Algorithms

授課老師：張景堯

Quicksort

Chapter.4-Sorting and Searching

Quicksort(快速排序)

- In practice, the fastest *internal* sorting algorithm is Quicksort, which uses *partitioning* as its main idea.
  - Example: pivot about 10. (挑選10為基準)
  - Before: 10 17 12 6 19 23 8 5
  - After: 5 6 8 10 19 23 12 17
- Partitioning places all the elements less than the pivot in the *left* part of the array, and all elements greater than the pivot in the *right* part of the array. The pivot fits in the slot between. (比基準小的都會其在左邊、大的都在右邊)
- Note that the pivot element ends up in the correct place in the total order!
  (每做完一輪基準的位置就是整體排序後的所在位置)

Partitioning the Elements 元素分割

- We can partition an array about the pivot in one linear scan, by maintaining three sections: <pivot, >pivot, and unexplored.
- https://upload.wikimedia.org/wikipedia/commons/9/9c/Quicksort-example.gif
- https://www.youtube.com/watch?v=ywWBy6J5gz8
- As we scan from left to right, we move the left bound to the right when the element is less than the pivot and swap it with the element of *left bound*, otherwise we stay the left bound at *rightmost* element of less than pivot and keep scanning. (從基準之左到右掃描，當掃描到小於基準就右移左邊界並與左邊界所在元素交換，否則就將邊界留在基準本身或小於基準的最右邊元素上不動，繼續向右掃描)

Why Partition?

- Since the partitioning step consists of at most n swaps, takes time linear in the number of keys. But what does it buy us?
  1. The pivot element ends up in the position it retains in the final sorted order. (每一輪的基準所在就是排序後的位置)
  2. After a partitioning, no element flops to the other side of the pivot in the final sorted order. (每一輪分割後，元素就會留在該區不會跳到另一邊)
- *Thus we can sort the elements to the left of the pivot and the right of the pivot independently, giving us a recursive sorting algorithm! (這樣才能讓我們以遞迴做法，獨立處理基準左邊與右邊而不會相互影響)*

Quicksort

```
def quicksort(nums:list, low:int, high:int):
    if low < high:
        pivot_idx = partition(nums, low, high)
        quicksort(nums, low, pivot_idx-1)
        quicksort(nums, pivot_idx+1, high)


nums = [3,0,1,8,7,2,5,4,9,6]
quicksort(nums, 0, len(nums)-1)
print(nums)
```

Partition Implementation

```python
def partition(nums:list, low:int, high:int)->int:
    pivot = nums[low]
    leftwall = low
    for i in range(low+1,high+1):
        if nums[i] < pivot:
            leftwall += 1
            nums[i], nums[leftwall] = nums[leftwall], nums[i]
    nums[low], nums[leftwall] = nums[leftwall], nums[low]
    return leftwall
```

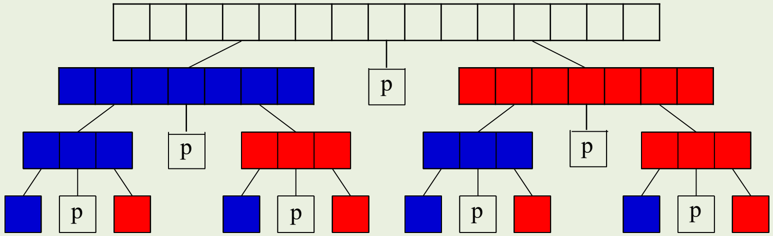
Partition (Dancing Version)

```python
def partition(nums:list, low:int, high:int)->int:
    pivot, hat, step = low, high, -1
    while pivot != hat:
        if step == -1 and nums[pivot] > nums[hat]:
            nums[pivot], nums[hat] = nums[hat], nums[pivot]
            pivot, hat, step = hat, pivot, 1
        elif step == 1 and nums[pivot] < nums[hat]:
            nums[pivot], nums[hat] = nums[hat], nums[pivot]
            pivot, hat, step = hat, pivot, -1
        hat += step
    return pivot
```

Best Case for Quicksort

- Since each element ultimately ends up in the correct position, the algorithm correctly sorts. But how long does it take?
- The best case for *divide-and-conquer* algorithms comes when we split the input as evenly as possible. Thus in the best case, each subproblem is of size $n/2$.
- The partition step on each subproblem is linear in its size.
- Thus the total effort in partitioning the 2^k problems of size $\frac{n}{2^k}$, $k = 0 \sim \log_2 n$ is $O(n)$

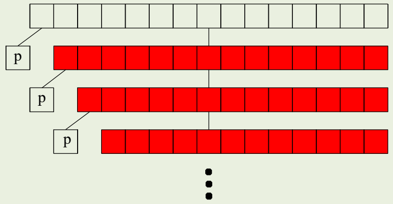
Best Case Recursion Tree



- The total partitioning on each level is $O(n)$, and it take $\log n$ levels of perfect partitions to get to single element subproblems. When we are down to single elements, the problems are sorted. Thus the total time in the best case is $O(n\log n)$.
(每一層分割區合起來最多n個元素，總共有$\log n$分割層)

Worst Case for Quicksort

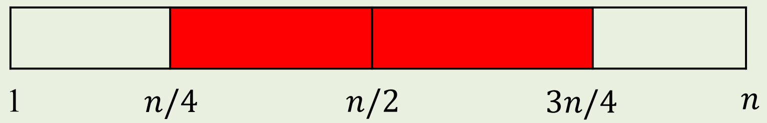
- Suppose instead our pivot element splits the array as unequally as possible. Thus instead of $n/2$ elements in the smaller half, we get zero, meaning that the pivot element is the biggest or smallest element in the array.



