Certificate of optimality

- Edges $\{ad, bd, sc\}$ disconnect s and t
 - (s, t) -cut
- Flow from s to t must go through this cut
- Cannot exceed cut capacity, 7
- Max flow cannot exceed capacity of min cut

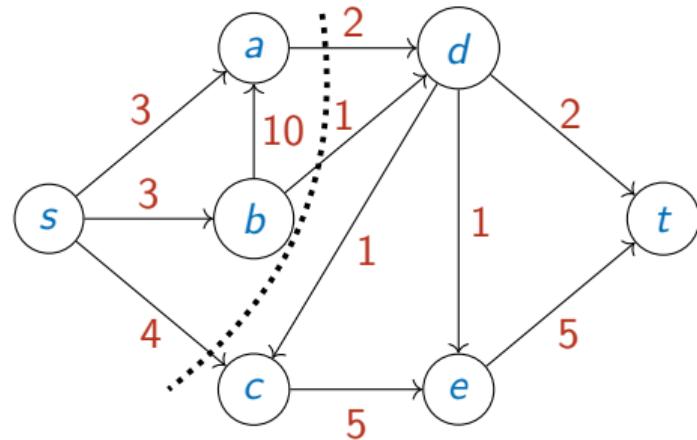


Certificate of optimality

- Edges $\{ad, bd, sc\}$ disconnect s and t
 - (s, t) -cut
- Flow from s to t must go through this cut
- Cannot exceed cut capacity, 7
- Max flow cannot exceed capacity of min cut

Max flow-min cut theorem

- In fact, max flow is always equal to min cut

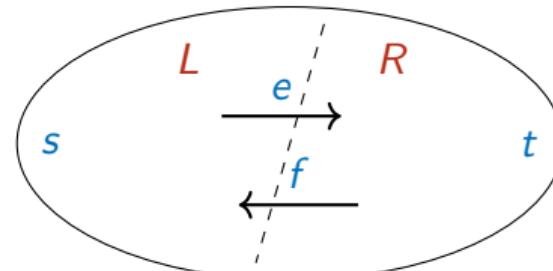
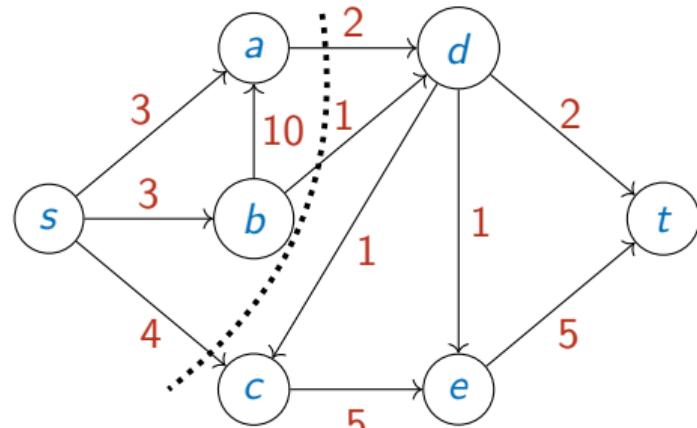


Certificate of optimality

- Edges $\{ad, bd, sc\}$ disconnect s and t
 - (s, t) -cut
- Flow from s to t must go through this cut
- Cannot exceed cut capacity, 7
- Max flow cannot exceed capacity of min cut

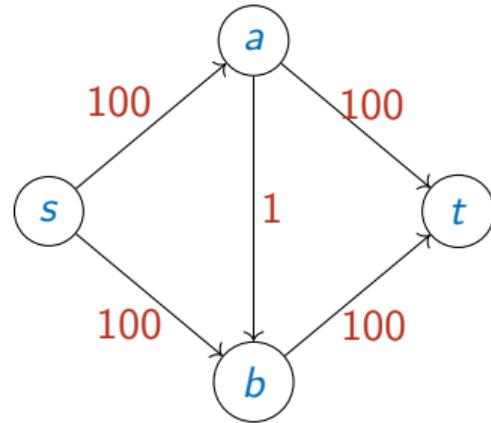
Max flow-min cut theorem

- In fact, max flow is always equal to min cut
- At max flow, no path from s to t in residual graph
 - s can reach L , R can reach t
 - Any edge from L to R must be at full capacity
 - Any edge from R to L must be at zero capacity



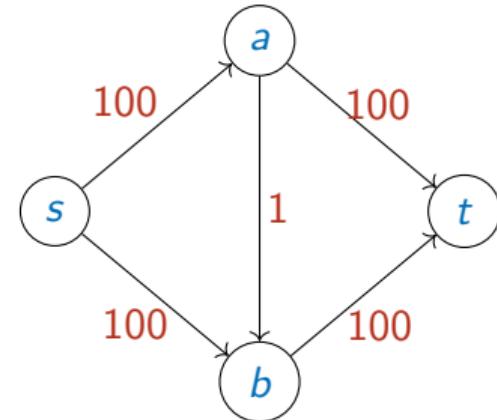
Ford-Fulkerson algorithm

- Choose augmenting paths wisely



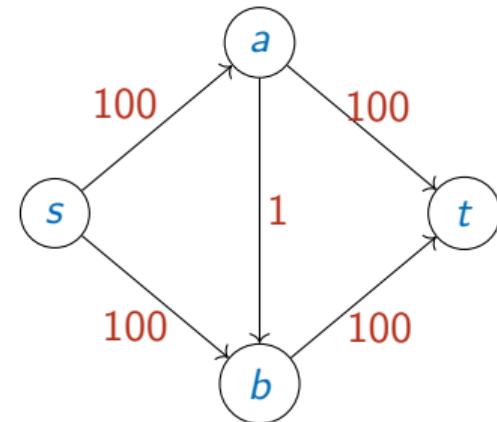
Ford-Fulkerson algorithm

- Choose augmenting paths wisely
- If we keep going through the middle edge, 200 iterations to find the max flow
 - Ford-Fulkerson can take time proportional to max capacity



Ford-Fulkerson algorithm

- Choose augmenting paths wisely
- If we keep going through the middle edge, 200 iterations to find the max flow
 - Ford-Fulkerson can take time proportional to max capacity
- Use BFS to find augmenting path with fewest edges
- Iterations bounded by $|V| \times |E|$, regardless of capacities



Reductions

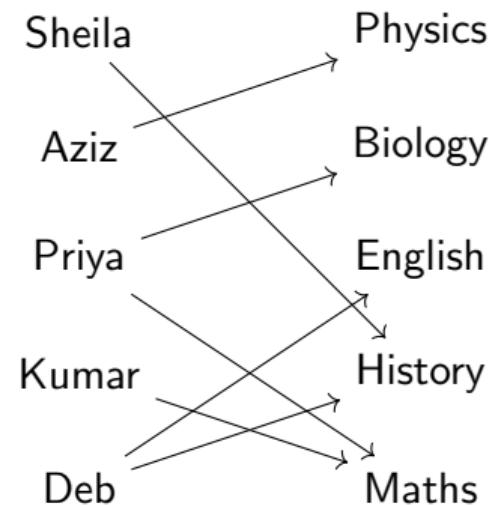
Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python
Week 11

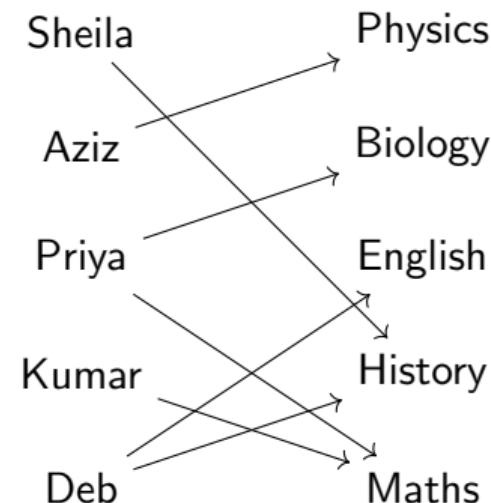
Bipartite matching

- Each instructor is willing to teach a set of courses



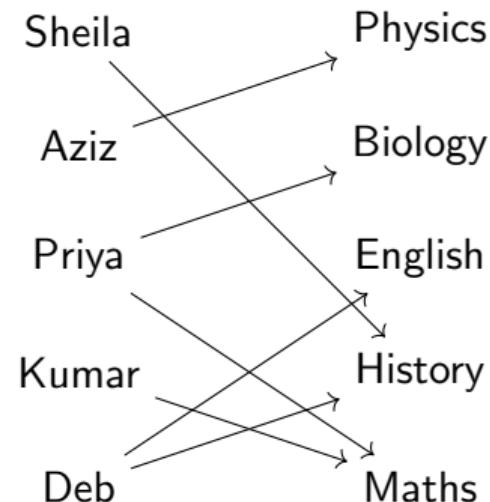
Bipartite matching

- Each instructor is willing to teach a set of courses
- Find an allocation so that
 - Each course is taught by a single instructor
 - Each instructor teaches only one course, which he/she is willing to teach



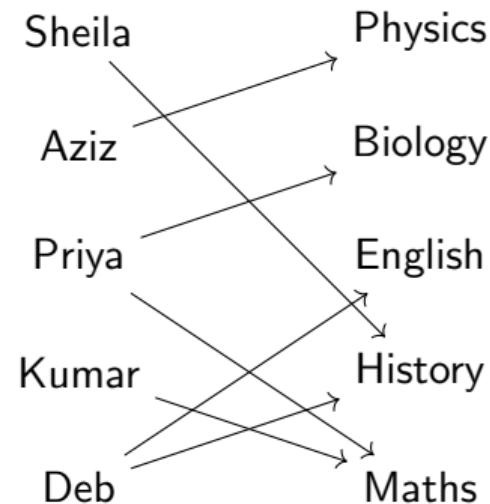
Bipartite matching

- V partitioned into V_0, V_1



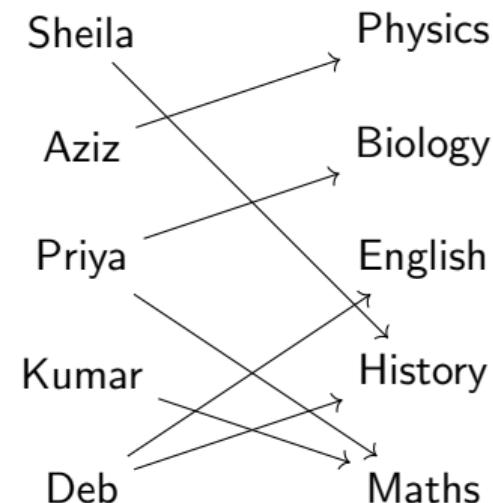
Bipartite matching

- V partitioned into V_0, V_1
- All edges from V_0 to V_1



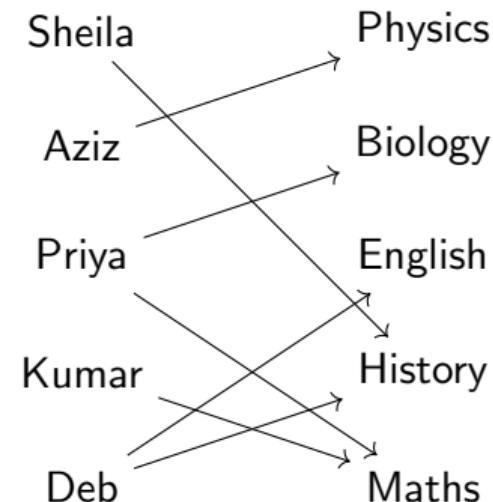
Bipartite matching

- V partitioned into V_0, V_1
- All edges from V_0 to V_1
- **Matching:** subset of edges so that no two of them share an endpoint



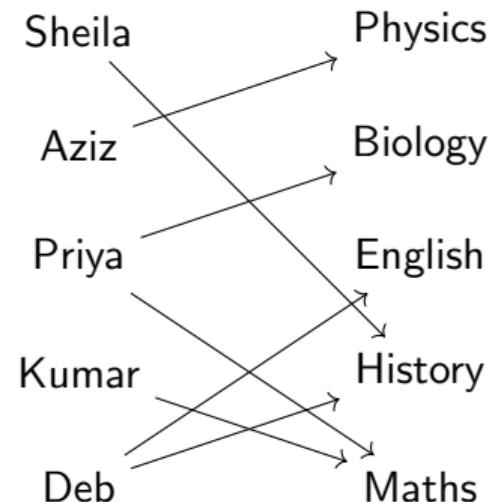
Bipartite matching

- V partitioned into V_0, V_1
- All edges from V_0 to V_1
- **Matching:** subset of edges so that no two of them share an endpoint
- Find largest matching



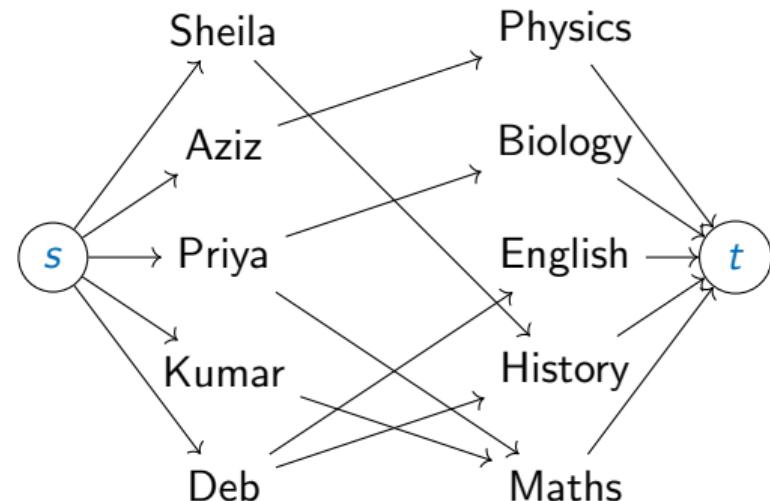
Bipartite matching

- V partitioned into V_0, V_1
- All edges from V_0 to V_1
- **Matching**: subset of edges so that no two of them share an endpoint
- Find largest matching
- If possible, a **perfect** matching, all nodes covered



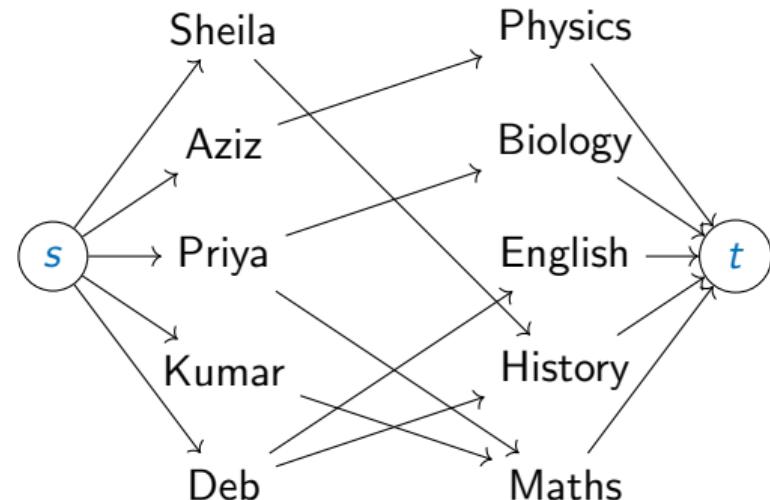
Bipartite matching

- V partitioned into V_0, V_1
- All edges from V_0 to V_1
- Matching: subset of edges so that no two of them share an endpoint
- Find largest matching
- If possible, a perfect matching, all nodes covered
- Add a source and a sink
 - All edge capacities 1



Bipartite matching

- V partitioned into V_0, V_1
- All edges from V_0 to V_1
- Matching: subset of edges so that no two of them share an endpoint
- Find largest matching
- If possible, a perfect matching, all nodes covered
- Add a source and a sink
 - All edge capacities 1
- Find a maximum flow from s to t !



