# Scientific Computation

## Spring, 2019

## Lecture 6

Imperial College
London

# Pattern search

- "Naive" approach:
  - Loop through *S* one character at a time
    - Check for matches with P one character at a time (breaking this check after first mismatch
  - What is the cost?
    - Worst-case, O(MN) operations
    - (How) can we do better?
  - Binary search?
    - N $\log_2$N to sort
    - Then $\log_2$(N) for each search
    - But this requires storing N length-M strings/arrays
  - Hash table? Faster, but still with wasteful memory usage

# Pattern search

- A (partial) solution:
  - Use a *rolling* hash function
    - Compute hash for pattern, *P*
    - Then apply function sequentially to each length-M substring in *S*
    - For a well-designed hash function, cost will be O(M + N)
    - And memory usage will also be O(M+N)

# Pattern search

- A (partial) solution:
  - Use a *rolling* hash function
    - Compute hash for pattern, *P*
    - Then apply function sequentially to each length-M substring in *S*
    - For a well-designed hash function, cost will be O(M + N)
    - And memory usage will also be O(M+N)

- What is a rolling hash function?
  - First, genetic sequences can be rewritten in base 4
    - A=0, C=1, G=2, T=3
  - A simplistic function – convert sequence from base 4 to base 10
  - Example: *S* = GCTAT = 21303

    $$H(S) = 2*4^4 + 1*4^3 + 3*4^2 + 0*4^1 + 3$$
  - Or more generally, evaluate a M-1$^{th}$-order polynomial for each length-M substring of *S*

# Pattern search

- **What is a rolling hash function?**
  - **First, genetic sequences can be rewritten in base 4**
    - **A=0, C=1, G=2, T=3**
  - **A simplistic function – convert sequence from base 4 to base 10**
  - **Example: *S* = GCTAT =** 21303

    $H(S) = 2*4^4 + 1*4^3 + 3*4^2 + 0*4^1 + 3$
  - **Or more generally, evaluate a M-1$^{th}$-order polynomial for each length-M substring of *S***
  - **This doesn't really help – O(M) operations for all (N-M) sub-strings**
  - **Need to think about computing hash of consecutive sub-strings**

# Pattern search

- **Need to think about computing hash of consecutive sub-strings**
- **Let $S_i$ be the i^th^ length-M sub-string in $S$**
- **And $H(S_i)$ is computed as before:**

$$H(S_i) = S_{i,1}\,4^{M-1} + S_{i,2}\,4^{M-2} + \ldots + S_{i,M-1}\,4 + S_{i,M}$$

- **Then:**

$$H(S_{i+1}) = H(S_i)*4 - S_{i,1}4^{M} + S_{i+1,M}$$

- **So, 4 rather than ~2M operations per hash evaluation (except i=1)**

# Pattern search

- **Need to think about computing hash of consecutive sub-strings**
- **Let $S_i$ be the $i^{th}$ length-M sub-string in $S$**
- **And $H(S_i)$ is computed as before:**

$$H(S_i) = S_{i,1}\, 4^{M-1} + S_{i,2}\, 4^{M-2} + \ldots + S_{i,M-1}\, 4 + S_{i,M}$$

- **Then:**

$$H(S_{i+1}) = H(S_i)*4 - S_{i,1}4^{M} + S_{i+1,M}$$

- **So, 4 rather than ~2M operations per hash evaluation (except i=1)**

- **Still have one potential problem – when M is large, fast arithmetic can be a problem (this is programming language dependent)**
  - **Particularly important for problems in base-26 rather than base-4)**

- **Can alleviate this problem with modulo operator…**

# Pattern search

**We have:**

$$H(S_i) = S_{i,1}\,4^{M-1} + S_{i,2}\,4^{M-2} + \ldots + S_{i,M-1}\,4 + S_{i,M}$$

$$H(S_{i+1}) = H(S_i)*4 - S_{i,1}4^{M} + S_{i+1,M}$$

- **Define** $h(S_i) = H(S_i) \bmod q$ **with** $q$ **a large prime number**
- **Use rules from modular arithmetic to simplify calculation of h:**

$$h(S_i+1) = (h(S_i)*4 \;\; - S_{i,1}\,(4^{M} \bmod q) + \; S_{i+1,M}) \bmod q$$

**How does this look in Python?**

# Pattern search

**We have:**

$$H(S_i) = S_{i,1}\, 4^{M-1} + S_{i,2}\, 4^{M-2} + \dots + S_{i,M-1}\, 4 + S_{i,M}$$

$$H(S_{i+1}) = H(S_i)*4 - S_{i,1}4^{M} + S_{i+1,M}$$

- **Define** $h(S_i) = H(S_i) \bmod q$ **with** $q$ **a large prime number**
- **Use rules from modular arithmetic to simplify calculation of h:**

$$h(S_i+1) = (h(S_i)*4 \;\; - S_{i,1}\,(4^{M} \bmod q) + \; S_{i+1,M}) \bmod q$$

```
bm = (4**m) % q

for ind in range(1,n-m+1):

    #Update fingerprint
    hi = (4*hi - int(S[ind-1])*bm + int(S[ind-1+m])) % q

    if hi==hp: #If fingerprints match, check if strings match
        if match(S[ind:ind+m],P): imatch.append(ind)
```

# Pattern search

$$h(S_i+1) = (h(S_i)*4 \;\; - S_{i,1}\,(4^M \bmod q) + \;\; S_{i+1,M}) \bmod q$$

```
bm = (4**m) % q

for ind in range(1,n-m+1):

    #Update fingerprint
    hi = (4*hi - int(S[ind-1])*bm + int(S[ind-1+m])) % q

    if hi==hp: #If fingerprints match, check if strings match
        if match(S[ind:ind+m],P): imatch.append(ind)
```