- Now we have $n-1$ levels, instead of $\log n$, for a worst case time of $\Theta(n^2)$, since the first $n/2$ levels each have $\geq n/2$ elements to partition.
- To justify its name, Quicksort had better be good in the average case. Showing this requires some intricate analysis. The divide and conquer principle applies to real life. If you break a job into pieces, make the pieces of equal size!

Intuition: The Average Case for Quicksort

- Suppose we pick the pivot element at random in an array of n keys.



- Half the time, the pivot element will be from the center half of the sorted array.
- Whenever the pivot element is from positions $n/4$ to $3n/4$, the larger remaining subarray contains at most $3n/4$ elements.

How Many Good Partitions

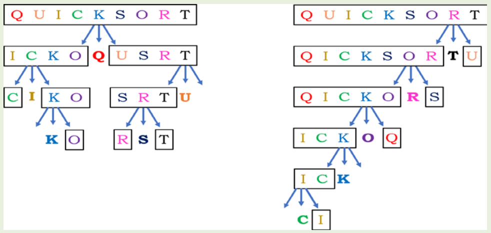
- If we assume that the pivot element is always in this range, what is the maximum number of partitions we need to get from n elements down to 1 element?

$$\left(\frac{3}{4}\right)^{l} \cdot n = 1 \rightarrow n = \left(\frac{4}{3}\right)^{l}$$

- Therefore $l = \log_{\frac{4}{3}}(n) \approx 2.4 \log_2(n)$ good partitions suffice.

How Many Bad Partitions?

- How often when we pick an arbitrary element as pivot will it generate a decent partition?
- Since any number ranked between $n/4$ and $3n/4$ would make a decent pivot, we get one half the time on average.
- If we need $2\log n$ levels of decent partitions to finish the job, and half of random partitions are decent, then on average the recursion tree to quicksort the array has $\approx 4\log n$ levels.
- Since $O(n)$ work is done partitioning on each level, the average time is $O(n\log n)$.



(因為有一半的機會去選到不錯的基準
使得分割層數為$2\log n$，所以平均來看也有$4\log n$ 層)

Randomized Quicksort

- Suppose you are writing a sorting program, to run on data given to you by your worst enemy. Quicksort is good on average, but bad on certain worst-case instances.
- If you used Quicksort, what kind of data would your enemy give you to run it on? Exactly the worst-case instance, to make you look bad.
- But suppose you picked the pivot element at *random*.
- Now your enemy cannot design a worst-case instance to give to you, because no matter which data they give you, you would have the same probability of picking a good pivot!

(如果基準是隨機選取的，反而能避開刻意設計的最差例)

Randomized Guarantees

- Randomization is a very important and useful idea. By either picking a random pivot or scrambling the permutation before sorting it, we can say:

  "With high probability, randomized quicksort runs in $\Theta(n \log n)$ time."

- Where before, all we could say is:

  "If you give me random input data, quicksort runs in expected $\Theta(n \log n)$ time."

Importance of Randomization

- Since the time bound how does not depend upon your input distribution, this means that unless we are *extremely* unlucky (as opposed to ill prepared or unpopular) we will certainly get good performance.
- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.
- The worst-case is still there, but we almost certainly won't see it.

(如果演算法有不錯的平均時間複雜度，即便有最差時間複雜度，還是可以運用隨機化策略來降低碰到最差狀況)

Pick a Better Pivot

- Having the worst case occur when they are sorted or almost sorted is *very bad*, since that is likely to be the case in certain applications.
- To eliminate this problem, pick a better pivot:
    1. Use the middle element of the subarray as pivot.
    2. Use a random element of the array as the pivot.
    3. Perhaps best of all, take the median of three elements(first, last, middle) as the pivot. Why should we use median instead of the mean?
- Whichever of these three rules we use, the worst case remains $O(n^2)$.

Is Quicksort *really* faster than Heapsort?

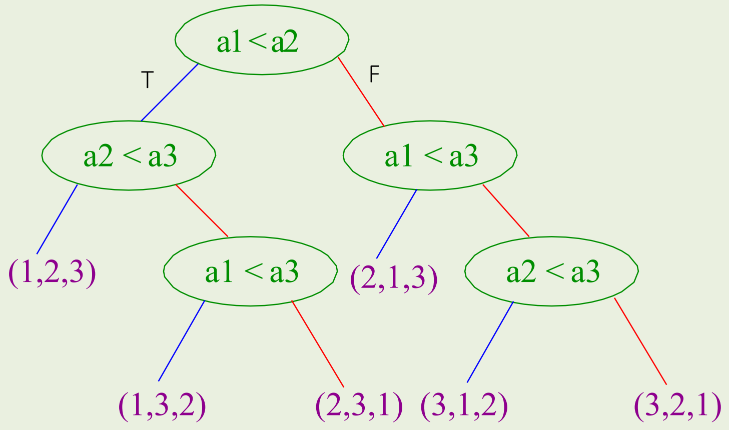
- Since Heapsort is $\Theta(n \log n)$ and selection sort is $\Theta(n^2)$, there is no debate about which will be better for decent-sized files. When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort.
- The primary reason is that the operations in the innermost loop are simpler.
- Since the difference between the two programs will be limited to a multiplicative *constant factor*, the details of how you program each algorithm will make a big difference.