Reductions

- We want to solve problem A

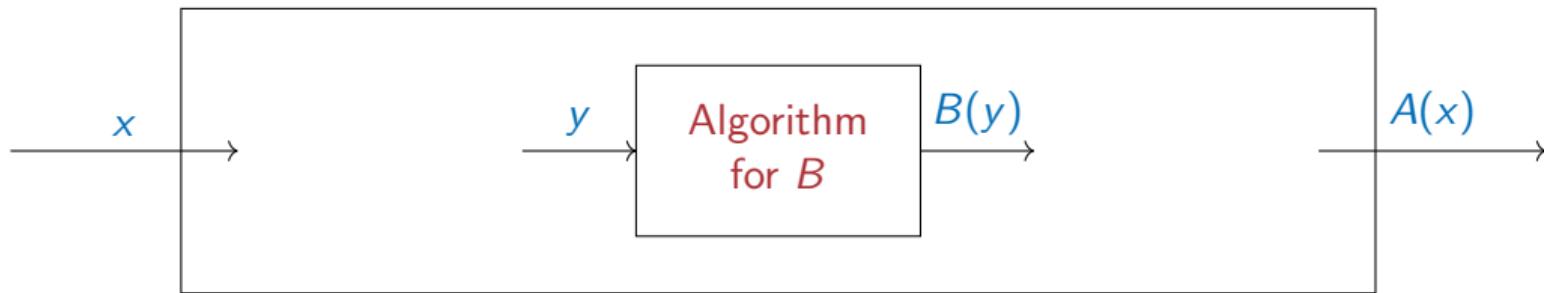
Algorithm for A



Reductions

- We want to solve problem A
- We know how to solve problem B

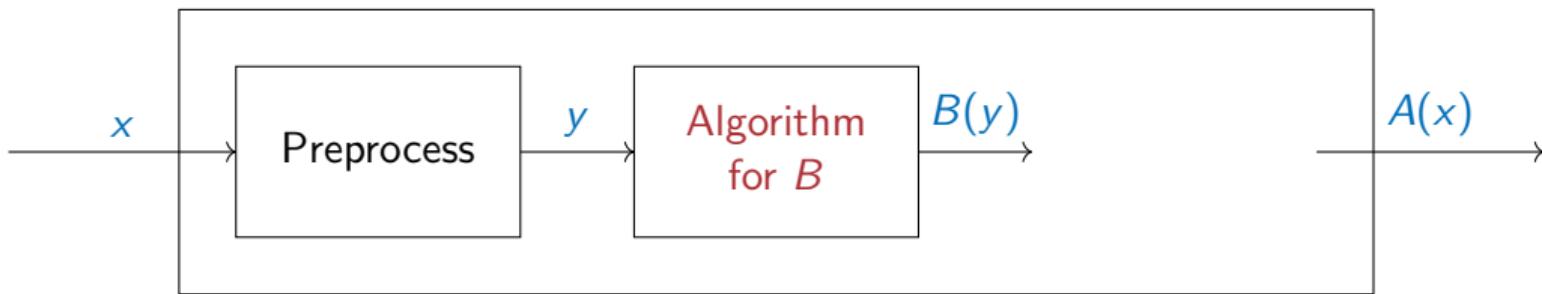
Algorithm for A



Reductions

- We want to solve problem A
- We know how to solve problem B
- Convert input for A into input for B

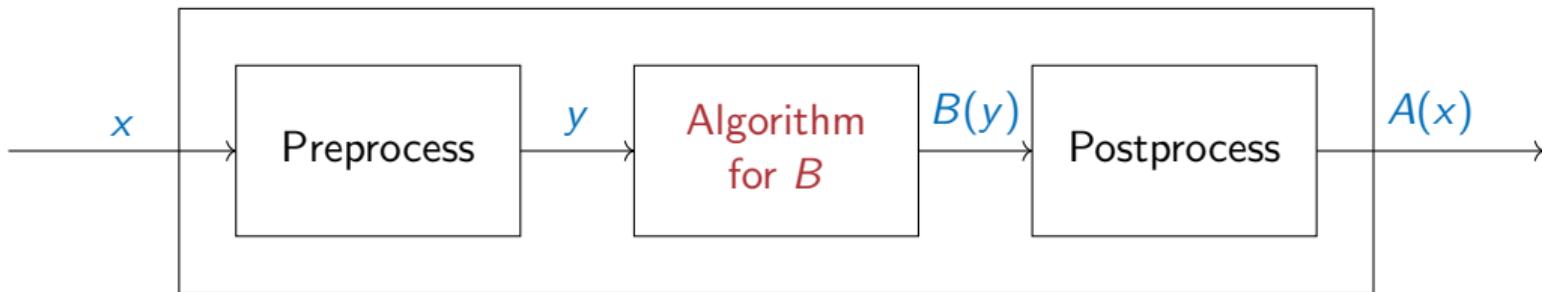
Algorithm for A



Reductions

- We want to solve problem A
- We know how to solve problem B
- Convert input for A into input for B
- Interpret output of B as output of A

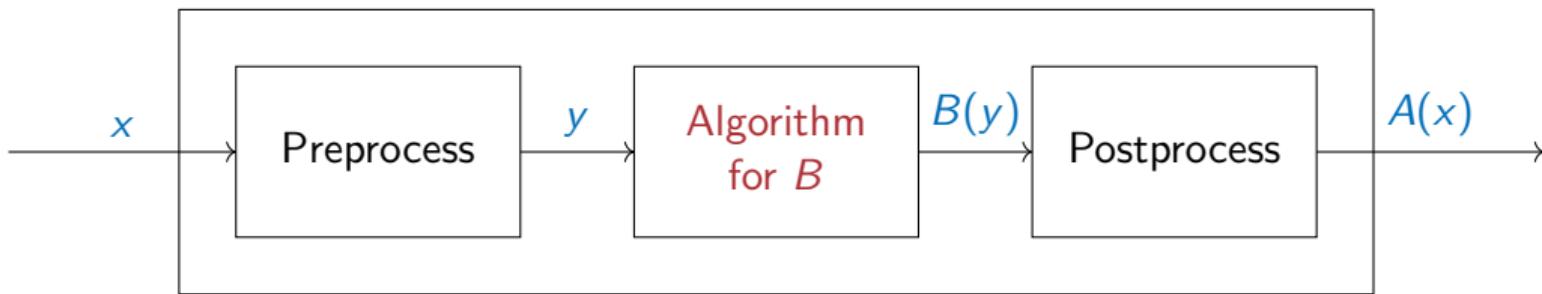
Algorithm for A



Reductions

- A reduces to B

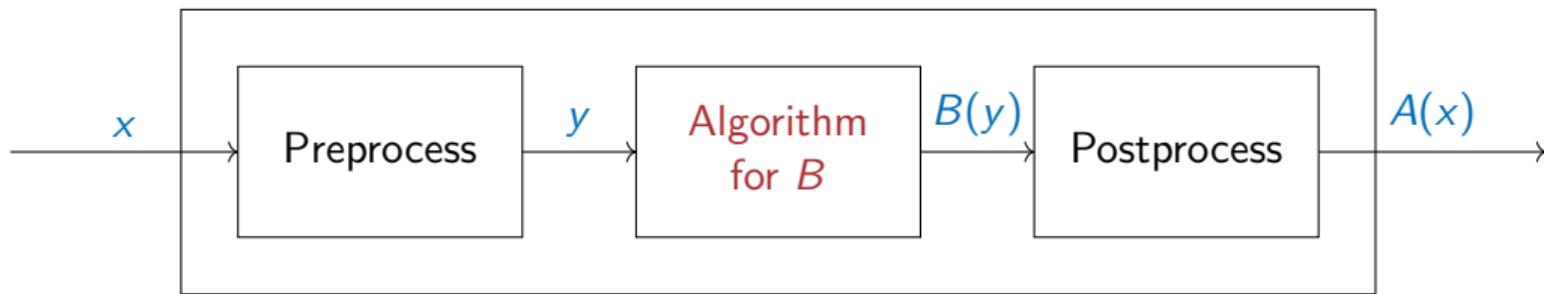
Algorithm for A



Reductions

- A reduces to B
- Can transfer efficient solution from B to A

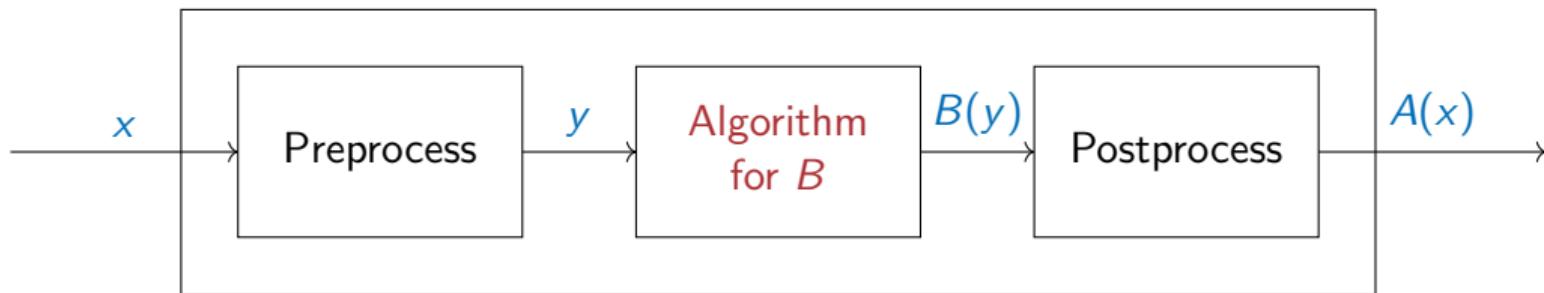
Algorithm for A



Reductions

- A reduces to B
- Can transfer efficient solution from B to A
- But preprocessing and postprocessing must also be efficient!

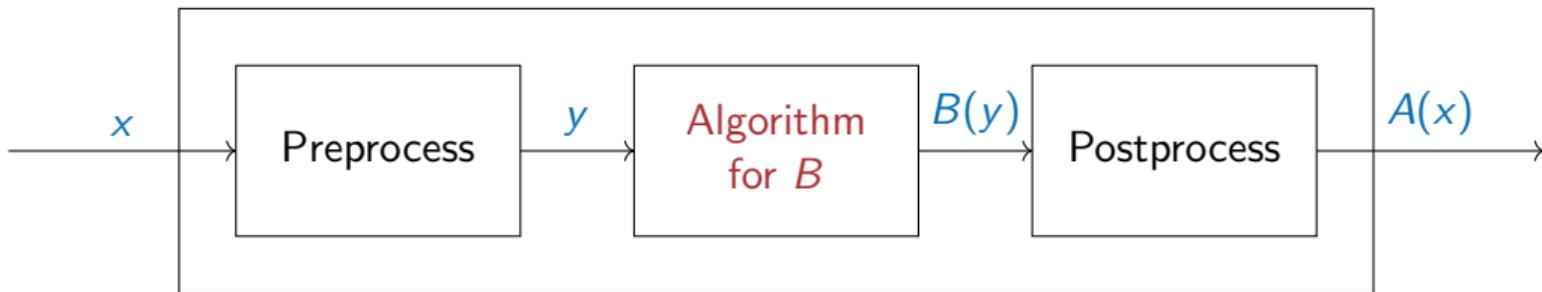
Algorithm for A



Reductions

- A reduces to B
- Can transfer efficient solution from B to A
- But preprocessing and postprocessing must also be efficient!
- Typically, both should be polynomial time

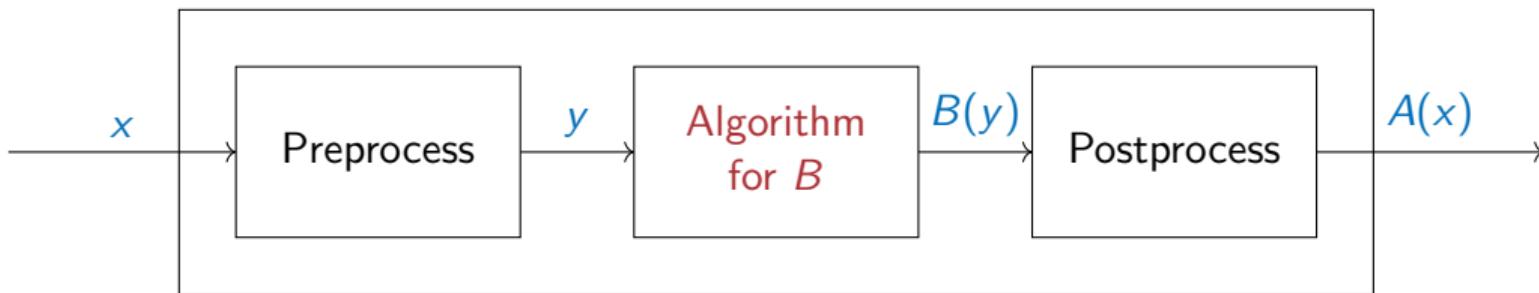
Algorithm for A



Reductions

- Bipartite matching reduces to max flow

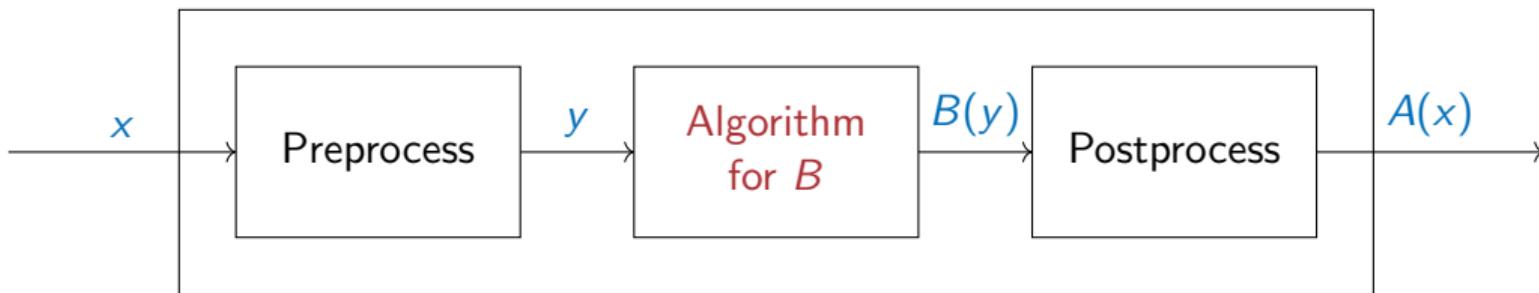
Algorithm for A



Reductions

- Bipartite matching reduces to max flow
- Max flow reduces to LP

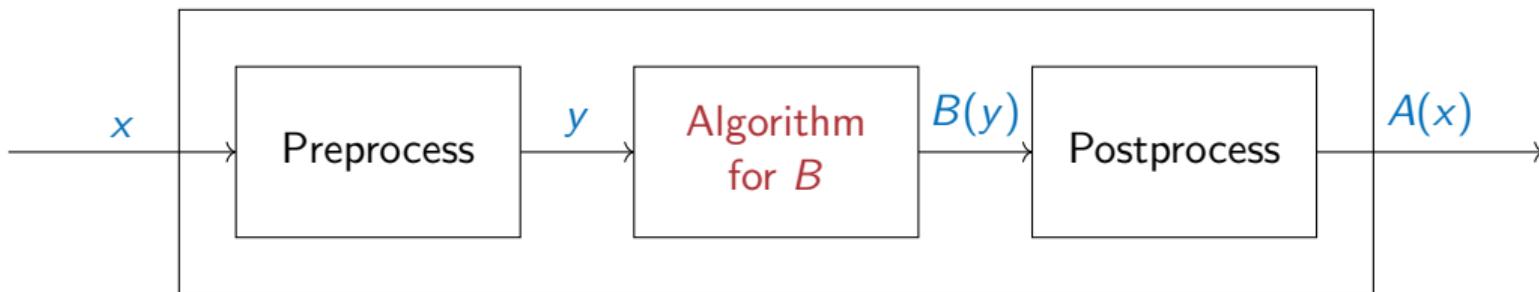
Algorithm for A



Reductions

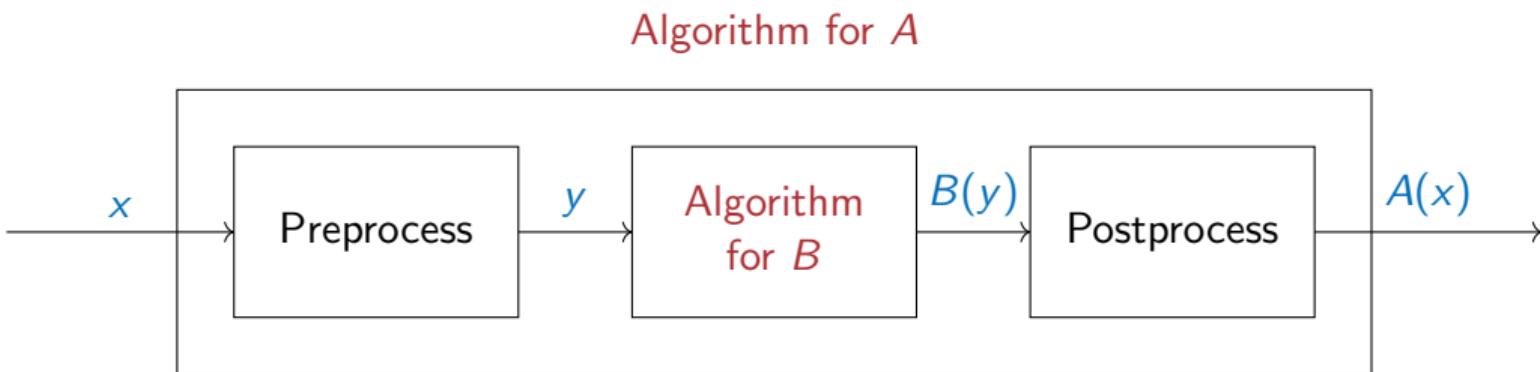
- Bipartite matching reduces to max flow
- Max flow reduces to LP
- Number of variables, constraints is linear in the size of the graph