**Notes:**

- hp **is the hash for the pattern and has been pre-computed**
- **As has** hi **for** ind=0
- **Function** match **uses a naïve search to check if substrings** S **and** P **match**
- **Worst-case cost for Rabin-Karp is O(NM) (why?)**
- **Benefits from R-K will be seen when there are many "near-misses", i.e. many substrings 1st x letters match the pattern with x close to but less than m**

# Pattern search

- Rabin-Karp is one of several algorithms that have been developed for string matching
- It isn’t “too” old – introduced in 1987
- There are many applications outside of bioinformatics
  - Plagiarism detection
  - “Find” function in software applications

# Complex networks

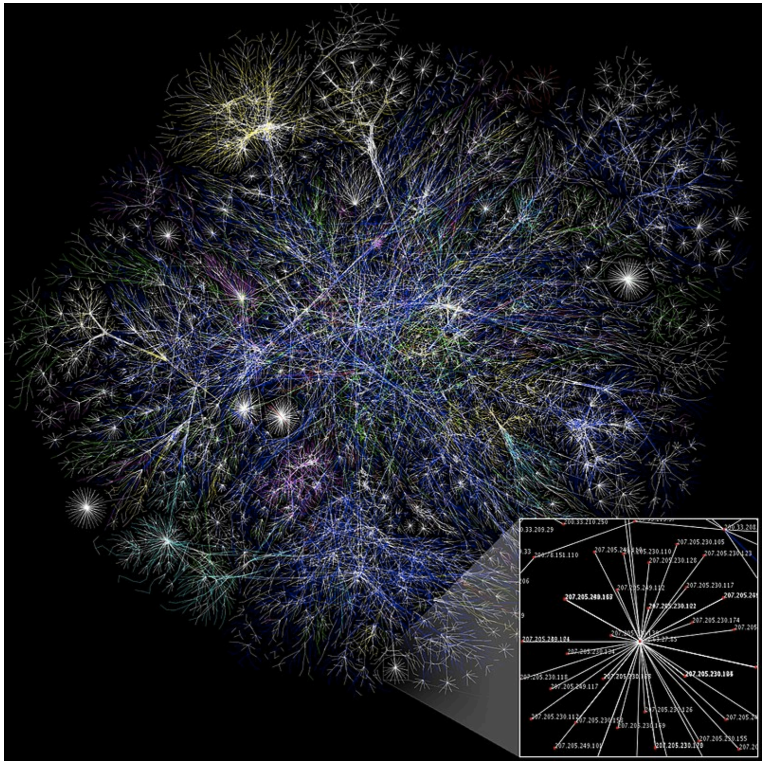
**Today:**

- **Basics of network science**
- **Using *networkx* package in python**

# Networks

**Examples of significant networks include:**

- Social networks
- World-wide web
- Internet
- Air transportation network
- Cellular network



https://en.wikipedia.org/wiki/Complex_network

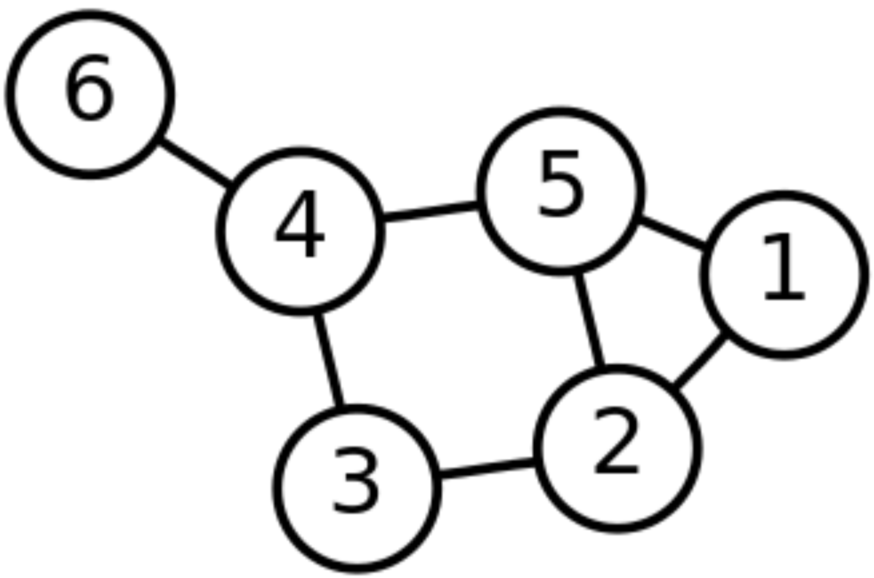
**The science of networks is an important, rapidly growing field**

# Networks: basics

- **A network has** N ***nodes*** **and** L **links between nodes**
- **Each node has a label, e.g.** 1, 2, …, N
- **Then a link between node i and j can be represented simply as (i, j)**