Linear Sorting/Bucketsort

Chapter.4-Sorting and Searching

Can we sort in $o(n \log n)$?

- Any comparison-based sorting program can be thought of as defining a decision tree of possible executions.
- Running the same program twice on the same permutation causes it to do exactly the same thing, but running it on different permutations of the same data causes a different sequence of comparisons to be made on each.
- Claim: the height of this decision tree is the worst-case complexity of sorting.



Lower Bound Analysis

- Since any two different permutations of n elements requires a different sequence of steps to sort, there must be at least $n!$ different paths from the root to leaves in the decision tree.
- Thus there must be at least $n!$ different leaves in this binary tree.
- Since a binary tree of height h has at most 2^h leaves, we know $n! \le 2^h$, or $h \ge \lg(n!)$.
- By inspection $n^n > n! > \left(\frac{n}{2}\right)^{\frac{n}{2}}$, since the last $\frac{n}{2}$ terms of the product are each greater than $\frac{n}{2}$. Thus

$$\boxed{\log(n!) > \log\left(\frac{n}{2}\right)^{\frac{n}{2}}} = \frac{n}{2}\log\left(\frac{n}{2}\right)$$

$= \Omega(n \log n)$

⇨ $\Theta(n \log n)$

(n的階乘裡至少有一半數字比中位數大)

Stirling's Approximation

- By Stirling's approximation, a better bound is $n! > \left(\frac{n}{e}\right)^n$ where $e = 2.718$.

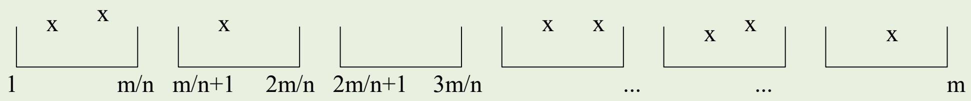
$$h \geq \lg \left(\frac{n}{e}\right)^n = n \lg n - n \lg e = \Omega(n \lg n)$$

Non-Comparison-Based Sorting

- All the sorting algorithms we have seen assume binary comparisons as the basic primitive, questions of the form “is x before y?”.
- But how would you sort a deck of playing cards?
- Most likely you would set up 13 piles and put all cards with the same number in one pile.
- With only a constant number of cards left in each pile, you can use insertion sort to order by suite and concatenate everything together.
- If we could find the correct pile for each card in constant time, and each pile gets $O(1)$ cards, this algorithm takes $O(n)$ time.

Bucketsort

- Suppose we are sorting m numbers from 1 to m, where we know the numbers are approximately uniformly distributed.
- We can set up n buckets, each responsible for an interval of m/n numbers from 1 to m



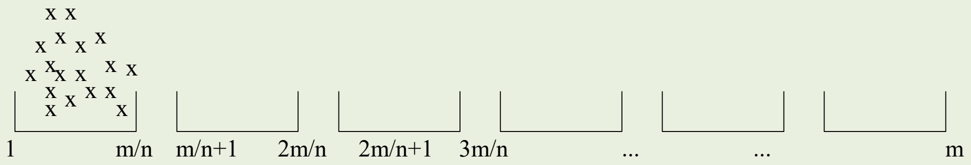
- Given an input number x, it belongs in bucket number $\left\lceil \frac{xn}{m} \right\rceil$.
- If we use an array of buckets, each item gets mapped to the right bucket in $O(1)$ time.
- Similar to: Counting Sort, Radix Sort
- Think about Direct-Address Table.

Bucketsort Analysis

- With uniformly distributed keys, the expected number of items per bucket is 1. Thus sorting each bucket takes $O(1)$ time!
- The total effort of bucketing, sorting buckets, and concatenating the sorted buckets together is $O(n)$.
- What happened to our $\Omega(n \lg n)$ lower bound!

Worst-Case vs. Assumed-Case

- Bad things happen to bucketsort when we assume the *wrong distribution*.



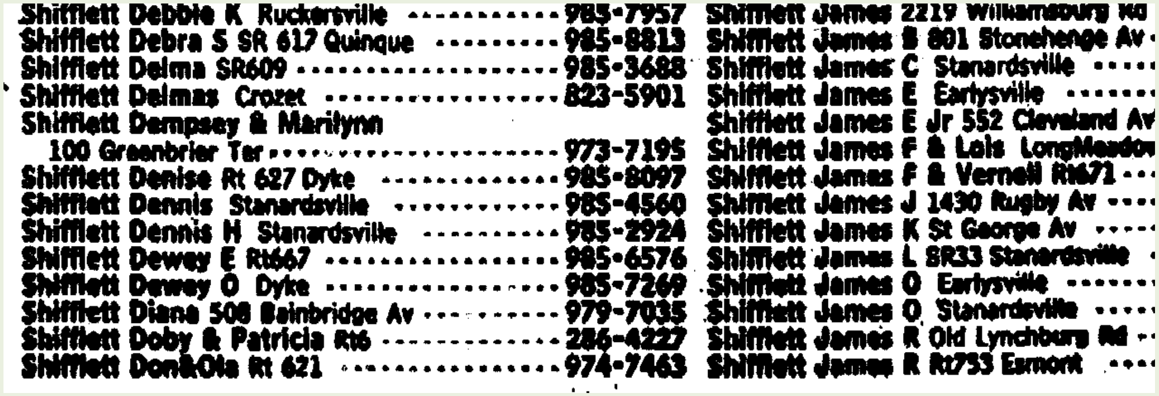
- We might spend linear time distributing our items into buckets and learn *nothing*.
- Problems like this are why we worry about the worst-case performance of algorithms!