Algorithm for A



Reductions

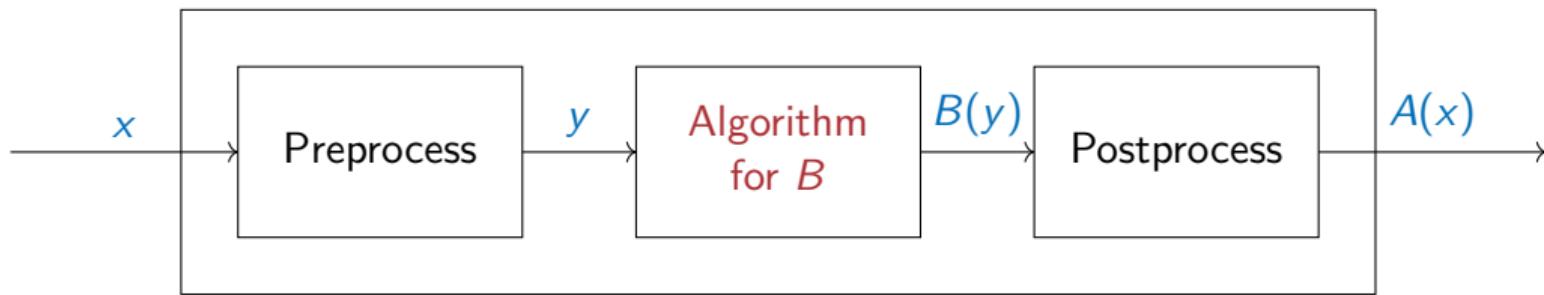
- Reverse interpretation is also useful



Reductions

- Reverse interpretation is also useful
- If A is known to be intractable and A reduces to B , then B must also be intractable

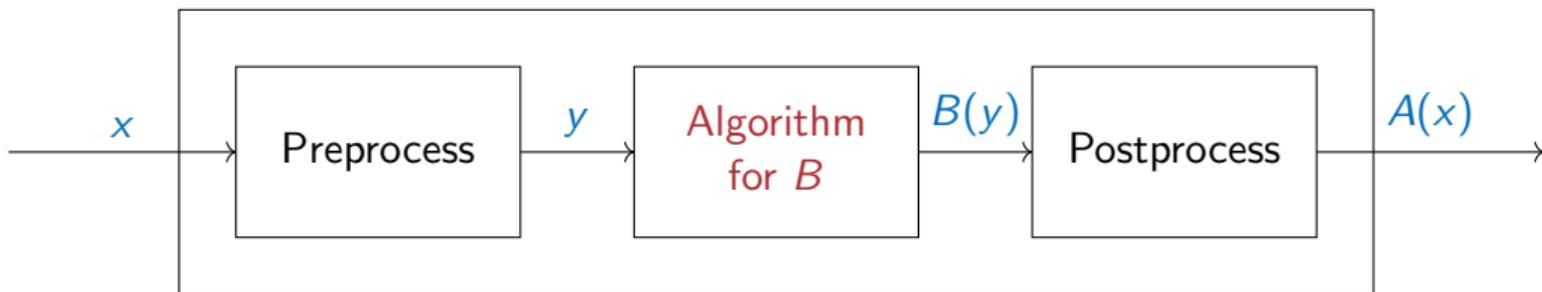
Algorithm for A



Reductions

- Reverse interpretation is also useful
- If A is known to be intractable and A reduces to B , then B must also be intractable
- Otherwise, efficient solution for B will yield efficient solution for A

Algorithm for A



Big hammers

- LP and network flows are powerful tools

Big hammers

- LP and network flows are powerful tools
- Many algorithmic problems can be reduced to them

Big hammers

- LP and network flows are powerful tools
- Many algorithmic problems can be reduced to them
- Efficient, off-the-shelf implementations are available

Big hammers

- LP and network flows are powerful tools
- Many algorithmic problems can be reduced to them
- Efficient, off-the-shelf implementations are available
- Useful to understand what can (and cannot) be modelled in terms of LP and flows

Intractability: Checking Algorithms

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 11

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, . . . have polynomial time algorithms

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, ... have polynomial time algorithms
- Search space for solutions is exponential
 - All possible paths, all possible spanning trees, all possible subsets of edges, ...

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, ... have polynomial time algorithms
- Search space for solutions is exponential
 - All possible paths, all possible spanning trees, all possible subsets of edges, ...
- Brute force: scan exponential possibilities and choose the best

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, ... have polynomial time algorithms
- Search space for solutions is exponential
 - All possible paths, all possible spanning trees, all possible subsets of edges, ...
- Brute force: scan exponential possibilities and choose the best
- Do all problems admit such efficient solutions?

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, ... have polynomial time algorithms
- Search space for solutions is exponential
 - All possible paths, all possible spanning trees, all possible subsets of edges, ...
- Brute force: scan exponential possibilities and choose the best
- Do all problems admit such efficient solutions?
 - Unfortunately not

Efficient algorithms

- Shortest path, minimum cost spanning tree, maximum flow, ... have polynomial time algorithms
- Search space for solutions is exponential
 - All possible paths, all possible spanning trees, all possible subsets of edges, ...
- Brute force: scan exponential possibilities and choose the best
- Do all problems admit such efficient solutions?
 - Unfortunately not
 - For a large class of “natural” problems, no shortcut is known to exist

Generating vs checking

- A teacher assigns homework:
 - Factorize a large number that is the product of two primes

Generating vs checking

- A teacher assigns homework:
 - Factorize a large number that is the product of two primes
- Student: Given N , find p, q such that
$$pq = N$$
 - Generate a solution

Generating vs checking

- A teacher assigns homework:
 - Factorize a large number that is the product of two primes
- Student: Given N , find p, q such that
$$pq = N$$
 - Generate a solution
- Teacher: Given a student's solution p, q ,
verify that $pq = N$
 - Check a solution

Generating vs checking

- A teacher assigns homework:

- Factorize a large number that is the product of two primes

- **Student:** Given N , find p, q such that

$$pq = N$$

- Generate a solution

- **Teacher:** Given a student's solution p, q , verify that $pq = N$

- Check a solution

Checking algorithms

- Checking algorithm C for problem P

Generating vs checking

- A teacher assigns homework:
 - Factorize a large number that is the product of two primes
- **Student:** Given N , find p, q such that $pq = N$
 - Generate a solution
- **Teacher:** Given a student's solution p, q , verify that $pq = N$
 - Check a solution

Checking algorithms

- Checking algorithm C for problem P
- Takes an input instance I for P and a solution "certificate" S for I

Generating vs checking

- A teacher assigns homework:
 - Factorize a large number that is the product of two primes
- **Student:** Given N , find p, q such that $pq = N$
 - Generate a solution
- **Teacher:** Given a student's solution p, q , verify that $pq = N$
 - Check a solution

Checking algorithms

- Checking algorithm C for problem P
- Takes an input instance I for P and a solution "certificate" S for I
- C outputs yes if S represents a valid solution for I , no otherwise

Generating vs checking

- A teacher assigns homework:
 - Factorize a large number that is the product of two primes
- **Student:** Given N , find p, q such that $pq = N$
 - Generate a solution
- **Teacher:** Given a student's solution p, q , verify that $pq = N$
 - Check a solution

Checking algorithms

- Checking algorithm C for problem P
- Takes an input instance I for P and a solution "certificate" S for I
- C outputs yes if S represents a valid solution for I , no otherwise
- For factorization
 - I is N
 - S is $\{p, q\}$
 - C involves verifying that $pq = N$

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Boolean operators
 - $\neg x_j$ — negation of x_j
 - $x_i \vee x_j$ — x_i or x_j
 - $x_i \wedge x_j$ — x_i and x_j

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Boolean operators
 - $\neg x_j$ — negation of x_j
 - $x_i \vee x_j$ — x_i or x_j
 - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form
$$(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_n)$$
 - Disjunction of literals (variables, negated variables)

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Boolean operators
 - $\neg x_j$ — negation of x_j
 - $x_i \vee x_j$ — x_i or x_j
 - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form
$$(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_n)$$
 - Disjunction of literals (variables, negated variables)
- Formula — conjunction of clauses
$$C_1 \wedge C_2 \wedge \dots \wedge C_k$$

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Boolean operators
 - $\neg x_j$ — negation of x_j
 - $x_i \vee x_j$ — x_i or x_j
 - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form
$$(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_n)$$
 - Disjunction of literals (variables, negated variables)
- Formula — conjunction of clauses
$$C_1 \wedge C_2 \wedge \dots \wedge C_k$$
- Assign suitable values {True, False} to x_1, x_2, x_3, \dots so that the formula evaluates to True
$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Boolean operators
 - $\neg x_j$ — negation of x_j
 - $x_i \vee x_j$ — x_i or x_j
 - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_n)$
 - Disjunction of literals (variables, negated variables)
- Formula — conjunction of clauses $C_1 \wedge C_2 \wedge \dots \wedge C_k$
- Assign suitable values {True, False} to x_1, x_2, x_3, \dots so that the formula evaluates to True
$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$
 - $x_1 = \text{True}$, $x_2 = \text{True}$, $x_3 = \text{False}$ makes this formula evaluate to True

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Boolean operators
 - $\neg x_j$ — negation of x_j
 - $x_i \vee x_j$ — x_i or x_j
 - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_n)$
 - Disjunction of literals (variables, negated variables)
- Formula — conjunction of clauses $C_1 \wedge C_2 \wedge \dots \wedge C_k$
- Assign suitable values {True, False} to x_1, x_2, x_3, \dots so that the formula evaluates to True
$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$
 - $x_1 = \text{True}$, $x_2 = \text{True}$, $x_3 = \text{False}$ makes this formula evaluate to True
 - Add a clause
$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_3 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Boolean operators
 - $\neg x_j$ — negation of x_j
 - $x_i \vee x_j$ — x_i or x_j
 - $x_i \wedge x_j$ — x_i and x_j
- Clause — formula C of the form $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_n)$
 - Disjunction of literals (variables, negated variables)
- Formula — conjunction of clauses $C_1 \wedge C_2 \wedge \dots \wedge C_k$
- Assign suitable values {True, False} to x_1, x_2, x_3, \dots so that the formula evaluates to True
$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$
 - $x_1 = \text{True}$, $x_2 = \text{True}$, $x_3 = \text{False}$ makes this formula evaluate to True
 - Add a clause
$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\textcolor{red}{x_3 \vee \neg x_1}) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$
 - Now there is no satisfying assignment

- Generating a solution

- Generating a solution
 - Try each possible assignment V to x_1, x_2, \dots, x_n

- Generating a solution

- Try each possible assignment V to x_1, x_2, \dots, x_n
- n variables — 2^n possible assignments

- Generating a solution

- Try each possible assignment V to x_1, x_2, \dots, x_n
- n variables — 2^n possible assignments
- Is there a better algorithm? Not known

- Generating a solution

- Try each possible assignment V to x_1, x_2, \dots, x_n
- n variables — 2^n possible assignments
- Is there a better algorithm? Not known

- Checking a solution

- Given formula F and valuation $V(x)$ for each x , substitute into formula and evaluate

- Generating a solution
 - Try each possible assignment V to x_1, x_2, \dots, x_n
 - n variables — 2^n possible assignments
 - Is there a better algorithm? Not known
- Checking a solution
 - Given formula F and valuation $V(x)$ for each x , substitute into formula and evaluate

- Generating a solution
 - Try each possible assignment V to x_1, x_2, \dots, x_n
 - n variables — 2^n possible assignments
 - Is there a better algorithm? Not known
- Checking a solution
 - Given formula F and valuation $V(x)$ for each x , substitute into formula and evaluate
- Input format is important
 - Suppose a clause is a conjunction of literals ...
$$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \dots \wedge x_k)$$

- Generating a solution
 - Try each possible assignment V to x_1, x_2, \dots, x_n
 - n variables — 2^n possible assignments
 - Is there a better algorithm? Not known
- Checking a solution
 - Given formula F and valuation $V(x)$ for each x , substitute into formula and evaluate
- Input format is important
 - Suppose a clause is a conjunction of literals ...
$$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \dots \wedge x_k)$$
 - ... and a formula is a disjunction of clauses $C_1 \vee C_2 \vee \dots \vee C_m$

- Generating a solution
 - Try each possible assignment V to x_1, x_2, \dots, x_n
 - n variables — 2^n possible assignments
 - Is there a better algorithm? Not known
- Checking a solution
 - Given formula F and valuation $V(x)$ for each x , substitute into formula and evaluate
- Input format is important
 - Suppose a clause is a conjunction of literals ...
$$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \dots \wedge x_k)$$
 - ... and a formula is a disjunction of clauses $C_1 \vee C_2 \vee \dots \vee C_m$
 - Each clause forces a unique valuation

Boolean satisfiability

- Generating a solution
 - Try each possible assignment V to x_1, x_2, \dots, x_n
 - n variables — 2^n possible assignments
 - Is there a better algorithm? Not known
- Checking a solution
 - Given formula F and valuation $V(x)$ for each x , substitute into formula and evaluate
- Input format is important
 - Suppose a clause is a conjunction of literals ...
$$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \dots \wedge x_k)$$
 - ... and a formula is a disjunction of clauses $C_1 \vee C_2 \vee \dots \vee C_m$
 - Each clause forces a unique valuation
 - Try each clause in sequence

Travelling salesman

- A network of cities with distances between each pair

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle x, y, z, \dots, x of minimum cost, visiting all vertices

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle x, y, z, \dots, x of minimum cost, visiting all vertices
- Designing a checking algorithm

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle x, y, z, \dots, x of minimum cost, visiting all vertices
- Designing a checking algorithm
- Checking algorithm must give a yes/no answer

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle x, y, z, \dots, x of minimum cost, visiting all vertices
- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle x, y, z, \dots, x of minimum cost, visiting all vertices
- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
 - Verify that S is a cycle

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle x, y, z, \dots, x of minimum cost, visiting all vertices
- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
 - Verify that S is a cycle
 - Compute its cost

Travelling salesman

- A network of cities with distances between each pair
- A complete graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle x, y, z, \dots, x of minimum cost, visiting all vertices
- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
 - Verify that S is a cycle
 - Compute its cost
 - How to check that S is the least cost cycle?