# Networks: basics

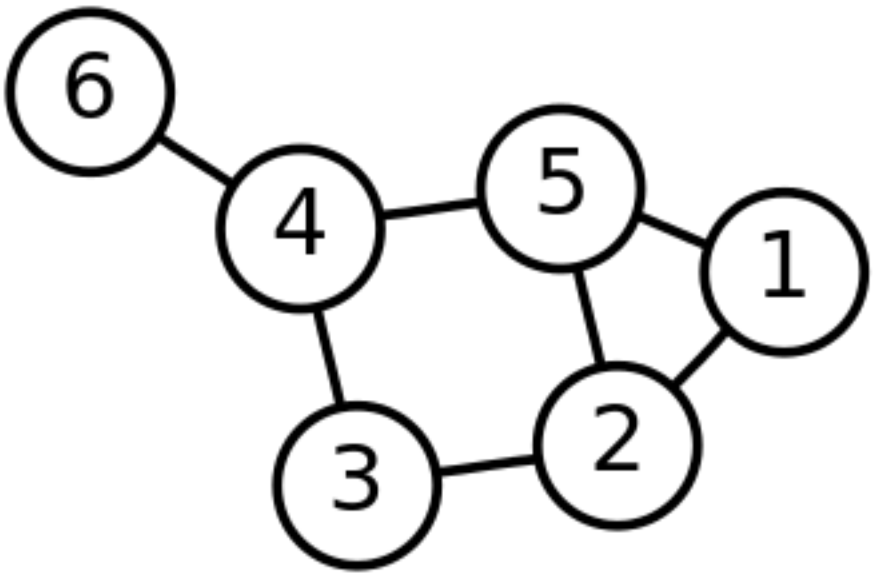
- **A network has** N ***nodes*** **and** L **links between nodes**
- **Each node has a label, e.g.** 1, 2, …, N
- **Then a link between node i and j can be represented simply as (i, j)**




**Example: 6 nodes, 7 links**
**Node one has two edges: (1,2) and (1,5)**

**The graph can be represented by the *adjacency matrix*, A**
**$A_{ij}$=1 if there is link between nodes i and j**

# Networks: basics



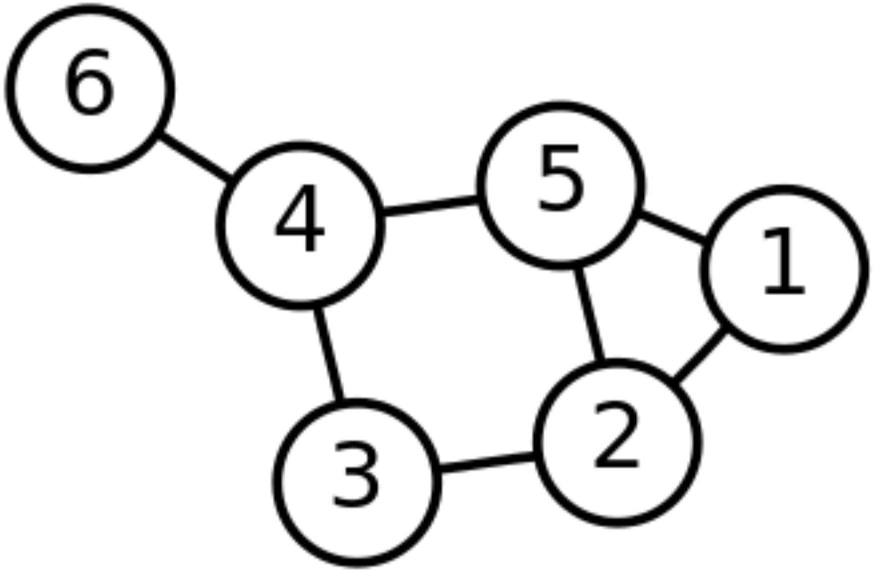

Example: 6 nodes, 7 links
Node one has two links: (1,2) and (1,5)

The graph can be represented by the *adjacency matrix*, A
$A_{ij}$=1 if there is link between nodes i and j

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

A is symmetric.

# Networks: basics




Example: 6 nodes, 7 links
Node one has two links: (1,2) and (1,5)

The graph can be represented by the *adjacency matrix*, A
$A_{ij}$=1 if there is link between nodes i and j

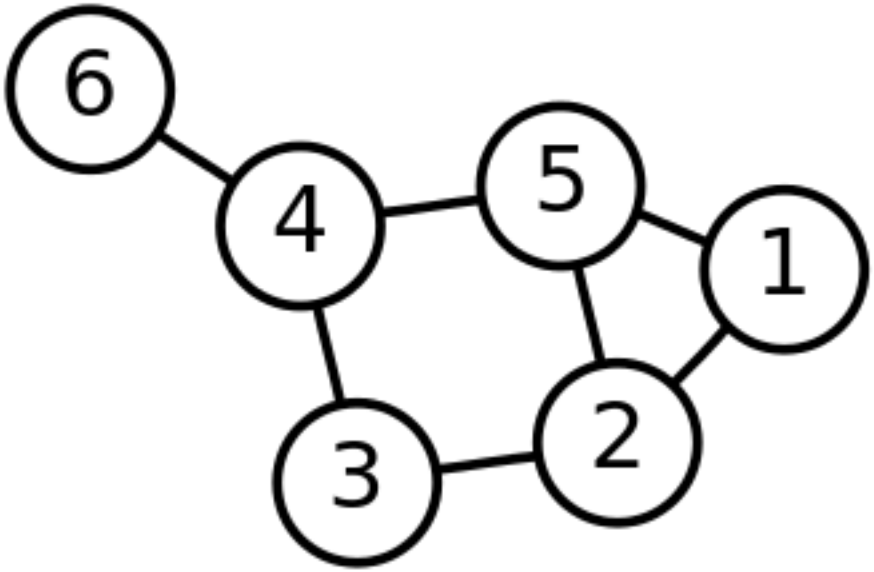
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

A is symmetric.