Real World Distributions

- The worst case "shouldn't" happen if we understand the distribution of our data.
- Consider the distribution of names in a telephone book.
  - Will there be a lot of Skiena's?
  - Will there be a lot of Smith's?
  - Will there be a lot of Shifflett's?
- Either make *sure* you understand your data, or <u>use a good worst-case or randomized algorithm</u>!

The Shifflett's of Charlottesville

- For comparison, note that there are seven Shifflett's (of various spellings) in the 1000 page Manhattan telephone directory.



Shifflett Debbie K Ruckersville	985-7957	Shifflett James 2219 Williamsburg Rd
Shifflett Debra S SR 617 Quinque	985-8813	Shifflett James B 801 Stonehenge Av
Shifflett Delma SR609	985-3688	Shifflett James C Stanardsville
Shifflett Delmas Crozet	823-5901	Shifflett James E Earlysville
Shifflett Dempsey & Marilynn		Shifflett James E Jr 552 Cleveland Av
100 Greenbrier Ter	973-7195	Shifflett James F & Lois LongMeadow
Shifflett Denise Rt 627 Dyke	985-8097	Shifflett James F & Vernell Rt671
Shifflett Dennis Stanardsville	985-4560	Shifflett James J 1430 Rugby Av
Shifflett Dennis H Stanardsville	985-2924	Shifflett James K St George Av
Shifflett Dewey E Rt667	985-6576	Shifflett James L SR33 Stanardsville
Shifflett Dewey O Dyke	985-7269	Shifflett James O Earlysville
Shifflett Diana 508 Bainbridge Av	979-7035	Shifflett James O Stanardsville
Shifflett Doby & Patricia Rt6	286-4227	Shifflett James R Old Lynchburg Rd
Shifflett Don&Ola Rt 621	974-7463	Shifflett James R Rt753 Esmont

(即便是電話簿上的姓名，看似可用同一套分布概念理解資料，但其實不同地區的資料分布情形也是不盡相同)

Non-Comparison Sorts Don't Beat the Bound

- Radix sort works by partitioning on the lowest order characters first, maintaining this order to break ties.
- It takes time $O(nm)$ to sort n strings of length m, or time *linear* in the input size!
- But m must be $\Omega(\log n)$ before the strings are all distinct!
- Sorting n arbitrary, distinct keys cannot be done better than $\Theta(n \log n)$.

- https://www.youtube.com/watch?v=ibtN8rY7V5k

Graph Data Structures

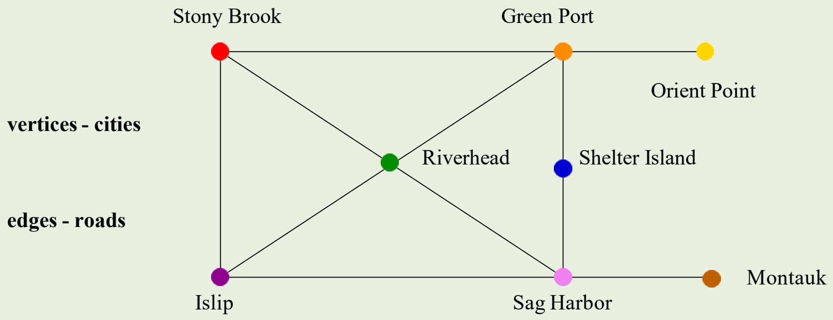
Chapter.5-Graph Traversal

Graphs

- Graphs are one of the unifying themes of computer science. A graph $G = (V, E)$ is defined by a set of vertices V, and a set of edges E consisting of ordered or unordered pairs of vertices from V.

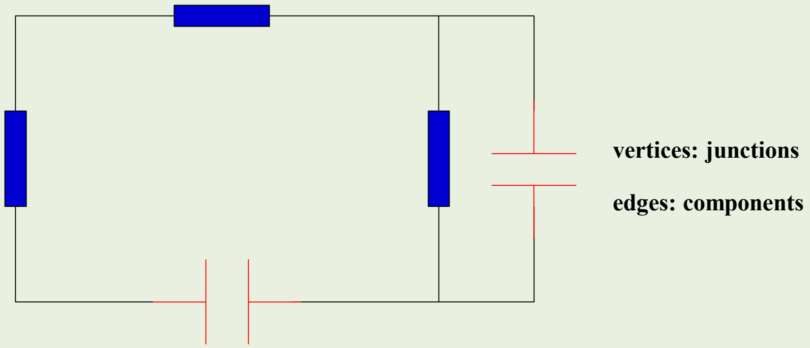
Road Networks

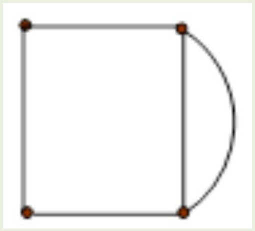
- In modeling a road network, the vertices may represent the *cities* or *junctions*, certain pairs of which are connected by *roads*/edges.