Travelling salesman

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
 - Verify that S is a cycle
 - Compute its cost
 - How to check that S is the least cost cycle?

Travelling salesman

- Designing a checking algorithm
- Transform the problem
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
 - Verify that S is a cycle
 - Compute its cost
 - How to check that S is the least cost cycle?

Travelling salesman

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
 - Verify that S is a cycle
 - Compute its cost
 - How to check that S is the least cost cycle?
- Transform the problem
- Is there a tour with cost at most K ?

Travelling salesman

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
 - Verify that S is a cycle
 - Compute its cost
 - How to check that S is the least cost cycle?
- Transform the problem
 - Is there a tour with cost at most K ?
 - Now, given a solution S , we can check it

Travelling salesman

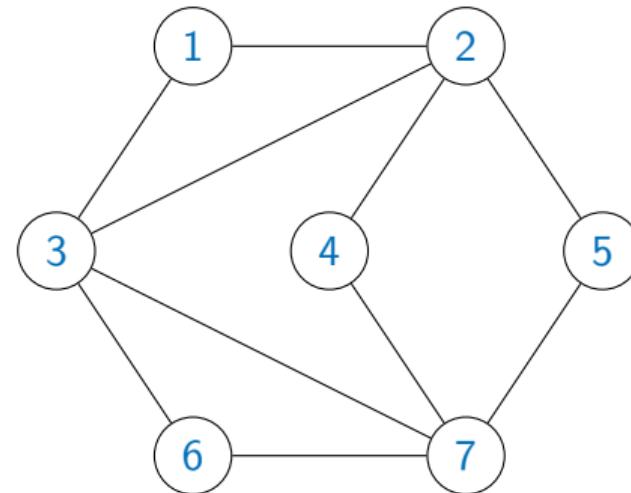
- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
 - Verify that S is a cycle
 - Compute its cost
 - How to check that S is the least cost cycle?
- Transform the problem
 - Is there a tour with cost at most K ?
 - Now, given a solution S , we can check it
 - For the original problem, cost is at most the sum of all the edge weights in the graph

Travelling salesman

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
 - Verify that S is a cycle
 - Compute its cost
 - How to check that S is the least cost cycle?
- Transform the problem
 - Is there a tour with cost at most K ?
 - Now, given a solution S , we can check it
 - For the original problem, cost is at most the sum of all the edge weights in the graph
 - Find optimum K — test different values using binary search

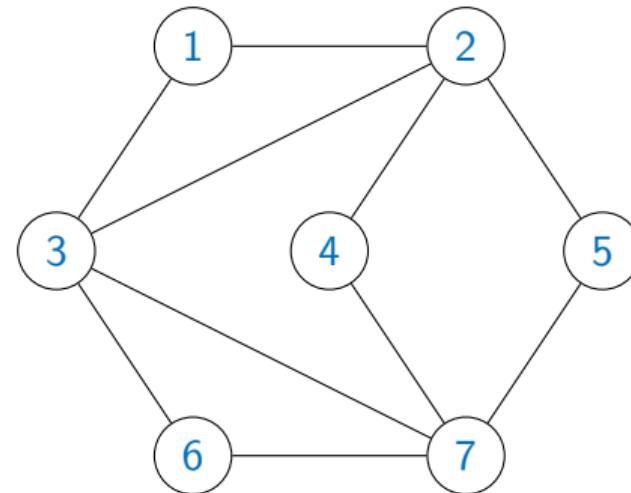
Independent set

- u, v are independent if there is no edge (u, v)



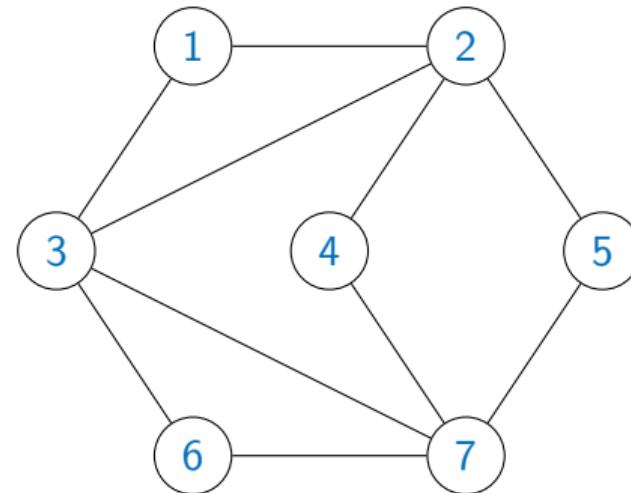
Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $u, v \in U$ is independent



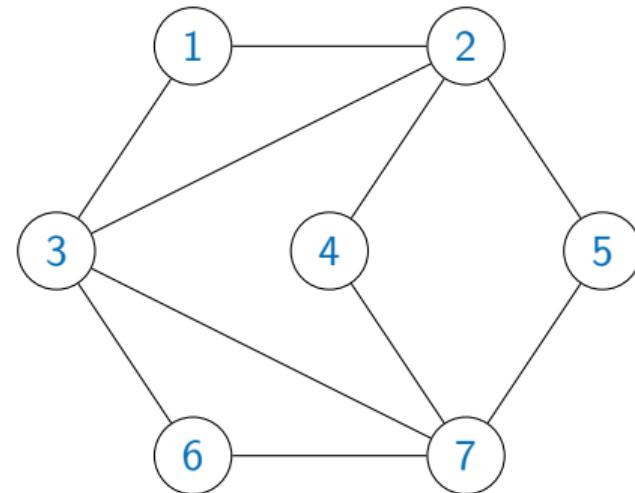
Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $u, v \in U$ is independent
- Constitute a neutral committee where none of the members know each other



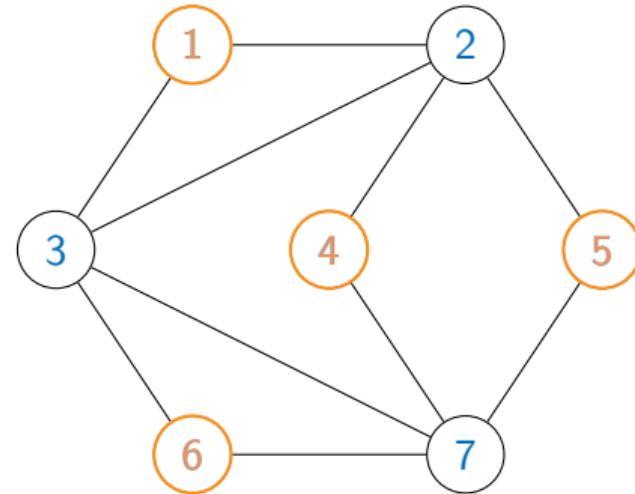
Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $u, v \in U$ is independent
- Constitute a neutral committee where none of the members know each other
- Find the largest independent set in a given graph



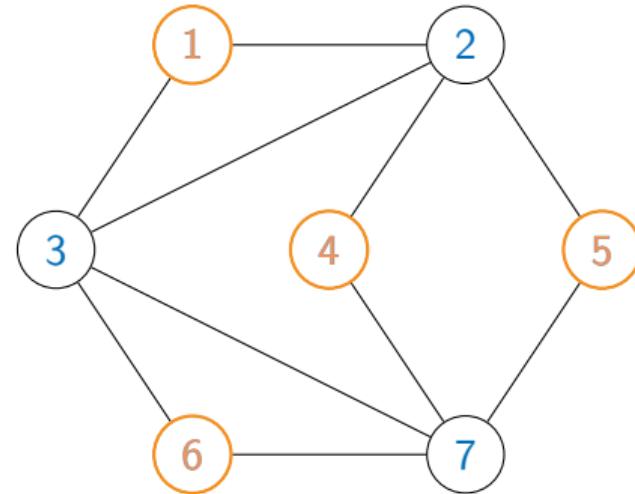
Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $u, v \in U$ is independent
- Constitute a neutral committee where none of the members know each other
- Find the largest independent set in a given graph



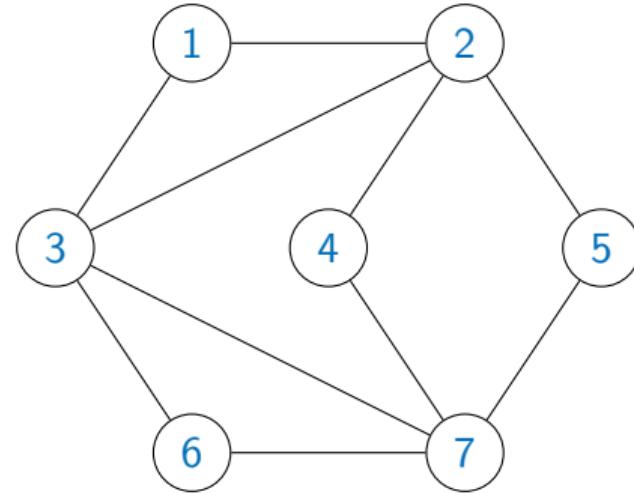
Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $u, v \in U$ is independent
- Constitute a neutral committee where none of the members know each other
- Find the largest independent set in a given graph
- Checking version: Is there an independent set of size K ?



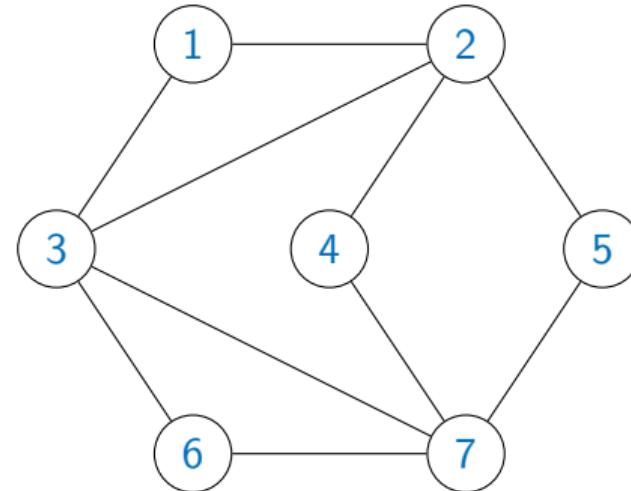
Vertex cover

- Node u covers every edge (u, v) incident on u



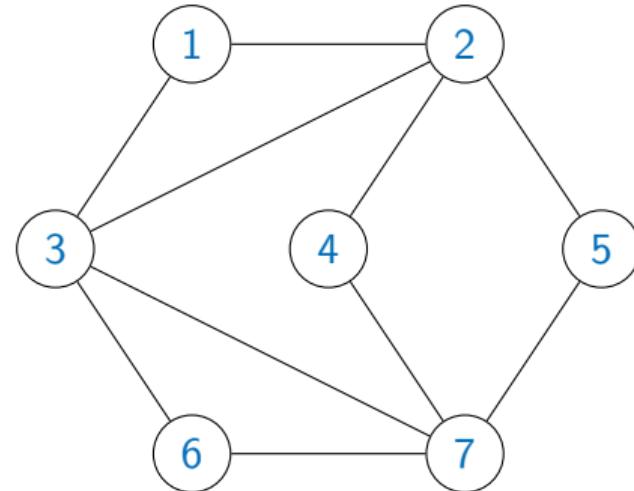
Vertex cover

- Node u covers every edge (u, v) incident on u
- U is a vertex cover if each edge in the graph is covered by some vertex in U



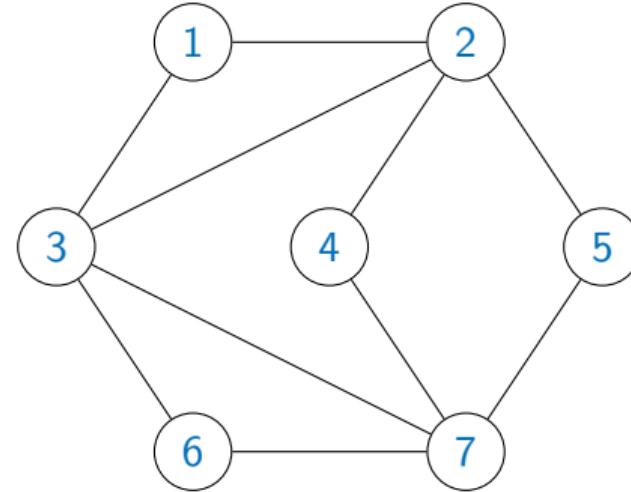
Vertex cover

- Node u covers every edge (u, v) incident on u
- U is a vertex cover if each edge in the graph is covered by some vertex in U
- Position surveillance cameras at intersections to watch all roads



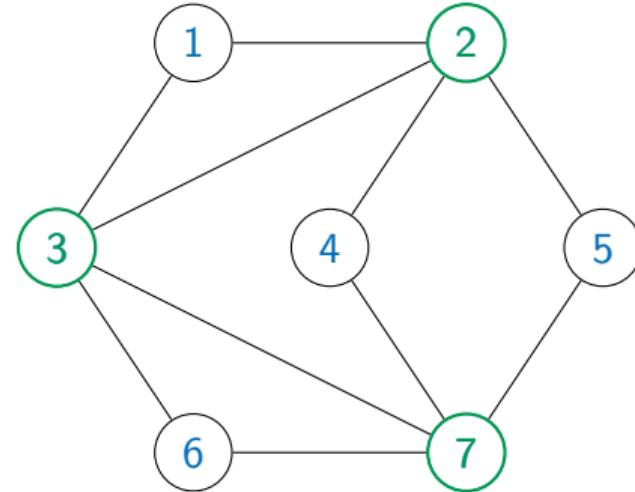
Vertex cover

- Node u covers every edge (u, v) incident on u
- U is a vertex cover if each edge in the graph is covered by some vertex in U
- Position surveillance cameras at intersections to watch all roads
- Find the smallest vertex cover in a given graph



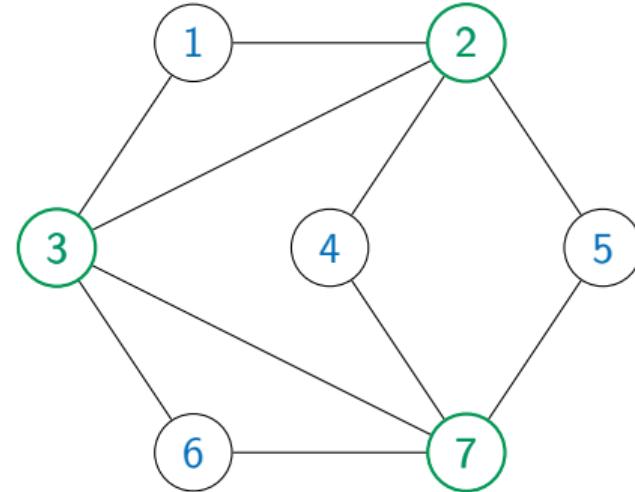
Vertex cover

- Node u covers every edge (u, v) incident on u
- U is a vertex cover if each edge in the graph is covered by some vertex in U
- Position surveillance cameras at intersections to watch all roads
- Find the smallest vertex cover in a given graph



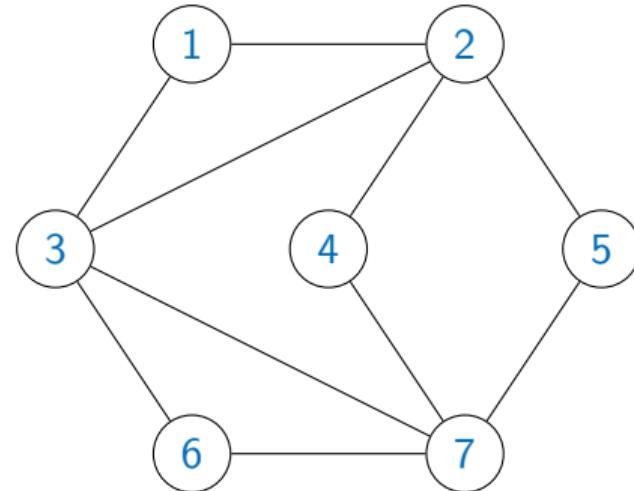
Vertex cover

- Node u covers every edge (u, v) incident on u
- U is a vertex cover if each edge in the graph is covered by some vertex in U
- Position surveillance cameras at intersections to watch all roads
- Find the smallest vertex cover in a given graph
- Checking version: Is there an vertex cover of size K?