Can also represent connected portions of graph with edge list:

$$\begin{bmatrix} 1 & 1 & 2 & 2 & 3 & 4 & 4 \\ 2 & 5 & 3 & 5 & 4 & 5 & 6 \end{bmatrix}$$

# Networks: basics

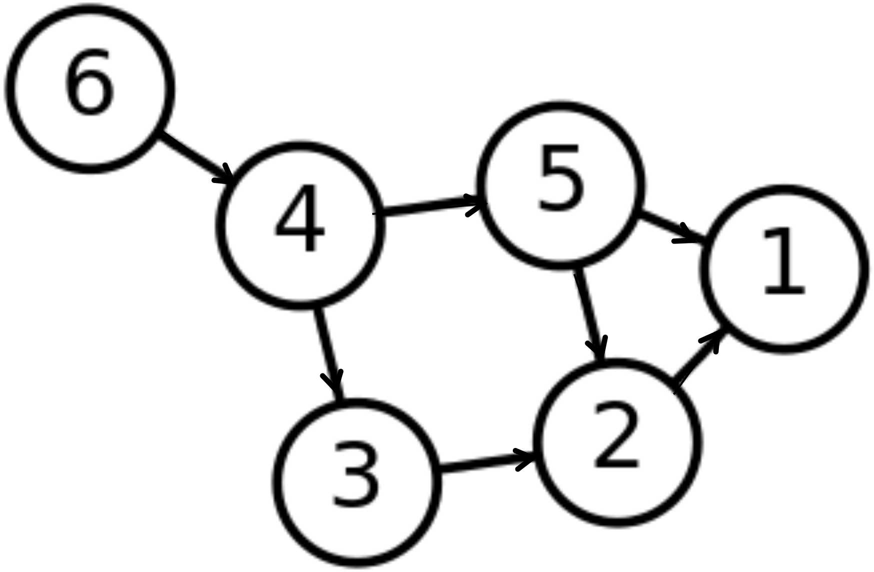



**The *degree* of a node is the the total number of links connected to it:**
$q_1 = 2$, $q_5 = 3$, …

**The *degree distribution*, P(q) is particularly important. P(q) is the fraction of nodes in the graph with degree = q**

**P(1) = 1/6, P(2) = 2/6, P(3) = 3/6**

# Networks: basics



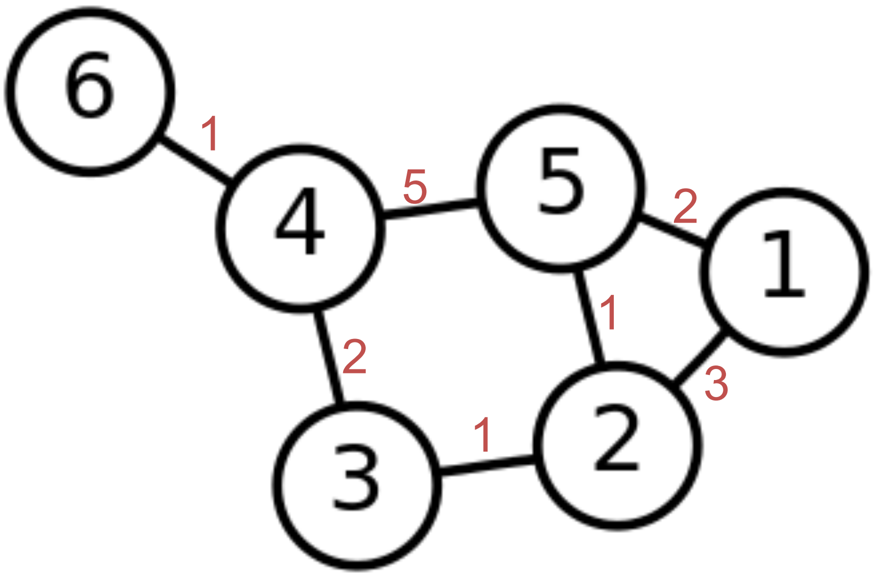

**Networks can also be *directed***

**Then**
**$A_{ij}$=1 if there is a link *to* i from j**

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

**A is *not* symmetric.**

# Networks: basics




**Networks can be weighted (e.g. transportation networks)**

$$\begin{pmatrix} 0 & 3 & 0 & 0 & 2 & 0 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 & 5 & 1 \\ 2 & 1 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

- **A is symmetric if network is undirected**
- **Network can be both directed and weighted**

# Networks: basics

| Network | Nodes | Links | Directed / Undirected | N | L | ‹K› |
|---|---|---|---|---|---|---|
| Internet | Routers | Internet connections | Undirected | 192,244 | 609,066 | 6.34 |
| WWW | Webpages | Links | Directed | 325,729 | 1,497,134 | 4.60 |
| Power Grid | Power plants, transformers | Cables | Undirected | 4,941 | 6,594 | 2.67 |
| Mobile-Phone Calls | Subscribers | Calls | Directed | 36,595 | 91,826 | 2.51 |
| Email | Email addresses | Emails | Directed | 57,194 | 103,731 | 1.81 |
| Science Collaboration | Scientists | Co-authorships | Undirected | 23,133 | 93,437 | 8.08 |
| Actor Network | Actors | Co-acting | Undirected | 702,388 | 29,397,908 | 83.71 |
| Citation Network | Papers | Citations | Directed | 449,673 | 4,689,479 | 10.43 |
| E. Coli Metabolism | Metabolites | Chemical reactions | Directed | 1,039 | 5,802 | 5.58 |
| Protein Interactions | Proteins | Binding interactions | Undirected | 2,018 | 2,930 | 2.90 |

Table 2.1

**Canonical Network Maps**

The basic characteristics of ten networks used throughout this book to illustrate the tools of network science. The table lists the nature of their nodes and links, indicating if links are directed or undirected, the number of nodes ($N$) and links ($L$), and the average degree for each network. For directed networks the average degree shown is the average in- or out-degrees $\langle k \rangle = \langle k_{in} \rangle = \langle k_{out} \rangle$ (see Equation (2.5)).