Electronic Circuits

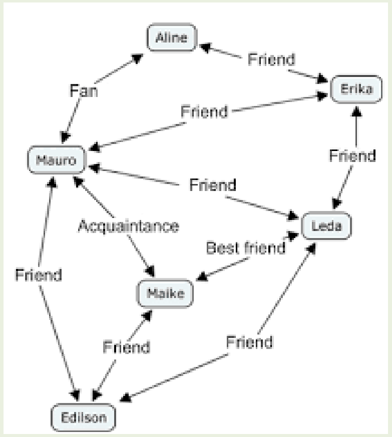
- In an electronic circuit, with *junctions* as vertices as *components* as edges.

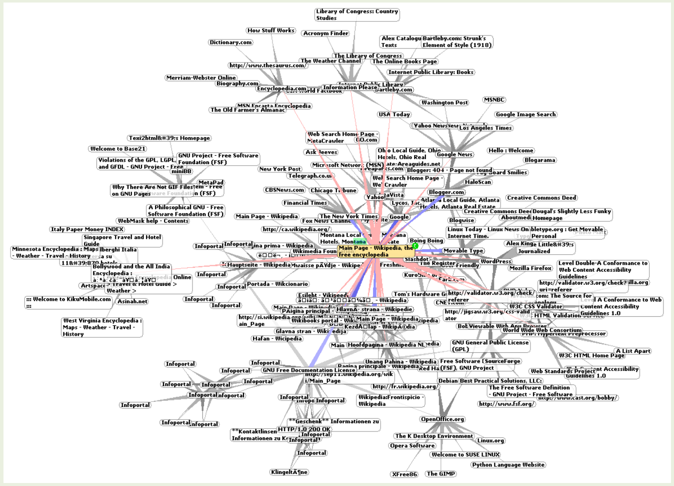


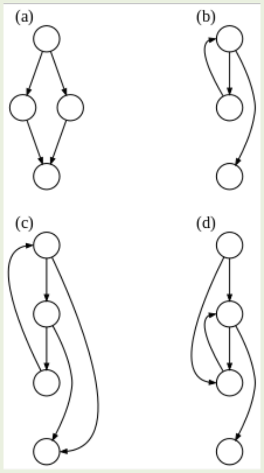


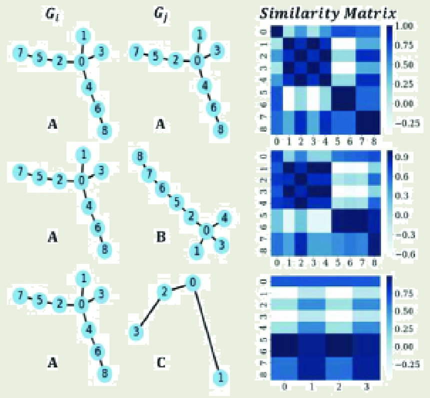
Other Graphs/Networks

- Social networks
- The World Wide Web
- Control flow within a computer program
- Pairwise similarities between items









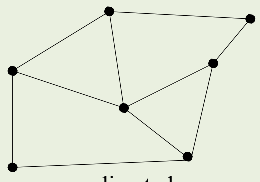
Flavors of Graphs

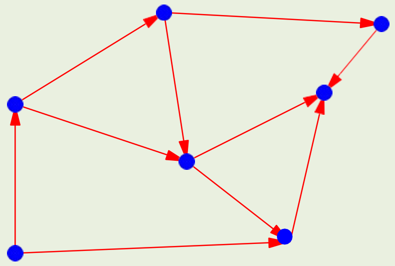
- The first step in any graph problem is determining which *flavor* of graph you are dealing with.
- Learning to talk the talk is an important part of walking the walk.
- The flavor of graph has a big impact on which algorithms are appropriate and efficient.

(解決圖論相關問題前，先弄清楚要面對的是哪種樣式的圖)

Directed vs. Undirected Graphs (有向圖 vs. 無向圖)

- A graph $G = (V, E)$ is undirected if edge $(x, y) \in E$ implies that (y, x) is also in E.

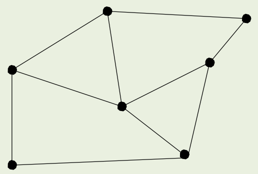

undirected

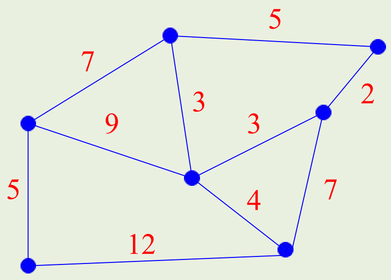

directed

- Road networks *between* cities are typically undirected. Street networks *within* cities are almost always directed because of one-way streets.
- Most graphs of graph-theoretic interest are **undirected**.
- [Zuvio]Which kind(directed or undirected) of graph are in Facebook and Twitter?