Connecting independent set, vertex cover

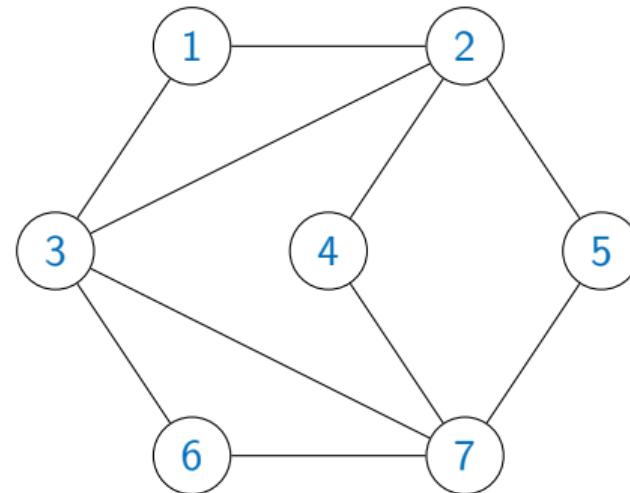
- U is an independent set of size K iff
 $V \setminus U$ is a vertex cover of size $N - K$



Connecting independent set, vertex cover

- U is an independent set of size K iff
 $V \setminus U$ is a vertex cover of size $N - K$

(\Rightarrow) Every edge (u, v) has at most one end point in U , so at least one end point in $V \setminus U$

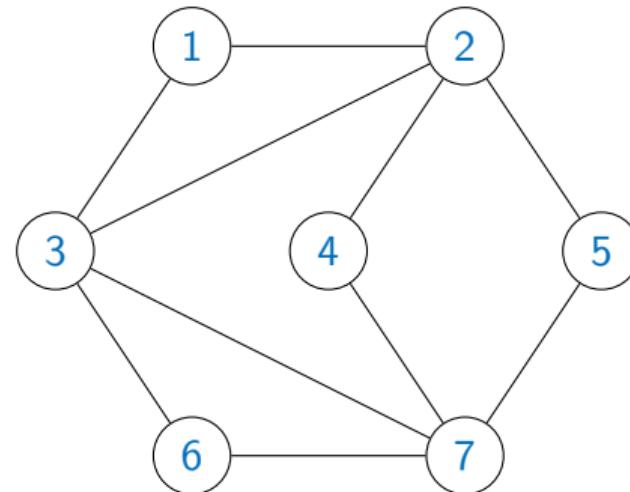


Connecting independent set, vertex cover

- U is an independent set of size K iff
 $V \setminus U$ is a vertex cover of size $N - K$

(\Rightarrow) Every edge (u, v) has at most one end point in U , so at least one end point in $V \setminus U$

(\Leftarrow) For any edge (u, v) , at least one endpoint is in $V \setminus U$, so there are no edges (u, v) within U

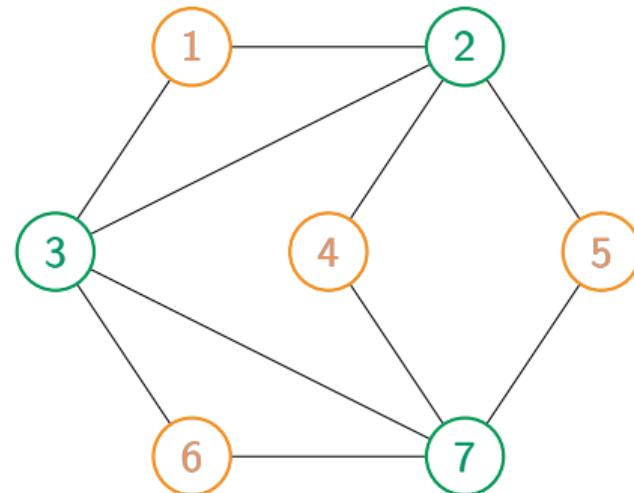


Connecting independent set, vertex cover

- U is an independent set of size K iff
 $V \setminus U$ is a vertex cover of size $N - K$

(\Rightarrow) Every edge (u, v) has at most one end point in U , so at least one end point in $V \setminus U$

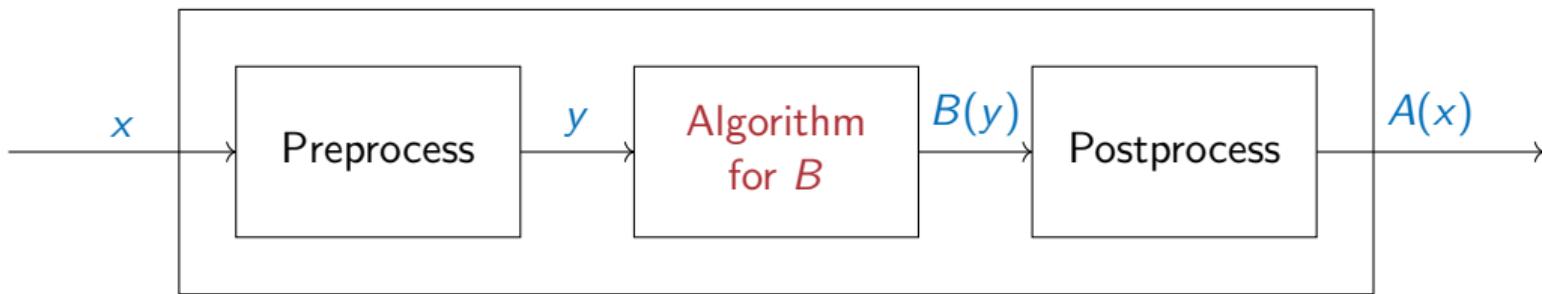
(\Leftarrow) For any edge (u, v) , at least one endpoint is in $V \setminus U$, so there are no edges (u, v) within U



Reductions

- Independent set and vertex cover reduce to each other

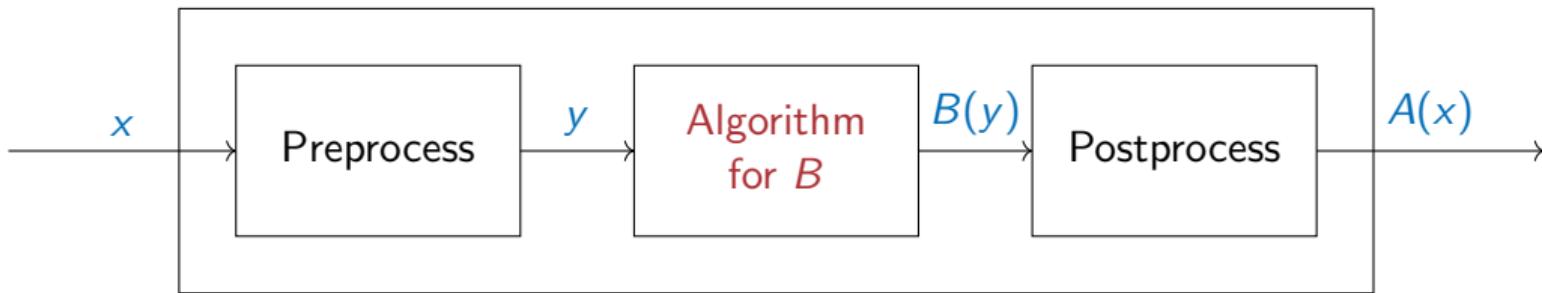
Algorithm for A



Reductions

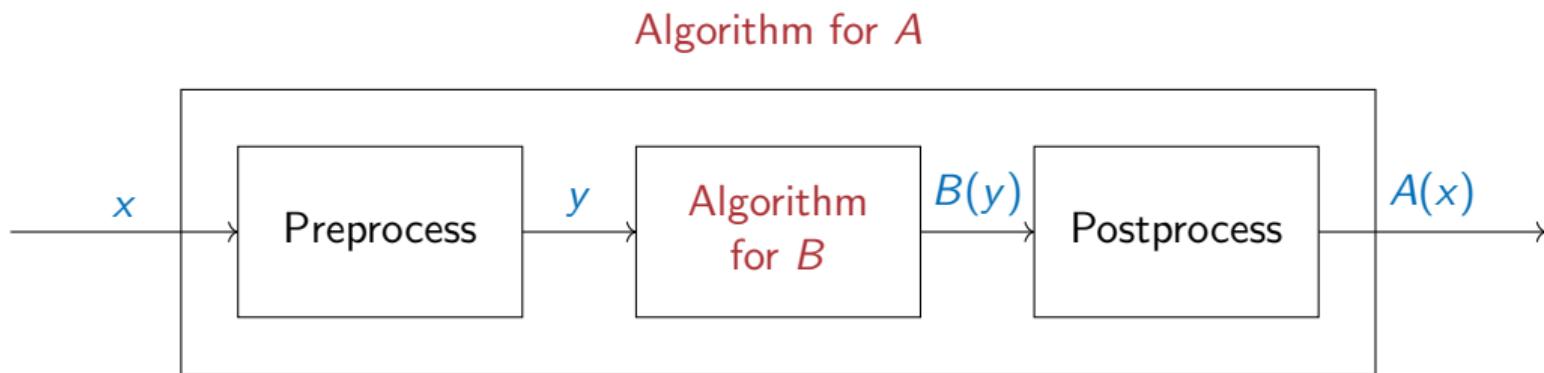
- Independent set and vertex cover reduce to each other
- Recall: if A reduces to B and A is intractable, so is B

Algorithm for A



Reductions

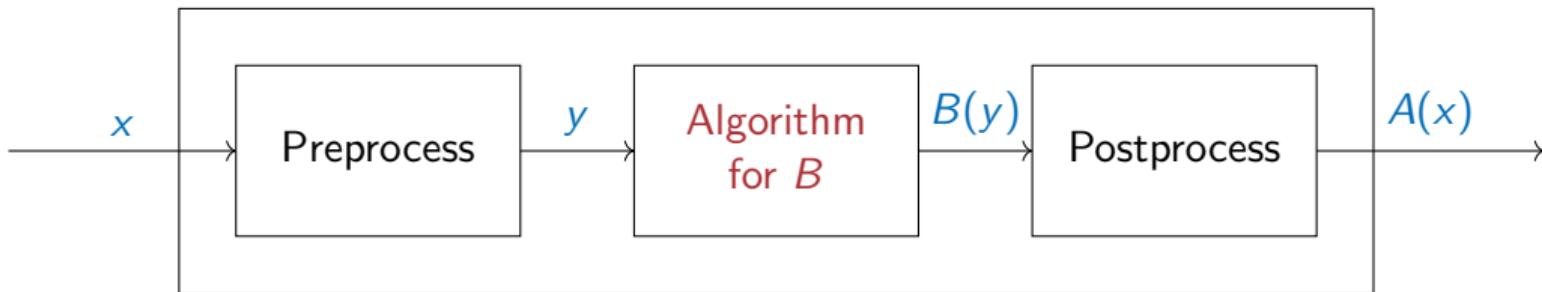
- Independent set and vertex cover reduce to each other
- Recall: if A reduces to B and A is intractable, so is B
- Many pairs of checkable problems are inter-reducible



Reductions

- Independent set and vertex cover reduce to each other
- Recall: if A reduces to B and A is intractable, so is B
- Many pairs of checkable problems are inter-reducible
- All “equally” hard

Algorithm for A



Intractability: P and NP

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 11

Checking algorithms

- Checking algorithm C for problem
- Takes in an input instance I and a solution “certificate” S for I
- C outputs yes if S represents a valid solution for I , no otherwise

The class NP

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in $\text{size}(I)$
 - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . . are all in NP

The class NP

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in $\text{size}(I)$
 - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . . are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
 - Binary search through solution space

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in $\text{size}(I)$
 - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . . are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
 - Binary search through solution space

Why “NP”?

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in $\text{size}(I)$
 - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . . are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
 - Binary search through solution space

Why “NP”?

- Non-deterministic Polynomial time

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in $\text{size}(I)$
 - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . . are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
 - Binary search through solution space

Why “NP”?

- Non-deterministic Polynomial time
- “Guess” a solution and check it

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in size(I)
 - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . . are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
 - Binary search through solution space

Why “NP”?

- Non-deterministic Polynomial time
- “Guess” a solution and check it
- Origins in computability theory

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in size(I)
 - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . . are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
 - Binary search through solution space

Why “NP”?

- Non-deterministic Polynomial time
- “Guess” a solution and check it
- Origins in computability theory
- Non-deterministic Turing machines . . .

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is P = NP?

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is P = NP?
- Is efficient checking same as efficient generation?

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is $P = NP$?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

$P \neq NP?$

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is $P = NP$?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is $P = NP$?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

$P \neq NP?$

- A more formal reason to believe this?

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is $P = NP$?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

$P \neq NP$?

- A more formal reason to believe this?
- Many “natural” problems are in NP
 - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . .

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is $P = NP$?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

$P \neq NP?$

- A more formal reason to believe this?
- Many “natural” problems are in NP
 - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . .
- These are all inter-reducible,
 - Like vertex cover, independent set

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is $P = NP$?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

$P \neq NP?$

- A more formal reason to believe this?
- Many “natural” problems are in NP
 - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . .
- These are all inter-reducible,
 - Like vertex cover, independent set
- If we can solve one efficiently, we can solve them all!

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Clause — disjunction of literals,
 $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_k)$

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Clause — disjunction of literals,
 $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_k)$
- Formula — conjunction of clauses,
 $C_1 \wedge C_2 \wedge \dots \wedge C_m$

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Clause — disjunction of literals,
 $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_k)$
- Formula — conjunction of clauses,
 $C_1 \wedge C_2 \wedge \dots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Clause — disjunction of literals,
 $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_k)$
- Formula — conjunction of clauses,
 $C_1 \wedge C_2 \wedge \dots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

Reducing SAT to 3-SAT

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Clause — disjunction of literals,
 $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_k)$
- Formula — conjunction of clauses,
 $C_1 \wedge C_2 \wedge \dots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

Reducing SAT to 3-SAT

- Consider a 5 literal clause
 $(v \vee \neg w \vee x \vee \neg y \vee z)$

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Clause — disjunction of literals,
 $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_k)$
- Formula — conjunction of clauses,
 $C_1 \wedge C_2 \wedge \dots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

Reducing SAT to 3-SAT

- Consider a 5 literal clause
 $(v \vee \neg w \vee x \vee \neg y \vee z)$
- Introduce a new literal and split the clause
 $(v \vee \neg w \vee a) \wedge (\neg a \vee x \vee \neg y \vee z)$

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Clause — disjunction of literals,
 $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_k)$
- Formula — conjunction of clauses,
 $C_1 \wedge C_2 \wedge \dots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

Reducing SAT to 3-SAT

- Consider a 5 literal clause
 $(v \vee \neg w \vee x \vee \neg y \vee z)$
- Introduce a new literal and split the clause
 $(v \vee \neg w \vee a) \wedge (\neg a \vee x \vee \neg y \vee z)$
- This formula is satisfiable iff original clause is

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Clause — disjunction of literals,
 $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_k)$
- Formula — conjunction of clauses,
 $C_1 \wedge C_2 \wedge \dots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

Reducing SAT to 3-SAT

- Consider a 5 literal clause
 $(v \vee \neg w \vee x \vee \neg y \vee z)$
- Introduce a new literal and split the clause
 $(v \vee \neg w \vee a) \wedge (\neg a \vee x \vee \neg y \vee z)$
- This formula is satisfiable iff original clause is
- Repeat till all clauses are of size 3 or less
 $(v \vee \neg w \vee a) \wedge (\neg a \vee x \vee b) \wedge (\neg b \vee \neg y \vee z)$