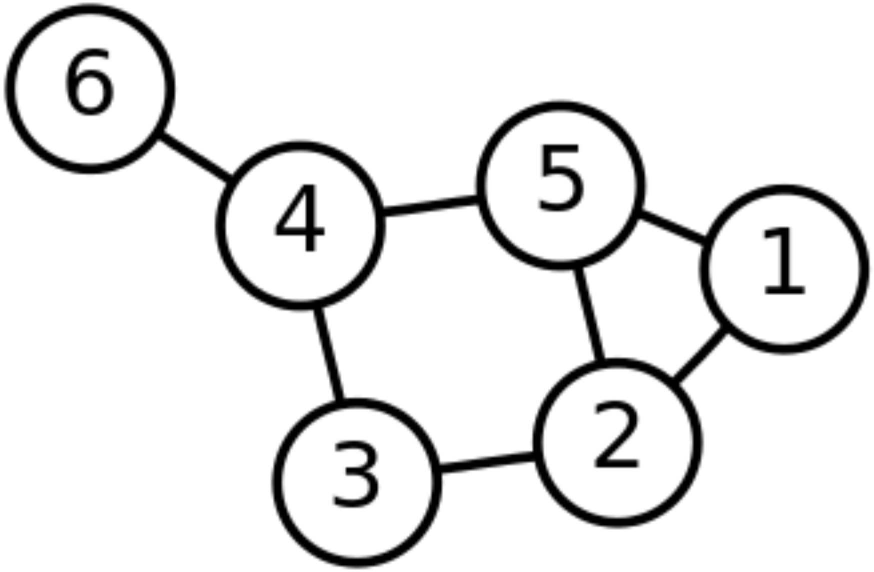
**Generally interested in large *complex* networks**

**Analysis can be complicated and expensive (classical example: computing shortest path between nodes)**

**Networkx package provides a suite of tools for working with complex networks**

**More generally: avoid writing own code whenever possible! Many powerful highly-efficient libraries are available**

# Networkx: basics

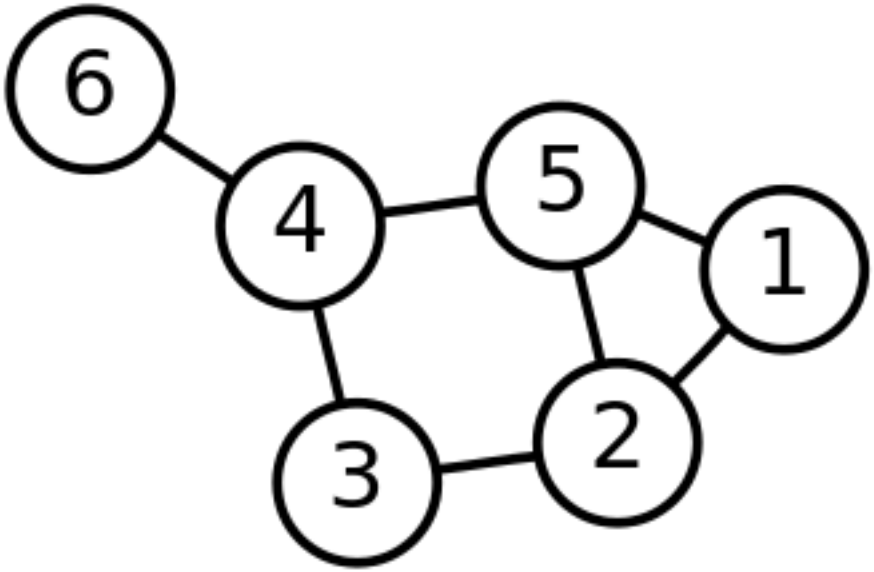



- **Let's work with this network in networkx**

**First, import the module, and initialize a graph:**

```
In [55]: import networkx as nx

In [56]: G = nx.Graph()
```

# Networkx: basics




- Let's work with this network in networkx

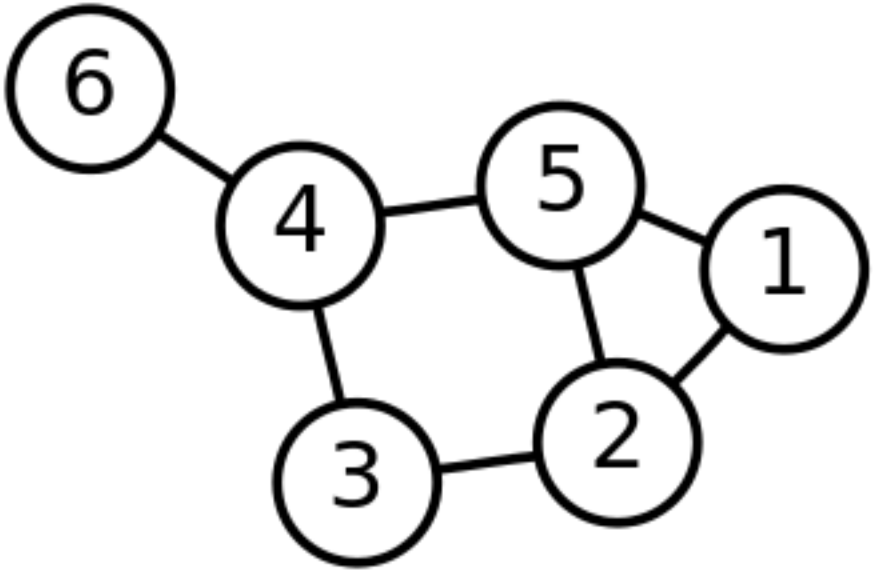
- First, import the module, and initialize a graph:

```
In [55]: import networkx as nx

In [56]: G = nx.Graph()
```

- There are numerous methods for building a graph

```
In [57]: G.add_edge(1,2)

In [58]: G.edges()
Out[58]: [(1, 2)]

In [59]: G.nodes()
Out[59]: [1, 2]
```