Weighted vs. Unweighted Graphs (加權圖 vs. 無加權圖)

- In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.


unweighted

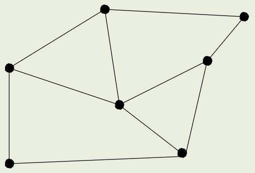


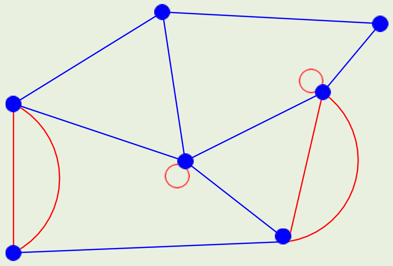
weighted

- The edges of a road network graph might be weighted with their length, drive-time or speed limit.
- In *unweighted* graphs, there is no cost distinction between various edges and vertices.

Simple vs. Non-simple Graphs

- Certain types of edges complicate working with graphs. A self-loop is an edge (x, x) involving only one vertex.
- An edge (x, y) is a *multi-edge* if it occurs more than once In the graph.

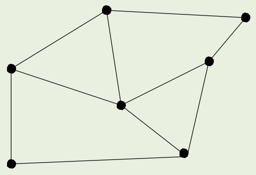

simple

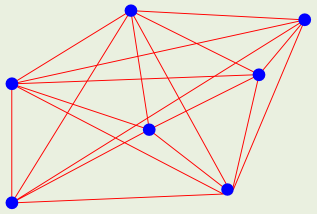

non−simple

- Any graph which avoids these structures is called simple.
- Are you your own friend?

Sparse vs. Dense Graphs (稀疏圖 vs.密集圖)

- Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.

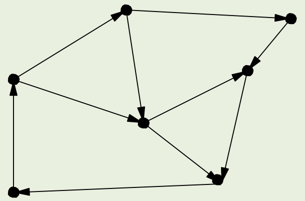

sparse

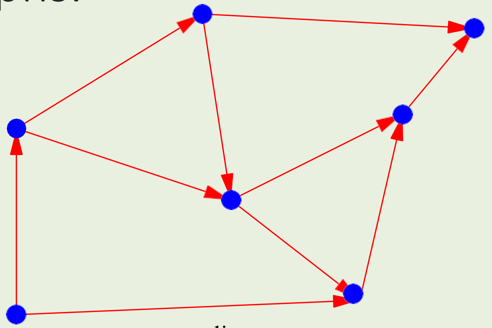

dense

- Graphs are usually sparse due to application-specific constraints. Road networks must be sparse because of road junctions.
- Typically dense graphs have a quadratic number $\binom{n}{2}$ of edges while sparse graphs are linear in size.

Cyclic vs. Acyclic Graphs (循環圖 vs. 非循環圖)

- An *acyclic* graph does not contain any cycles. *Trees* are connected acyclic *undirected* graphs.

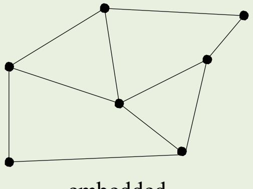

cyclic

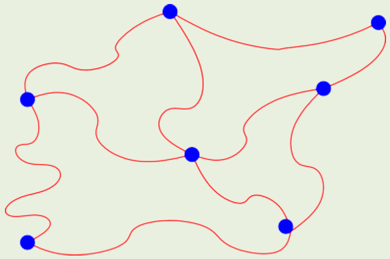

acyclic

- Directed acyclic graphs are called *DAG*s. They arise naturally in scheduling problems, where a directed edge (x, y) indicates that x must occur before y.

Embedded vs. Topological Graphs (嵌入圖 vs. 拓樸圖)

- A graph is *embedded* if the vertices and edges have been assigned geometric positions.


embedded

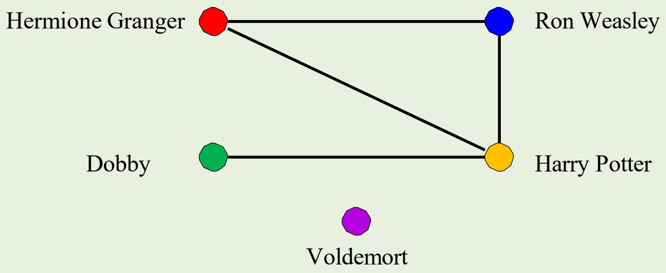

topological

- Is there a drawing of the graph that matters, as in the road and electrical circuit networks?
- Example: TSP or Shortest path on points in the plane.
- Example: Grid graphs and planar graphs.

(即是否帶有幾何位置意義，如：路網圖 vs. 朋友關係圖)

The Friendship Graph

- Consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.



- This graph is well-defined on any set of people: Wenshan, Taipei, or the world.
- What questions might we ask about the friendship graph?

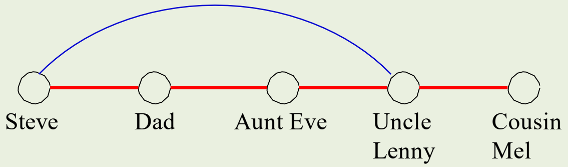
If I am your friend, does that mean you are my friend?

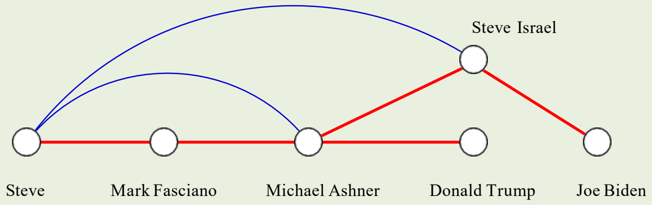
- A graph is *undirected* if (x, y) implies (y, x). Otherwise the graph is directed.
    - The “heard-of” graph is directed since countless famous people have never heard of me!
    - The “handshake-with” graph is presumably undirected, since it requires a partner.