Boolean satisfiability

- Boolean variables x_1, x_2, x_3, \dots
- Clause — disjunction of literals,
 $(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_k)$
- Formula — conjunction of clauses,
 $C_1 \wedge C_2 \wedge \dots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

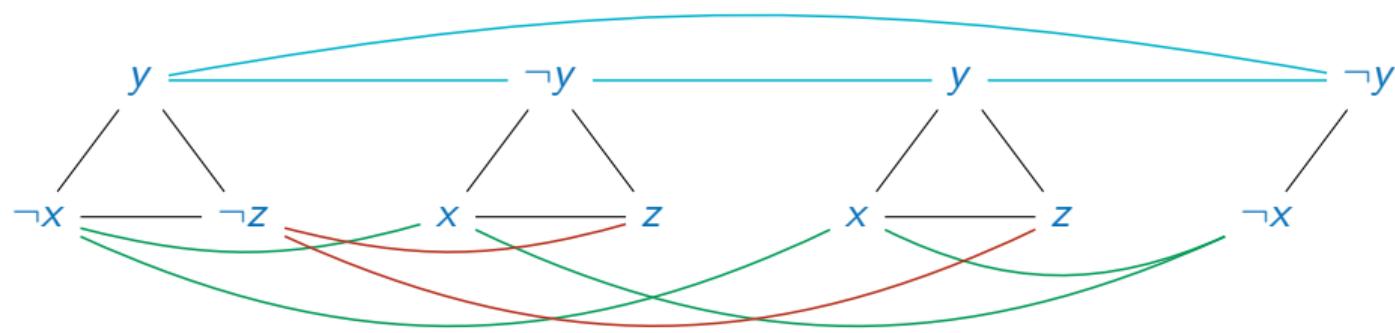
Reducing SAT to 3-SAT

- Consider a 5 literal clause
 $(v \vee \neg w \vee x \vee \neg y \vee z)$
- Introduce a new literal and split the clause
 $(v \vee \neg w \vee a) \wedge (\neg a \vee x \vee \neg y \vee z)$
- This formula is satisfiable iff original clause is
- Repeat till all clauses are of size 3 or less
 $(v \vee \neg w \vee a) \wedge (\neg a \vee x \vee b) \wedge (\neg b \vee \neg y \vee z)$
- If SAT is hard, so is 3-SAT

3-SAT to independent set

- Construct a graph from a 3-SAT formula

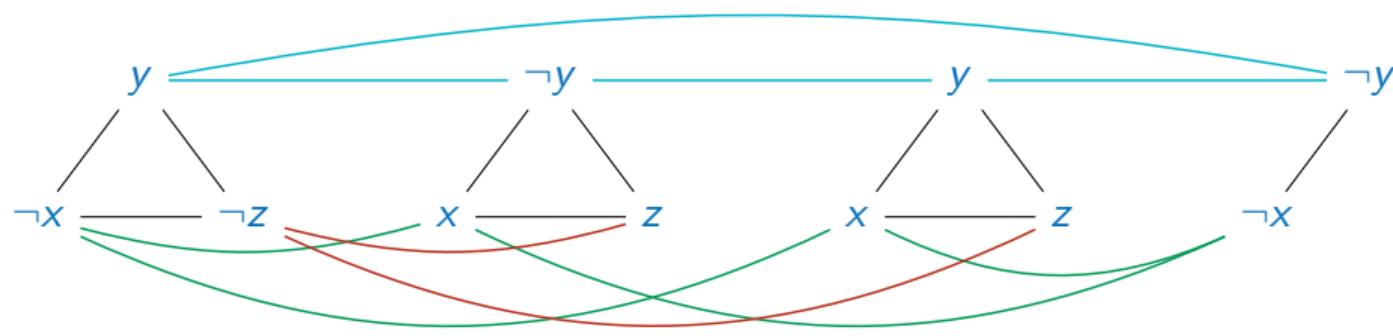
$$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$$



3-SAT to independent set

- Construct a graph from a 3-SAT formula

$$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$$

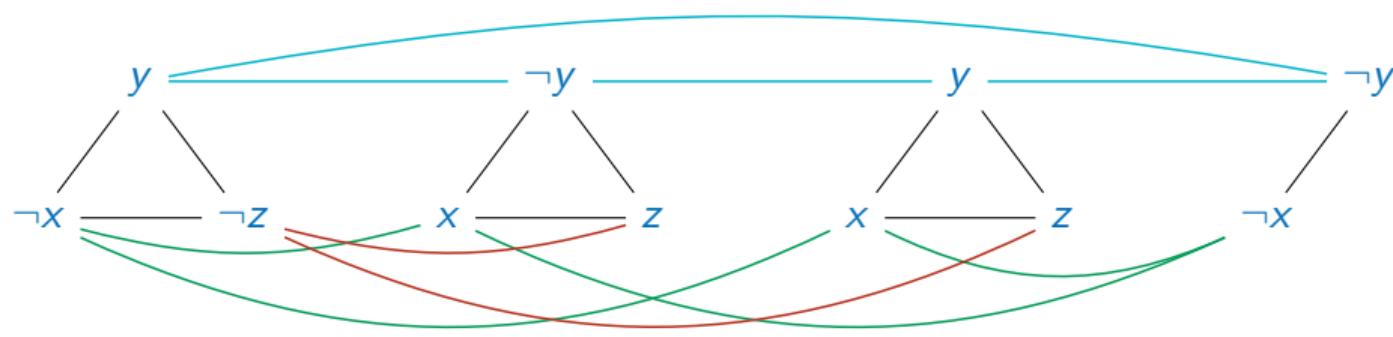


- Independent set picks one literal per clause to satisfy

3-SAT to independent set

- Construct a graph from a 3-SAT formula

$$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$$

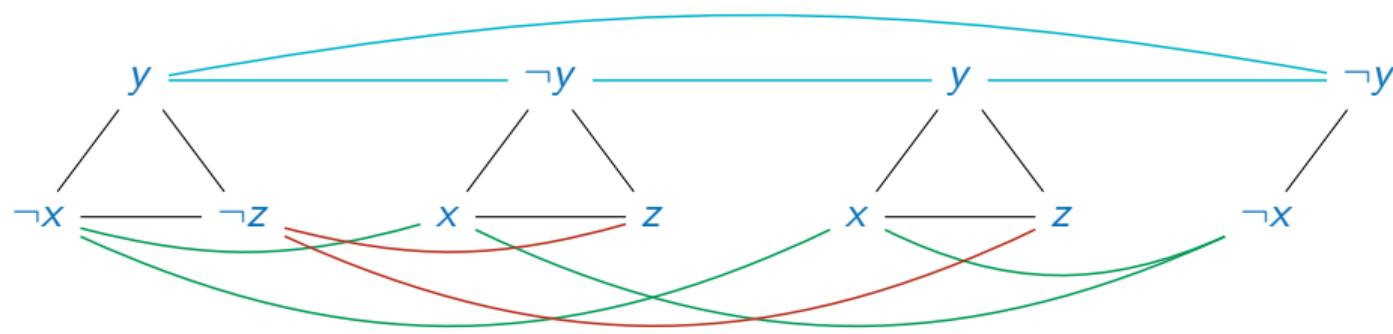


- Independent set picks one literal per clause to satisfy
- Edges enforce consistency across clauses

3-SAT to independent set

- Construct a graph from a 3-SAT formula

$$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$$



- Independent set picks one literal per clause to satisfy
- Edges enforce consistency across clauses
- Ask for size of independent set = number of clauses

Reductions within NP

- SAT \rightarrow 3-SAT, 3-SAT \rightarrow independent set, independent set \leftrightarrow vertex cover
- Reduction is transitive, so SAT \rightarrow vertex cover, ...
- Other inter-reducible NP problems
 - Travelling salesman, integer linear programming ... All these problems are “equally” hard

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- Original proof is by encoding computations of Turing machines

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any “generic” computation model — boolean circuits, register machines ...

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- SAT is said to be complete for NP

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any “generic” computation model — boolean circuits, register machines ...

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- SAT is said to be complete for NP
 - It belongs to NP

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any “generic” computation model — boolean circuits, register machines ...

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any “generic” computation model — boolean circuits, register machines ...

- SAT is said to be complete for NP
 - It belongs to NP
 - Every problem in NP reduces to it

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any “generic” computation model — boolean circuits, register machines ...

- SAT is said to be complete for NP
 - It belongs to NP
 - Every problem in NP reduces to it
- Since SAT reduces to 3-SAT, 3-SAT is also NP-complete

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any “generic” computation model — boolean circuits, register machines ...

- SAT is said to be complete for NP
 - It belongs to NP
 - Every problem in NP reduces to it
- Since SAT reduces to 3-SAT, 3-SAT is also NP-complete
- In general, to show P is NP-complete, reduce some existing NP-complete problem to P

$P \neq NP?$

- A large class of practically useful problems are NP-complete
 - Scheduling, bin-packing, optimal tours . . .

$P \neq NP?$

- A large class of practically useful problems are NP-complete
 - Scheduling, bin-packing, optimal tours . . .
- If one of them has a solution in P, all of them do

$P \neq NP?$

- A large class of practically useful problems are NP-complete
 - Scheduling, bin-packing, optimal tours . . .
- If one of them has a solution in P, all of them do
- Many smart people have been working on these problems for centuries

- A large class of practically useful problems are NP-complete
 - Scheduling, bin-packing, optimal tours . . .
- If one of them has a solution in P, all of them do
- Many smart people have been working on these problems for centuries
- Empirical evidence that NP is different from P

- A large class of practically useful problems are NP-complete
 - Scheduling, bin-packing, optimal tours . . .
- If one of them has a solution in P, all of them do
- Many smart people have been working on these problems for centuries
- Empirical evidence that NP is different from P
- But a formal proof is elusive, and worth \$1 million!



Programming, Data Structures and Algorithms using Python

Prof. Madhavan Mukund
Director
Chennai Mathematical Institute

Mr. Omkar Joshi
Course Instructor
IITM Online Degree Programme



Programming, Data Structures and Algorithms using Python

Summary of weeks 1 to 3

Content

- Timing our code
- Analysing an algorithm
- Asymptotic notations
- Orders of magnitude
- Calculating complexity
- Searching algorithms
- Selection sort
- Insertion sort
- Merge sort
- Quicksort
- Comparing sorting algorithms
- Lists vs. Arrays
- Implementation of lists in Python
- Implementation of dictionaries in Python

Timing our code

```
import time  
  
start = time.perf_counter()  
  
...  
  
# Execute some code  
  
...  
  
end = time.perf_counter()  
  
time_elapsed = end - start
```

Analysing an algorithm

- Two parameters to measure an algorithm
 - Running time (time complexity)
 - Memory requirement (space complexity)
- Running time $T(n)$ is a function of input size n
- Upper bound on worst case gives us an overall guarantee on performance

Asymptotic notations

- Big O notation:

$f(n)$ is $O(g(n))$ means $g(n)$ is an upper bound for $f(n)$

- Ω notation:

$f(n)$ is $\Omega(g(n))$ means $g(n)$ is a lower bound for $f(n)$

- Θ notation:

$f(n)$ is $\Theta(g(n))$ means we have found matching upper and lower bounds (optimal solution)

Orders of magnitude

- Commonly encountered classes of functions
- In each case c is a positive constant and n increases without bound
- The slower-growing functions are listed first

Notation	Name
$O(c)$	Constant
$O(\log \log n)$	Double logarithmic
$O(\log n)$	Logarithmic
$O((\log n)^c), c > 1$	Polylogarithmic
$O(n^c), 0 < c < 1$	Fractional power
$O(n)$	Linear
$O(n \log n)$	Loglinear
$O(n^2)$	Quadratic
$O(n^c)$	Polynomial
$O(c^n), c > 1$	Exponential
$O(n!)$	Factorial

Calculating complexity

- Depends on the type of algorithm
- Iterative programs
 - Focus on loops
- Recursive programs
 - Write and solve a recurrence relation

Searching algorithms

Linear search

- Naïve solution
- Check every element in the list one by one
- Worst case is when element is not present in the list

Best case	Average case	Worst case
$O(1)$	$O(n)$	$O(n)$

Binary search

- Prerequisite: list must be sorted
- Compare with midpoint element
- Halve the list till interval becomes empty
- Recurrence: $T(n) = T(n / 2) + 1$

Best case	Average case	Worst case
$O(1)$	$O(\log n)$	$O(\log n)$

Selection sort

- It is an intuitive algorithm
- Repeatedly find the minimum/maximum element and append it to the sorted list
- Swapping elements helps us avoid use of second list
- Number of comparisons: $T(n) = n + (n - 1) + \dots + 1$
 $= n(n + 1) / 2$
- The time complexity of Selection sort remains the same irrespective of the sequence of elements

Insertion sort

- It is another intuitive algorithm
- Repeatedly pick an element and insert it into the sorted list
- Number of comparisons: $T(n) = n + (n - 1) + \dots + 1$
 $= n(n + 1) / 2$
- The time complexity of Insertion sort varies based on the sequence of elements

Merge sort

- Divide the list into two halves
- Separately sort the left and right half
- Combine the two sorted lists A and B to get a fully sorted list C
 - If A is empty, copy B into C
 - If B is empty, copy A into C
 - Otherwise, compare first elements of A and B
 - Move the smaller of the two to C
 - Repeat till you exhaust A and B
- Recurrence: $T(n) = 2T(n / 2) + n$

Quicksort

- Choose a pivot element (typically the first element)
- Partition the list into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions
- This allows an in-place sort
- Iterative implementation is possible to avoid the cost of recursive calls

Comparing sorting algorithms

Parameter	Selection sort	Insertion sort	Merge sort	Quicksort
Best case	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Average case	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Worst case	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
In-place	Yes	Yes	No	Yes
Stable	No	Yes	Yes	No

Lists vs. Arrays

Lists	Arrays
Flexible length	Fixed size
Values are scattered in memory	Allocate a contiguous block of memory
Need to follow links to access	Random access
Insertion and deletion is easy	Insertion and deletion is expensive
Swapping elements takes constant time	Swapping elements takes linear time

Implementation of lists in Python

- Python lists are NOT implemented as flexible linked lists
- Underlying interpretation maps the list to an array
 - Assign a fixed block when you create a list
 - Double the size if the list overflows the array
- Keep track of the last position of the list in the array
 - `l.append()` and `l.pop()` are constant time, amortized – $O(1)$
 - Insertion/deletion require time $O(n)$
- Effectively, Python lists behave more like arrays than lists

Implementation of dictionaries in Python

- A dictionary is implemented as a hash table
 - An array plus a hash function
- Creating a good hash function is important (and hard!)
- Need a strategy to deal with collisions
 - Open addressing/closed hashing – probe for free space in the array
 - Open hashing – each slot in the hash table points to a list of key-value pairs



Programming, Data Structures and Algorithms using Python

Prof. Madhavan Mukund
Director
Chennai Mathematical Institute

Mr. Omkar Joshi
Course Instructor
IITM Online Degree Programme



Programming, Data Structures and Algorithms using Python

Summary of weeks 4 to 6

Content

- Graphs
- Graph representation
- Breadth First Search(BFS)
- Depth First Search(DFS)
- Applications of BFS & DFS
- Directed Acyclic Graphs(DAGs)
- Shortest Paths in Weighted Graphs
- Dijkstra's algorithm
- Bellman-Ford Algorithm
- Floyd-Warshall algorithm
- Trees
- Spanning trees
- Prim's algorithm
- Kruskal's algorithm
- Efficient data structures

Graphs

- A graph represents relationships between entities
 - Entities are vertices/nodes
 - Relationships are edges
- A graph can be directed or undirected
 - A is a parent of B – directed
 - A is a friend of B – undirected
- Paths are sequence of connected edges
- Reachability: is there a path from node **u** to node **v**?