# Networkx: basics



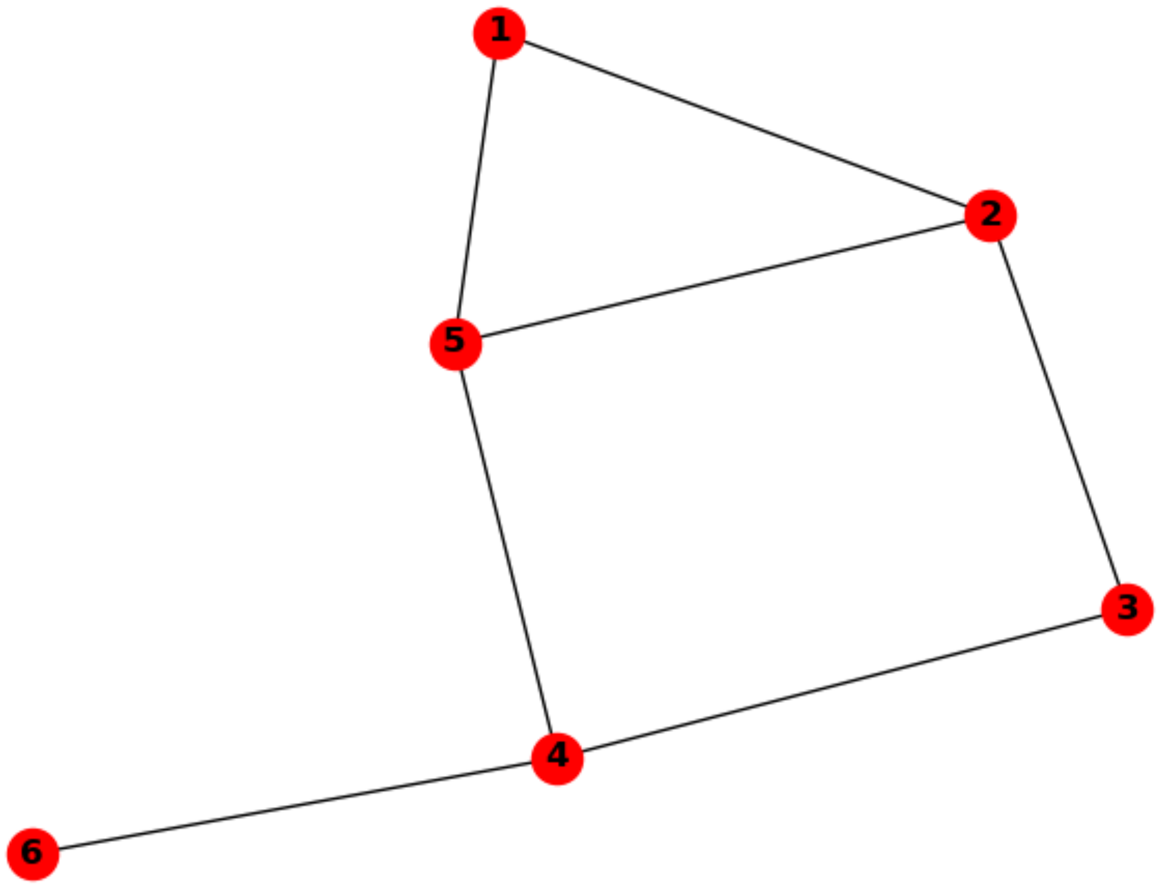

**Can add several edges (or nodes) at once:**

```
In [65]: e = [(1,5),(2,5),(2,3),(3,4),(4,5),(4,6)]

In [66]: G.add_edges_from(e)

In [67]: G.edges()
Out[67]: [(1, 2), (1, 5), (2, 3), (2, 5), (5, 4), (3, 4), (4, 6)]

In [68]: G.nodes()
Out[68]: [1, 2, 5, 3, 4, 6]
```

# Networkx: basics

**Use** nx.draw **to visualize the network:**

```
In [69]: figure()
Out[69]: <matplotlib.figure.Figure at 0x1515e3fef0>

In [70]: nx.draw(G, with_labels=True, font_weight='bold')
```



# Networkx: basics

## Can now analyze our graph:

```
In [74]: A = nx.adjacency_matrix(G)


In [75]: type(A)
Out[75]: scipy.sparse.csr.csr_matrix


In [76]: A.todense()
Out[76]:
matrix([[0, 1, 1, 0, 0, 0],
[1, 0, 1, 1, 0, 0],
[1, 1, 0, 0, 1, 0],
[0, 1, 0, 0, 1, 0],
[0, 0, 1, 1, 0, 1],
[0, 0, 0, 0, 1, 0]], dtype=int64)
```

# Networkx: basics

**Can now analyze our graph:**

```
In [78]: G.adjacency_list()
Out[78]: [[2, 5], [1, 3, 5], [1, 4, 2], [2, 4], [3, 5, 6], [4]]


In [79]: G.nodes()
Out[79]: [1, 2, 5, 3, 4, 6]
```

- **Adjacency list representation is much more efficient for sparse networks!**
- **Most complex networks are sparse**

# Networkx: basics

**Can now analyze our graph:**

```
In [83]: nx.degree_histogram?
Signature: nx.degree_histogram(G)
Docstring:
Return a list of the frequency of each degree value.
Returns
-------
hist : list
A list of frequencies of degrees.
The degree values are the index in the list.

In [84]: h = nx.degree_histogram(G)

In [85]: h
Out[85]: [0, 1, 2, 3]
```