    （耳聞跟握過手分別可繪製成有向和無向圖）

Am I linked by some chain of friends to a particular celebrity?

- A *path* is a sequence of edges connecting two vertices. Since *Mel Brooks* is my father's-sister's-husband's cousin, there is a path between me and him!





- If I were trying to impress you with how tight I am with Mel Brooks, I would be much better off saying that Uncle Lenny knew him than to go into the details of how connected I am to Uncle Lenny.
- Thus we are often interested in the *shortest path* between two nodes. (一般不會講那麼長的關係，而說是我姑丈表親)

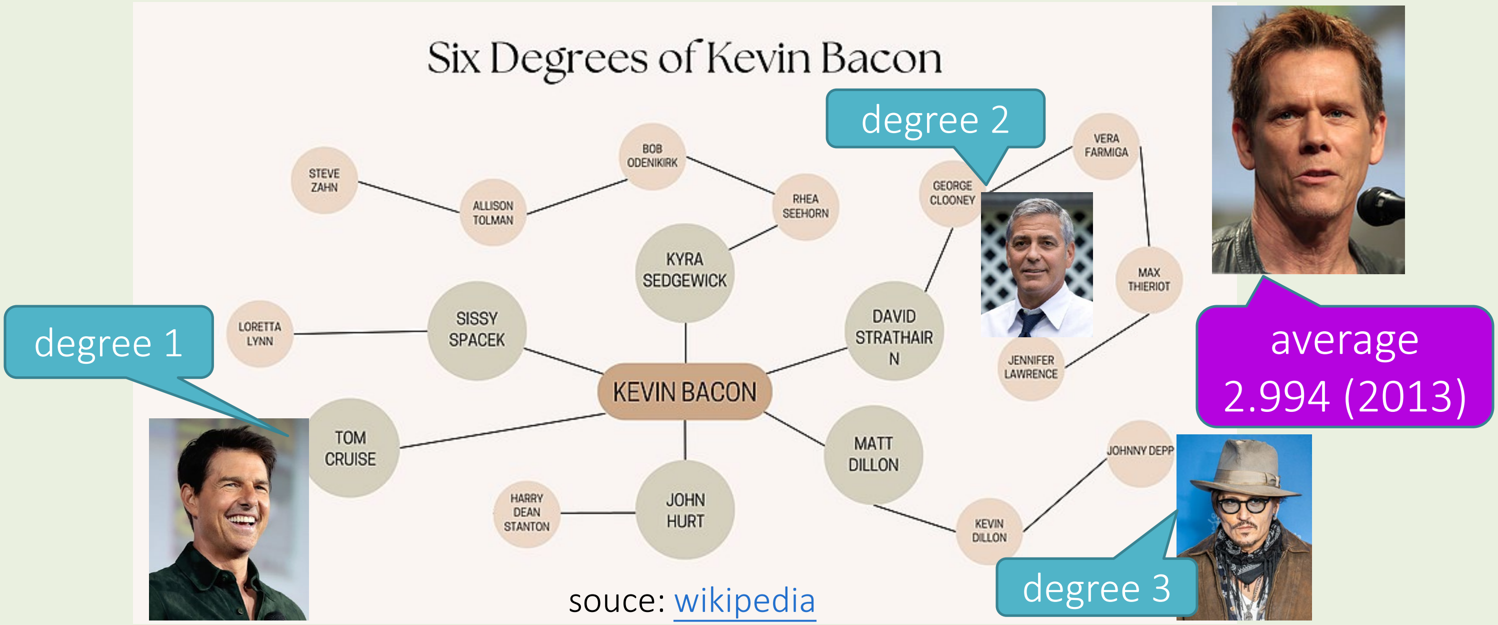
Is there a path of friends between any two people?

- A graph is *connected* if there is a path between any two vertices.
- A directed graph is *strongly connected* if there is a directed path between any two vertices.
- The notation of *six degrees of separation* presumes the world’s social network is a connected graph.

(六度分隔理論：哈佛大學心理學教授斯坦利·米爾格拉姆於1967年根據這個概念做過一次連鎖信實驗，嘗試證明平均只需要6步就可以聯繫任何兩個互不相識的人。)

Six Degrees of Kevin Bacon

- Connect an actor to another actor via a film that both actors have appeared in together, and find the shortest path that ultimately leads to prolific American actor Kevin Bacon.



souce: wikipedia

Who has the most friends?

- The *degree* of a vertex is the number of edges adjacent to it.

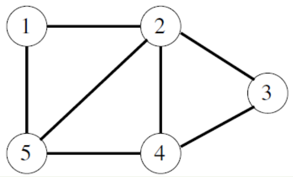
What is the largest clique(最大集團)?

- A social clique is a group of mutual friends who all hang around together. （集團中每個成員幾乎互相都是朋友）
- A graph theoretic *clique*(集團型) is a complete subgraph, where each vertex pair has an edge between them. Cliques are the <u>densest possible subgraphs</u>.
- Within the friendship graph, we would expect that large cliques correspond to workplaces, neighborhoods, religious organizations, schools, and the like.