Graph representation

- $G = (V, E)$
 - $|V| = n$
 - $|E| = m$
 - If G is connected, m can vary from $n - 1$ to $n(n - 1) / 2$
- Adjacency matrix
 - $n \times n$ matrix, $\text{AMat}[i, j] = 1$ iff $(i, j) \in E$
- Adjacency list
 - Dictionary of lists
 - For each vertex i , $\text{AList}[i]$ is the list of neighbors of i

Breadth First Search(BFS)

- Breadth first search is a systematic strategy to explore a graph, level by level
- Maintain visited but unexplored vertices in a queue
- Complexity is $O(n^2)$ using adjacency matrix, $O(m + n)$ using adjacency list
- Add parent information to recover the path to each reachable vertex
- Maintain level information to record length of the shortest path, in terms of number of edges

Depth First Search(DFS)

- DFS is another systematic strategy to explore a graph
- DFS uses a stack to suspend exploration and move to unexplored neighbors
- Complexity is $O(n^2)$ using adjacency matrix, $O(m + n)$ using adjacency list
- Useful features can be found by recording the order in which DFS visits vertices

Applications of BFS & DFS

- Paths discovered by BFS are shortest paths in terms of number of edges
- BFS and DFS can be used to identify connected components in an undirected graph
- BFS and DFS identify an underlying tree
- Use of DFS numbering
 - Strongly connected components
 - Articulation points(cut vertices) and bridges(cut edges)

Directed Acyclic Graphs(DAGs)

- Directed acyclic graphs are a natural way to represent dependencies
- Topological sorting
 - It gives a feasible schedule that represents dependencies
 - Complexity is $O(n^2)$ using adjacency matrix, $O(m + n)$ using adjacency list
- Longest paths
 - Directed acyclic graphs are a natural way to represent dependencies
 - Complexity is $O(m + n)$

Shortest Paths in Weighted Graphs

- Single source shortest paths (Dijkstra's algorithm)
 - Find shortest paths from a fixed vertex to every other vertex
- All pairs shortest paths (Floyd-Warshall algorithm)
 - Find shortest paths between every pair of vertices i and j
- Negative edge weights and Negative cycles
 - If a graph has a negative cycle, shortest paths are not defined
 - Without negative cycles, we can compute shortest paths even if some weights are negative (Bellman-Ford Algorithm)

Dijkstra's algorithm

- Dijkstra's algorithm computes single source shortest paths
- Uses a greedy strategy to identify vertices to visit
 - Next vertex to visit is based on shortest distance computed so far
 - Correctness requires edge weights to be non-negative
- Complexity is $O(n^2)$
 - Even with adjacency lists
 - Bottleneck is identifying unvisited vertex with minimum distance
- Complexity can be improved to $O((m + n) \log n)$ by using efficient min-heap data structure

Bellman-Ford Algorithm

- Bellman-Ford algorithm computes single source shortest paths with negative weights
- Dijkstra's algorithm assumes non-negative edge weights
 - Final distance is frozen each time a vertex “burns”
 - Should not encounter a shorter route discovered later
 - Without negative cycles, every shortest route is a path
 - Every prefix of a shortest path is also a shortest path
- Iteratively find shortest paths of length 1, 2, ..., n – 1
- Update distance to each vertex with every iteration
- Complexity is $O(n^3)$ using adjacency matrix, $O(mn)$ using adjacency list

Floyd-Warshall algorithm

- Floyd-Warshall algorithm computes all pairs shortest paths
- Complexity using simple nested loop implementation is $O(n^3)$

Trees

- A tree is a connected acyclic graph
- A tree with n vertices has exactly $n - 1$ edges
- Adding an edge to a tree creates a cycle
- Deleting an edge from a tree splits the tree
- In a tree, every pair of vertices is connected by a unique path

Spanning trees

- Retain a minimal set of edges so that graph remains connected
- A graph can have multiple spanning trees
- Minimum Cost Spanning Tree (MCST) – among the different spanning trees, choose one with minimum cost
- A graph can have multiple MCSTs, but the cost will always be unique
- Building a MCST
 - Prim's algorithm
 - Kruskal's algorithm

Prim's algorithm

- Start with a smallest weight edge overall
- Incrementally grow the MCST from any vertex
- Extend the current tree by adding the smallest edge from some vertex in the tree to a vertex not yet in the tree
- Implementation is similar to Dijkstra's algorithm
- Complexity is $O(n^2)$
 - Even with adjacency lists
 - Bottleneck is identifying unvisited vertex with minimum distance
- Complexity can be improved to $O((m + n) \log n)$ by using efficient min-heap data structure

Kruskal's algorithm

- Start with n components, each an isolated vertex
- Process edges in ascending order of cost
- Include edge if it does not create a cycle
 - Challenge is to keep track of connected components
 - Maintain a dictionary to record component of each vertex
 - Initially each vertex is an isolated component
 - When we add an edge (u, v) , merge the components of u and v
- Complexity is $O(n^2)$ due to naive handling of components, can be improved to $O(m \log n)$ by using efficient union-find data structure

Efficient data structures

1. Union-Find

- Across m operations, amortized complexity of each `Union()` operation is $\log m$
- With clever updates to the tree, `Find()` has amortized complexity very close to $O(1)$

2. Priority queues

- `insert()` operation is $O(\sqrt{n})$
- `delete_max()` operation is $O(\sqrt{n})$
- Processing n items is $O(n\sqrt{n})$

Efficient data structures

3. Heaps

- Binary tree filled level by level, left to right
- The value at each node is at least as big the values of its children (max-heap)
- No “holes” allowed and cannot leave a level incomplete
- `insert()` operation is $O(\log n)$
- `delete_max()` / `delete_min()` operation is $O(\log n)$
- `heapify()` builds a heap in $O(n)$
- Heaps can also be used to sort a list in place in $O(n \log n)$

Efficient data structures

4. Search trees

- For each node with value v , all values in the left subtree are $< v$
- For each node with value v , all values in the right subtree are $> v$
- No duplicate values
- Each node has three fields, `value`, `left`, `right`
- Traversals: In-order, Pre-order, Post-order
- Worst case: An unbalanced tree with n nodes may have height $O(n)$
- `find()`, `insert()` and `delete()` all walk down a single path

Efficient data structures

5. Balanced search trees

- Left and right subtrees should be “equal”
- Two possible measures: `size` and `height`
- Height balanced trees: height of left child and height of right child differ by at most 1 (AVL trees)
- Using rotations, we can maintain height balance
- AVL trees with `n` nodes will have height $O(\log n)$
- `find()`, `insert()` and `delete()` all walk down a single path, take only $O(\log n)$



Programming, Data Structures and Algorithms using Python

Prof. Madhavan Mukund
Director
Chennai Mathematical Institute

Mr. Omkar Joshi
Course Instructor
IITM Online Degree Programme



Programming, Data Structures and Algorithms using Python

Summary of weeks 7 to 9

Content

- Greedy algorithms
 - Interval scheduling
 - Minimizing lateness
 - Huffman coding
- Divide and conquer
 - Counting inversions
 - Closest pair of points
 - Integer multiplication
 - Quick select
 - Recursion trees
- Dynamic programming
 - Memoization
 - Grid paths
 - Common subwords and subsequences
 - Edit distance
 - Matrix multiplication

Greedy algorithms

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion
- Never go back and revise an earlier decision
- Drastically reduces space to search for solutions
- Greedy strategy needs a proof of optimality
- Examples: Dijkstra's algorithm, Prim's algorithm, Kruskal's algorithm
- Applications: Interval Scheduling, Minimizing Lateness, Huffman Coding

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
 - Different teachers want to book the classroom
 - Slots may overlap, so not all bookings can be honored
 - Choose a subset of bookings to maximize the number of teachers who get to use the room
1. Choose the booking whose starting time is earliest
 2. Choose the booking spanning the shortest interval
 3. Choose the booking that overlaps with minimum number of other bookings
 4. Choose the booking whose finish time is the earliest

Minimizing lateness

- IIT Madras has a single 3D printer
 - A number of users need to use this printer
 - Each user will get access to the printer, but may not finish before deadline
 - Goal is to minimize the lateness
1. Schedule requests in increasing order of length
 2. Give priority to requests with smaller slack time
 3. Schedule requests in increasing order of deadlines

Huffman coding

- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency
- Store frequencies in an array A
 - Linear scan to find minimum values
 - $|A| = k$, number of recursive calls is $k - 1$
 - Complexity is $O(k^2)$
- Instead, maintain frequencies in an heap
 - Extracting two minimum frequency letters and adding back compound letter are both $O(\log k)$
 - Complexity drops to $O(k \log k)$

Divide and conquer

- Break the problem into disjoint subproblems
- Combine these subproblem solutions efficiently
- Examples: Merge sort, Quicksort
- Applications: Counting inversions, Closest pair of points, Integer multiplication, Quick select

Counting inversions

- Compare your profile with other customers
- Identify people who share your likes and dislikes
- No inversions – rankings identical
- Every pair inverted – maximally dissimilar
- Number of inversions ranges from 0 to $n(n - 1) / 2$
- An inversion is a pair (i, j) , $i < j$, where j appears before i
- Recurrence: $T(n) = 2T(n / 2) + n = O(n \log n)$

Closest pair of points

- Split the points into two halves by vertical line
- Recursively compute closest pair in each half
- Compare shortest distance in each half to shortest distance across the dividing line
- Recurrence: $T(n) = 2T(n / 2) + O(n)$
- Overall: $O(n \log n)$

Integer multiplication

- Traditional method: $O(n^2)$
- Naïve divide and conquer strategy:
$$\begin{aligned} T(n) &= 4T(n / 2) + n \\ &= O(n^2) \end{aligned}$$
- Karatsuba's algorithm:
$$\begin{aligned} T(n) &= 3T(n / 2) + n \\ &= O(n^{\log 3}) \\ &\approx O(n^{1.59}) \end{aligned}$$

Quick select

- Find the k^{th} largest value in a sequence of length k
- Sort in descending order and look at position k – $O(n \log n)$
- For any fixed k , find maximum for k times – $O(kn)$
 - $k = n / 2$ (median) – $O(n^2)$
- Median of medians – $O(n)$
- Selection becomes $O(n)$ – Fast select algorithm
- Quicksort becomes $O(n \log n)$

Recursion trees

- Uniform way to compute the asymptotic expression for $T(n)$
- Rooted tree with one node for each recursive subproblem
- Value of each node is time spent on that subproblem excluding recursive calls
- Concretely, on an input of size n
 - $f(n)$ is the time spent on non-recursive work
 - r is the number of recursive calls
 - Each recursive call works on a subproblem of size n / c
- Recurrence: $T(n) = rT(n / c) + f(n)$
- Root of recursion tree for $T(n)$ has value $f(n)$
- Root has r children, each (recursively) the root of a tree for $T(n / c)$
- Each node at level d has value $f(n / c^d)$

Dynamic programming

- Solution to original problem can be derived by combining solutions to subproblems
- Examples: Factorial, Insertion sort, Fibonacci series
- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies form a DAG
- Solve subproblems in topological order
 - Never need to make a recursive call

Memoization

- Inductive solution generates same subproblem at different stages
- Naïve recursive implementation evaluates each instance of subproblem from scratch
- Build a table of values already computed – Memory table
- Store each newly computed value in a table
- Look up the table before making a recursive call

Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n) ?
- Every path has $(m + n)$ segments
- What if an intersection is blocked?
- Need to discard paths passing through
- Identify DAG structure
 - Fill the grid by row, column or diagonal
- Complexity is $O(mn)$ using dynamic programming,
 $O(m + n)$ using memoization

Common subwords and subsequences

- Given two strings, find the (length of the) longest common subword
- Subproblems are $\text{LCW}(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1)(n + 1)$ values
- $\text{LCW}(i, j)$, depends on $\text{LCW}(i + 1, j + 1)$
- Start at bottom right and fill row by row or column by column
- Complexity: $O(mn)$

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	0	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0	0
4	c	0	0	1	0	0	0	0	0
5	t	0	0	0	0	0	1	0	0
6	•	0	0	0	0	0	0	0	0

Common subwords and subsequences

- Subsequence – can drop some letters in between
- Subproblems are $\text{LCS}(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1)(n + 1)$ values
- $\text{LCS}(i, j)$, depends on $\text{LCS}(i + 1, j + 1)$, $\text{LCS}(i, j + 1)$, $\text{LCS}(i + 1, j)$
- Start at bottom right and fill row by row, column or diagonal
- Complexity: $O(mn)$

		0	1	2	3	4	5	6	
		s	e	c	r	e	t		•
0	b	4	3	2	2	2	1	0	
1	i	4	3	2	2	2	1	0	
2	s	4	3	2	2	2	1	0	
3	e	3	3	2	2	2	1	0	
4	c	2	2	2	1	1	1	0	
5	t	1	1	1	1	1	1	0	
6	•	0	0	0	0	0	0	0	0

The diagram illustrates the computation of the Longest Common Subsequence (LCS) between the strings "sec" and "retrat". The strings are aligned horizontally above a grid. The grid has rows labeled 0 through 6 (representing "sec") and columns labeled 0 through 6 (representing "retrat"). The cells contain either the characters from the strings or numerical values representing the length of the LCS up to that point. Red boxes highlight the characters 's', 'e', 'c' from "sec" and 'r', 'e', 't' from "retrat". A yellow arrow traces a path from the bottom-right cell (labeled 0) back towards the top-left, passing through cells containing 's', 'e', 'c', 'r', 'e', and 't', which corresponds to the LCS "set".

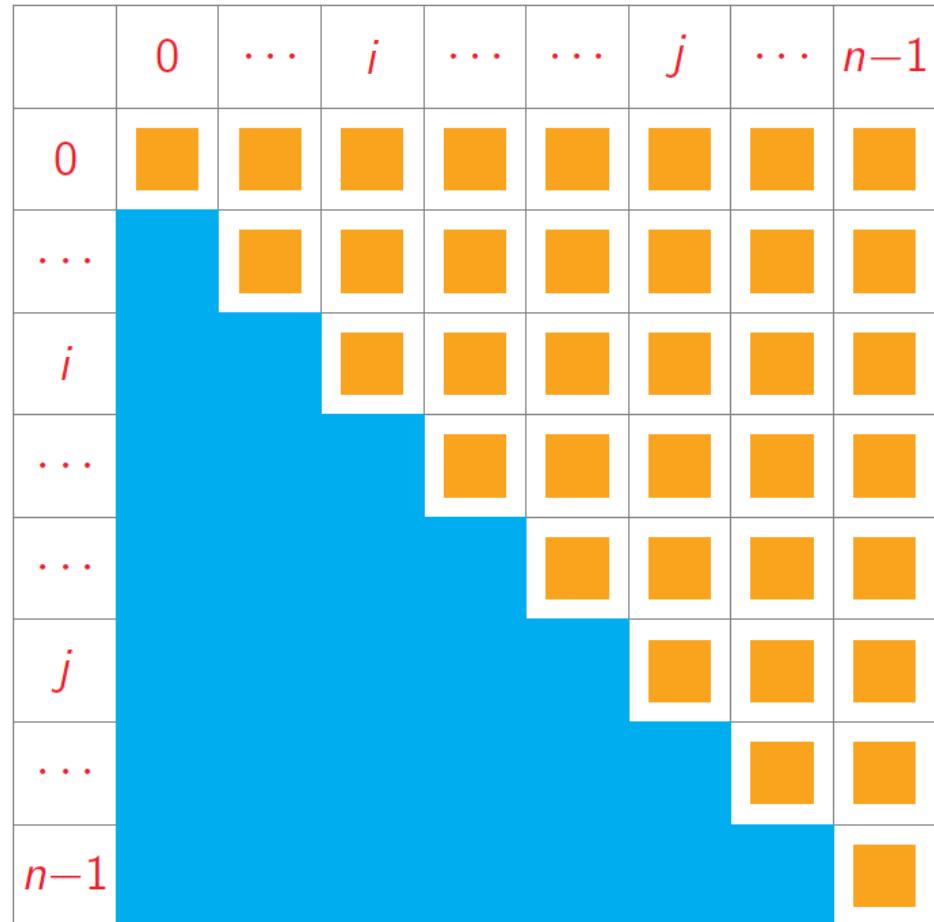
Edit distance

- Minimum number of editing operations needed to transform one document to the other
- Subproblems are $\text{ED}(i, j)$, for $0 \leq i \leq m$, $0 \leq j \leq n$
- Table of $(m + 1)(n + 1)$ values
- $\text{ED}(i, j)$, depends on $\text{ED}(i + 1, j + 1)$, $\text{ED}(i, j + 1)$, $\text{ED}(i + 1, j)$
- Start at bottom right and fill row, column or diagonal
- Complexity: $O(mn)$