- **Graph has one degree-1 node, two degree-2 nodes, and three degree-3 nodes**

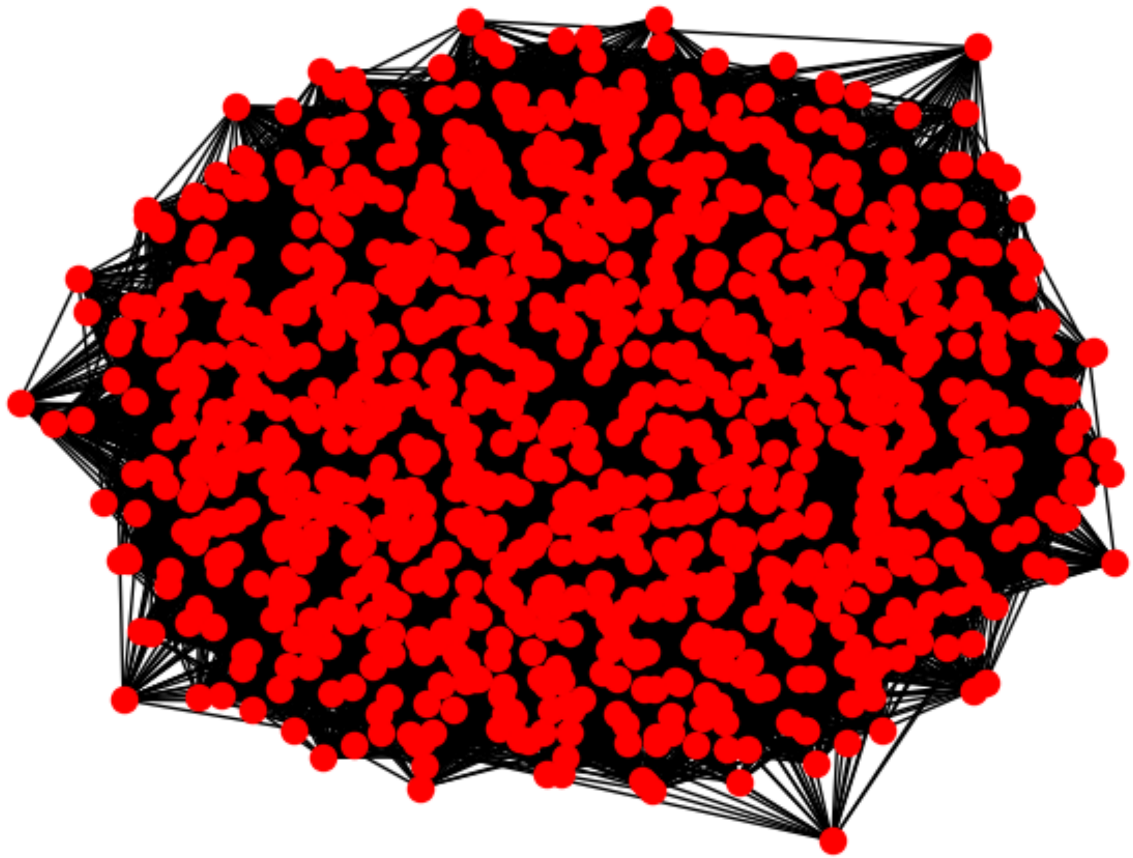


- **Degree distribution is more interesting for large networks:**

# Networkx: basics

**Erods-Renyi network N nodes, a link is placed between a pair of nodes with probability p:**

```
In [119]: Grandom = nx.gnp_random_graph(1000,0.05)

In [120]: nx.draw(Grandom,node_shape='.')
```

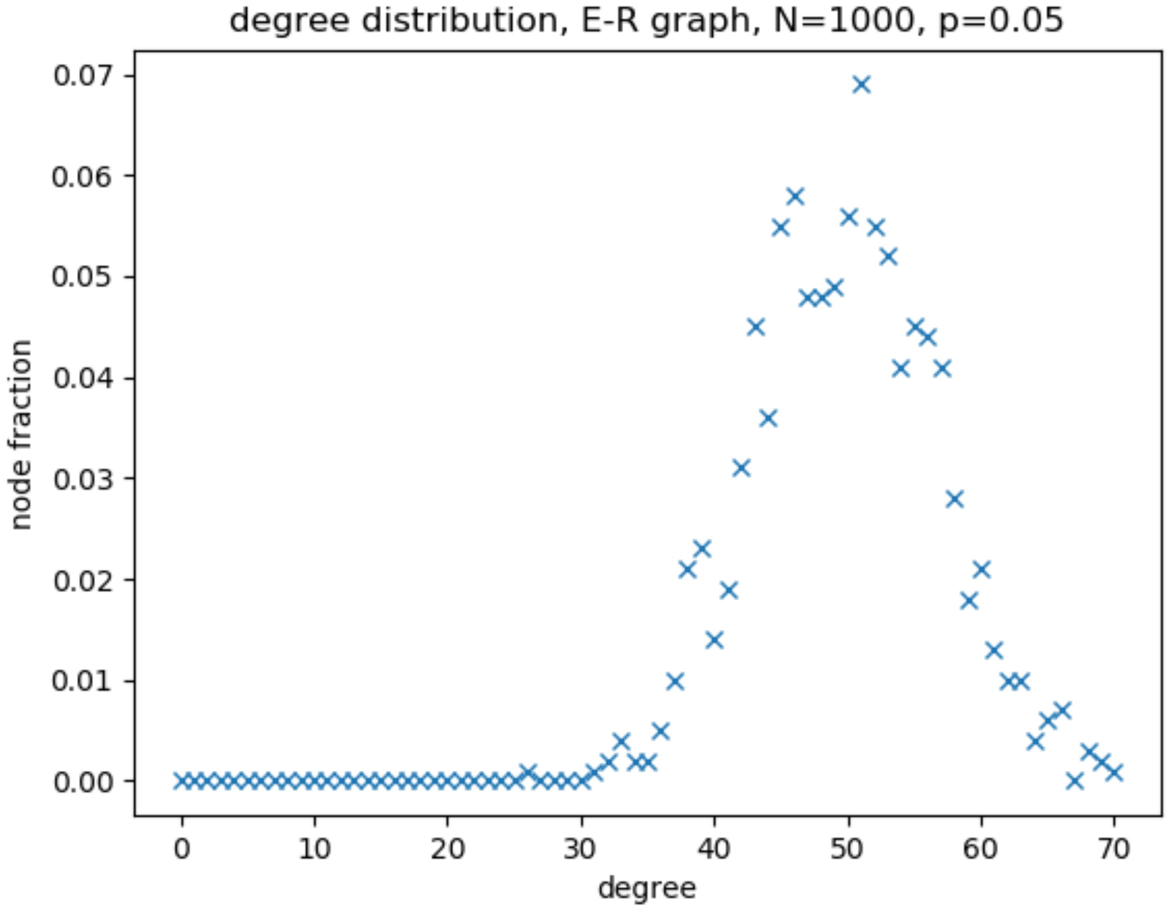


# Networkx: basics

**Erods-Renyi network**
**N nodes, a link is placed between a pair of nodes with probability p**

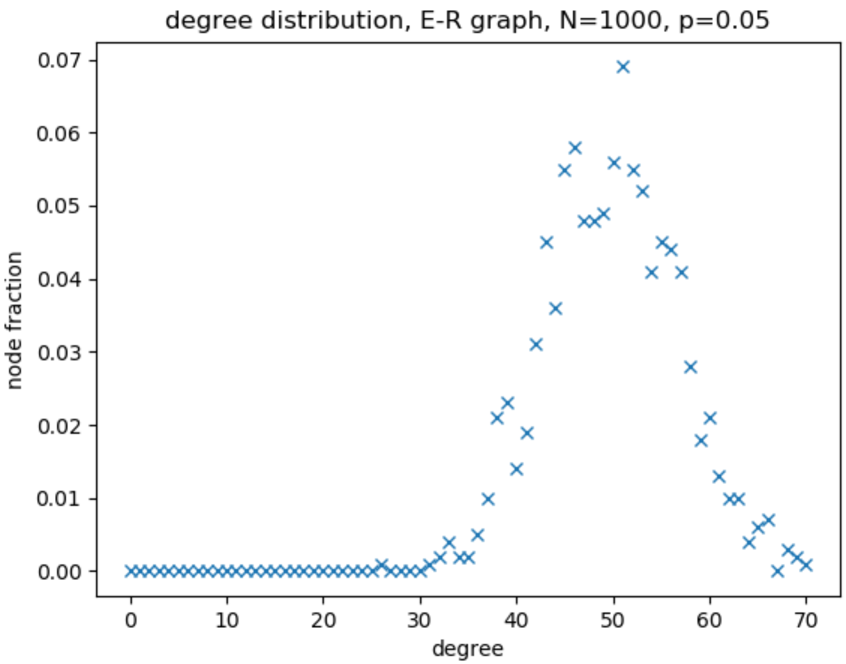
**Degree distribution follows the binomial distribution**

```
In [119]: Grandom = nx.gnp_random_graph(1000,0.05)

In [120]: nx.draw(Grandom,node_shape='.')
```



# Getting started with NetworkX

- **Degree distribution follows a binomial distribution:**




- Should compute degree distributions for several graphs (with fixed N,P) and average
- Generally, when there is randomness in the problem, statistics are the quantities of interest (mean, variance, etc…)
- For large degree, distribution decays away exponentially – most real complex networks have large-degree hubs

# Getting started with NetworkX

- **Two other important quantities are the *clustering coefficient* and *shortest path***
- **Clustering coefficient for node *i* with degree $q_i$:**

  $$C_i = \text{\# of links between neighbors}/(q_i/2*(q_i-1))$$

```
In [16]: nx.clustering(G,500)
Out[16]: 0.044096728307254626

In [17]: nx.clustering(G,100)
Out[17]: 0.06464646464646465

In [18]: nx.clustering(G,0)
Out[18]: 0.046457607433217191
```

**For $G_{N,P}$ graph, expect $C_i = P$**

# Getting started with NetworkX

- **Two other important quantities are the *clustering coefficient* and *shortest path***
- **Shortest path: find route between two nodes traversing fewest number of links**

```
In [20]: nx.shortest_path(G,source=0,target=500)
Out[20]: [0, 233, 15, 500]
```

# Getting started with NetworkX

- **Two other important quantities are the *clustering coefficient* and *shortest path***

- **Shortest path: find route between two nodes traversing fewest number of links**

```
In [20]: nx.shortest_path(G,source=0,target=500)
Out[20]: [0, 233, 15, 500]
```

**→ Very important in study *algorithms (lectures 7+8)***

**Notes: GNP graph is not a good model for large complex networks**

- **Degree distribution should include large-degree nodes, power-law decay for large q**

- **Clustering coefficient should be large and the average degree should be small**

- **Will consider a more-realistic model in this week's lab**

# Networkx: getting started

- **Read the online tutorial:**
  **https://networkx.github.io/documentation/stable/tutorial.html**
- **Browse through the online reference section:**
  **https://networkx.github.io/documentation/stable/reference/index.html**
- **Try out one or two graph generators**
- **Use networkx 2.x (I'm using 2.2)**

# Python notes

**Main differences between arrays and lists:**

**Lists are *flexible*: heterogeneous data, can grow or shrink, numerical calculations can be slow/cumbersome**

**Arrays: calculations are generally faster, but elements must be homogeneous, difficult to adjust size**