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	4	4	4	4	5	6	
1	i	3	4	3	3	3	4	5	
2	s	2	3	3	2	2	3	4	
3	e	3	2	3	2	1	2	3	
4	c	4	3	2	2	1	1	2	
5	t	5	4	3	2	1	0	1	
6	•	6	5	4	3	2	1	0	

Matrix multiplication

- Matrix multiplication is associative
- Bracketing does not change answer but can affect the complexity
- Find an optimal order to compute the product
- Compute $C(i, j)$, $0 \leq i, j < n$, only for $i \leq j$
- $C(i, j)$, depends on $C(i, k - 1)$, $C(i, k)$ for every $i < k \leq j$
- Diagonal entries are base case, fill matrix from main diagonal
- Complexity: $O(n^3)$





Programming, Data Structures and Algorithms using Python

Prof. Madhavan Mukund
Director
Chennai Mathematical Institute

Mr. Omkar Joshi
Course Instructor
IITM Online Degree Programme



Programming, Data Structures and Algorithms using Python

Summary of weeks 10 & 11

Content

- String matching
 - Boyer-Moore Algorithm
 - Rabin-Karp Algorithm
 - Automata
 - Knuth-Morris-Pratt Algorithm
 - Tries
 - Regular Expressions
- Linear programming
- Network flows
 - Ford-Fulkerson algorithm
- Reductions
- Checking algorithms
- P, NP and NP-Complete

String matching

- Searching for a pattern is a fundamental problem when dealing with text
- Formal definition:
 - A *text* string t of length n
 - A *pattern* string p of length m
 - Both t and p are drawn from an *alphabet* of valid letters, denoted by Σ
 - Find every position i in t such that $t[i : i + m] == p$
- Complexity of naïve algorithm: $O(nm)$

Boyer-Moore Algorithm

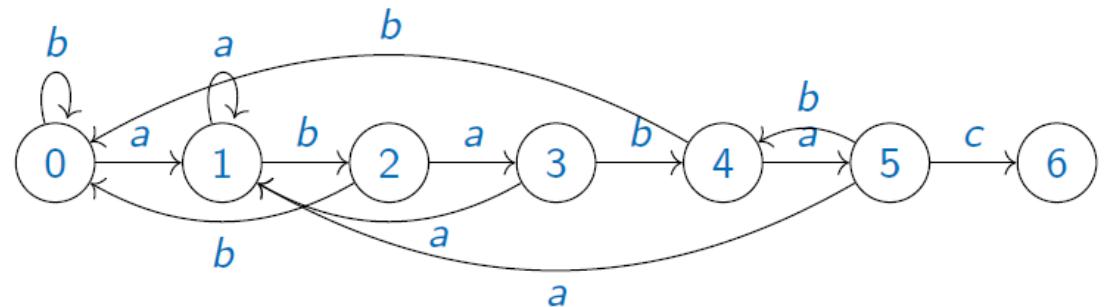
- For each starting position i in t , compare $t[i : i + m]$ with p
 - Scan $t[i : i + m]$ right to left
- If a letter c in t that does not appear in p
 - Shift the next scan to position after mismatched letter c
- If a letter c in t that does appear somewhere in p
 - Align rightmost occurrence of c in p with t
- Use a dictionary last
 - For each c in p , $\text{last}[c]$ records right most position of c in p
- Without dictionary, computing last is a bottleneck, complexity is $O(|\Sigma|)$
- The algorithm works well in practice but the worst case complexity remains $O(nm)$

Rabin-Karp Algorithm

- Any string over Σ can be thought of as a number in base 10
- Pattern p is an m digit number n_p
- Each substring of length m in the text t is again an m digit number
- Scan t and compare the number n_b generated by each block of m letters with the pattern number n_p
- Whenever $n_b = n_p$, scan and validate like brute force
 - It can be a false positive (spurious hit)
- In practice number of spurious matches will be small but the worst case complexity remains $O(nm)$
- If $|\Sigma|$ is small enough to not require modulo arithmetic, overall time is $O(n+m)$ or $O(n)$, since $m \ll n$

Automata

- It is used to keep track of longest prefix that we have matched
- It is a special type of graph where nodes are states and edges describe how to extend the match
- Using this automaton, we can do string matching in $O(n)$
- Bottleneck is precomputing the automaton
 - Overall complexity: $O(m^3 \cdot |\Sigma|)$



Processing *abababac*

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{c} 6$

Knuth-Morris-Pratt Algorithm

- It is very similar to what we did in automaton
- Precomputing step can be handled effectively using `fail()`
- The Knuth-Morris-Pratt algorithm efficiently computes the automaton describing prefix matches in the pattern p
- Complexity of preprocessing the `fail()` is $O(m)$
- After preprocessing, can check matches in the text t in $O(n)$
- Overall, KMP algorithm works in time $O(m + n)$
- However, the Boyer-Moore algorithm can be faster in practice, skipping many positions

Tries

- A *trie* is a special kind of tree derived from “information retrieval”
- Rooted tree
 - Other than root, each node labelled by a letter from Σ
 - Children of a node have distinct labels
- Each maximal path is a word
 - One word should not be a prefix of another
 - Add special end of word symbol $\$$
- Build a trie T from a set of words S with s words and n total symbols
- To search for a word w , follow its path
 - If the node we reach has $\$$ as a successor, $w \in S$
 - If we cannot complete the path, $w \notin S$
- Using a suffix trie we can answer the following:
 - Is w a substring of s
 - How many times does w occur as a substring in s
 - What is the longest repeated substring in s

Regular Expressions

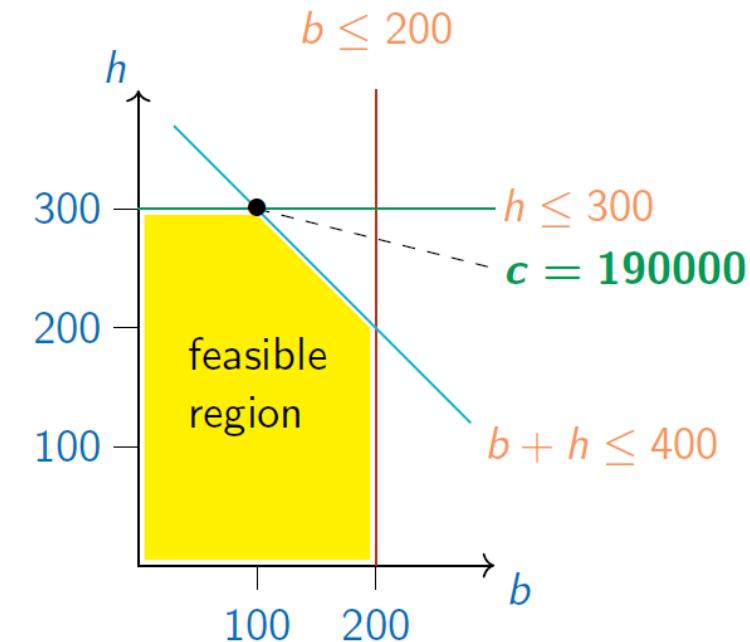
- The automaton we built had a linear structure, with a single path from start to finish
 - In general, automata can follow multiple paths and accept multiple words
 - The set of words that automata can accept are called regular sets
 - Each pattern p describes a set of words, those that it matches
 - The sets we can describe using patterns are exactly the same as those that can be accepted by automata
 - Those patterns are called regular expressions
-
- For every automaton, we can construct a pattern p that matches exactly the words that the automaton accepts
 - For every pattern p , we can construct an automaton that accepts all words that match p
 - We can extend string matching to pattern matching by building an automaton for a pattern p and processing the text through this automaton
 - Python provides a library for matching regular expressions

Linear programming

- Profit for each box of barfis is Rs.100
- Profit for each box of halwa is Rs.600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes
- Staff can produce 400 boxes a day, altogether
- What is the most profitable mix of barfis and halwa to produce?

Constrains:

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$



Objective:

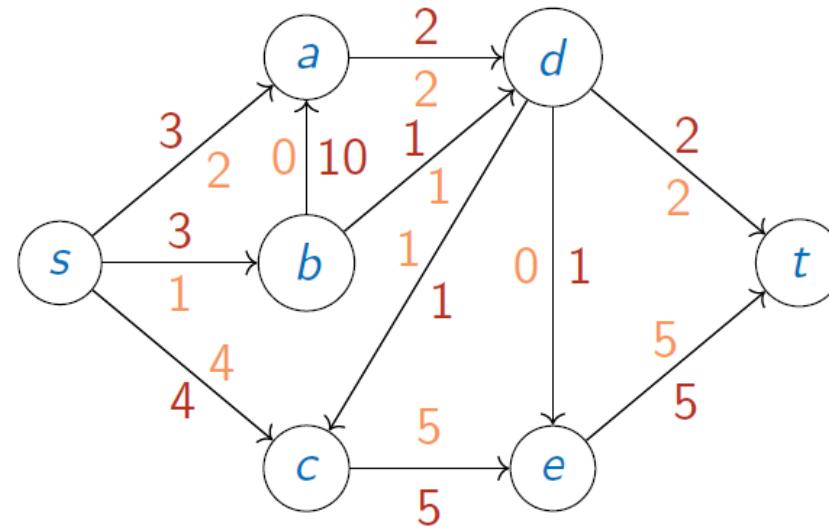
- Maximize $100b + 600h$

Linear programming

- Constraints and objective to be optimized are linear functions
 - Constrains: $a_1x_1 + a_2x_2 + \dots + a_mx_m \leq K$, $b_1x_1 + b_2x_2 + \dots + b_mx_m \geq L$
 - Objective: $c_1x_1 + c_2x_2 + \dots + c_mx_m$
- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbors, stop
- Can be exponential, but efficient in practice
- Feasible region is convex
- May be empty – constraints are unsatisfiable, no solutions
- May be unbounded – no upper/lower limit on objective

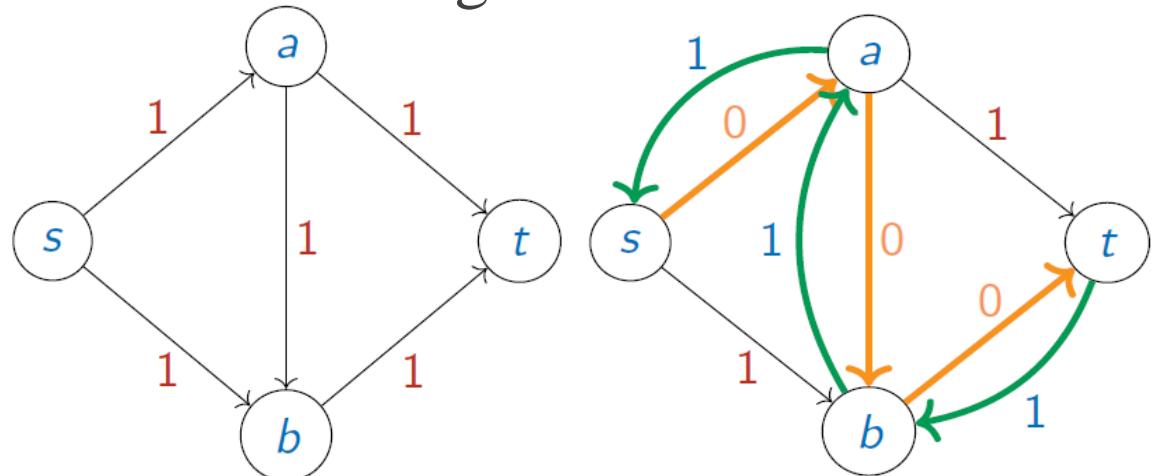
Network flows

- Network: graph $G = (V, E)$
- Special nodes: s (source), t (sink)
- Each edge e has capacity c_e
- Flow: f_e for each edge e
 - $f_e \leq c_e$
 - At each node, except s and t , sum of incoming flows equal sum of outgoing flows
- Total volume of flow is sum of outgoing flow from s



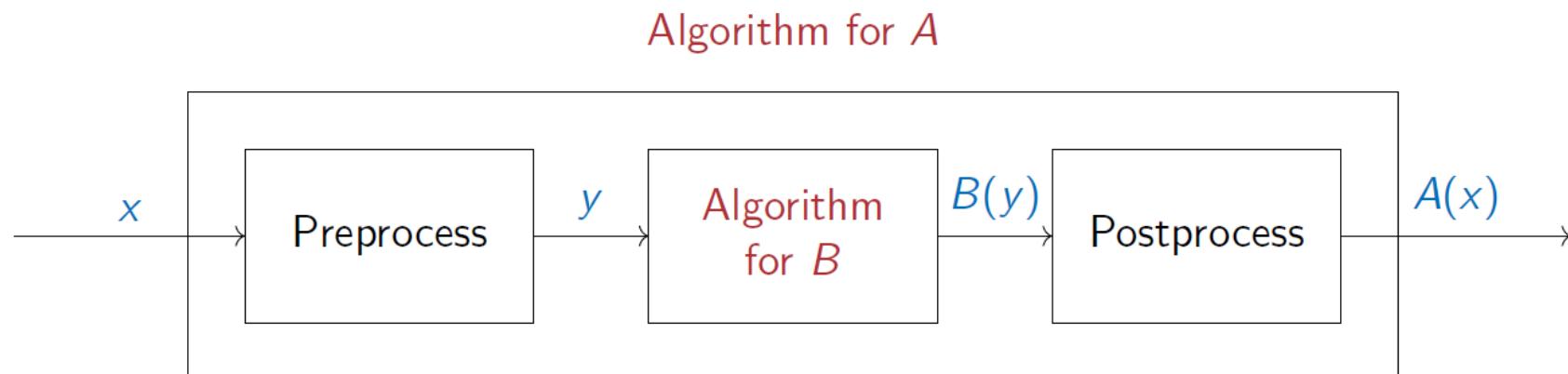
Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Network given in the example has max flow 2
- What if one chooses a bad flow to begin with?
- Add reverse edges to undo flow from previous steps
- Residual graph: for each edge e with capacity c_e and current flow f_e
 - Reduce capacity to $c_e - f_e$
 - Add reverse edge with capacity f_e
- Use BFS to find augmenting path with fewest edges



Reductions

- We want to solve problem A
- We know how to solve problem B
- Convert input for A into input for B
- Interpret output of B as output of A
- A reduces to B
- Can transfer efficient solution from B to A
- But preprocessing and postprocessing must also be efficient
- Typically, both should be polynomial time



Reductions

- Bipartite matching reduces to max flow
- Max flow reduces to LP
- Number of variables, constraints is linear in the size of the graph
- Reverse interpretation is also useful
- If A is known to be intractable and A reduces to B , then B must also be intractable
- Otherwise, efficient solution for B will yield efficient solution for A

Checking algorithms

- Factorize a large number that is the product of two primes
- Generate a solution
 - Given a large number N , find p and q such that $pq = N$
- Check a solution
 - Given a solution p and q , verify that $pq = N$
- Examples: satisfiability, travelling salesman, vertex cover, independent set, etc.
- Checking algorithm C for problem P
- Takes an input instance I for P and a solution “certificate” S for I
- C outputs yes if S represents a valid solution for I , no otherwise
- For factorization
 - I is N
 - S is $\{p, q\}$
 - C involves verifying that $pq = N$

P, NP and NP-Complete

- P (**P**olynomial) is the class of problems with regular polynomial time algorithms (worst-case complexity)
- NP (**N**on-deterministic **P**olynomial) is the class of problems with checking algorithms
- An algorithm **A** is NP-Complete if it satisfies two conditions:
 - **A** is in NP
 - Every algorithm in NP is polynomial time reducible to **A**