Data Structures for Graphs: Adjacency Matrix (相鄰矩陣)

- There are two main data structures used to represent graphs. We assume the graph $G = (V, E)$ contains n vertices and m edges.
- We can represent G using an $n \times n$ matrix M , where element $M[i, j]$ is, say, **1**, if (i, j) is an edge of G, and **0** if it isn't. It may use excessive space for graphs with many vertices and relatively few edges, however.

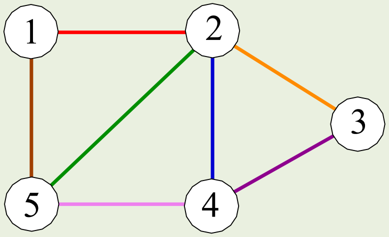
$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

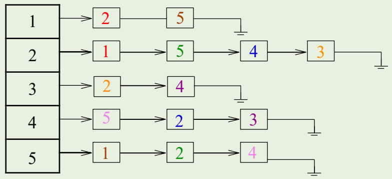


- Can we save space if (1) the graph is undirected? (2) if the graph is sparse?

Adjacency Lists(相鄰串列)

■An adjacency list consists of a $N \times 1$ array of pointers, where the ith element points to a linked list of the edges incident on vertex i.





■To test if edge (i, j) is in the graph, we search the ith list for j, which takes $O(d_i)$, where d_i is the degree of the ith vertex. Note that d_i can be much less than n when the graph is sparse. If necessary, the two *copies* of each edge can be linked by a pointer to facilitate deletions.

Tradeoffs Between Adjacency Lists and Adjacency Matrices

Comparison	Winner
Faster to test if (x, y) exists?	matrices
Faster to find vertex degree?	lists
Less memory on small graphs?	lists $(m + n)$ vs. (n^2)
Less memory on big graphs?	matrices (small win)
Edge insertion or deletion?	matrices $O(1)$
Faster to traverse the graph?	lists $m + n$ vs. n^2
Better for most problems?	lists

- Both representations are very useful and have different properties, although *adjacency lists* are probably better for most problems.

Graph Vertex Representation

```
class GraphVertex:
    def __init__(self, label):
        self.label = label
        self.edges = []
    def __str__(self):
        output = str(self.label)+': '
        for e in self.edges:
            output += f"{self.label}->{e.label} "
        return output
```

- An undirected edge (x, y) appears twice in any adjacency-based graph structure, once as y in x's list, and once as x in y's list.

Initializing a Graph

```
def initial_graph(n:int, edge_list:list)->list:
    graph = []
    for i in range(n):
        graph.append(GraphVertex(i))
    for e in edge_list:
        if  0 <= e[0] < n and 0 <= e[1] < n:
            graph[e[0]].edges.append(graph[e[1]])
    return graph
```

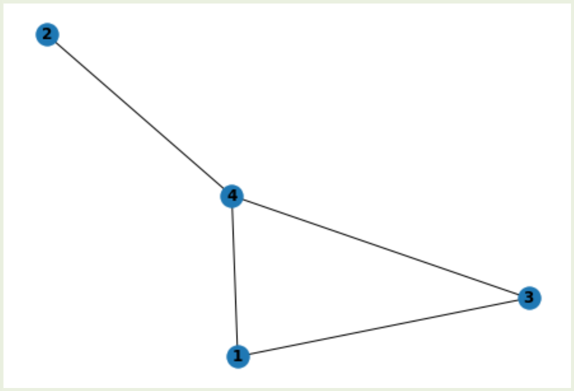
Create an Instance of Graph

```
edge_list=[[2,4],[4,2],[1,4],[4,1],[3,4],[4,3],[1,3],[3,1]]

graph = initial_graph(5,edge_list)
```

Other Options - NetworkX

- NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. (http://networkx.github.io/)

```
import networkx as nx


graph = nx.Graph()
graph.add_nodes_from(range(1,5))
edge_list=[[2,4],[4,2],[1,4],[4,1],[3,4],[4,3],[1,3],[3,1]]
graph.add_edges_from(edge_list)
nx.draw(graph, with_labels=True, font_weight='bold')
```



Other Options - igraph

- igraph is a library for creating and manipulating graphs. It is intended to be as powerful (ie. fast) as possible to enable the analysis of large graphs. (https://igraph.org/python/)

```
from igraph import *


graph = Graph()
graph.add_vertices(5)
edge_list=[[2,4],[1,4],[3,4],[1,3]]
graph.add_edges(edge_list)
print(graph)
```

```
IGRAPH U--- 5 4 --
+ edges:
2--4 1--4 3--4 1--3
```

Exercise

Partition Implementation

```
def partition(nums:list, low:int, high:int)->int:
    pivot = nums[low]
    leftwall = low
    for i in range(low+1,high+1):
        if nums[i] < pivot:
            leftwall += 1
            nums[i], nums[leftwall] = nums[leftwall], nums[i]
    nums[low], nums[leftwall] = nums[leftwall], nums[low]
    return leftwall
```

Problems of the Day

- Given an array A follows the steps of Quicksort partition implementation, What is the result of list nums after running partition(nums, 0, len(nums)-1)?
  - For Example: nums=[10,17,12,6,19,23,8,5]
  - Before: 10 17 12 6 19 23 8 5
  - After: 5 6 8 10 19 23 12 17
- Q1:
  - Before: 5 17 12 6 19 23 8 10
  - After: ?
- Q2:
  - Before: 17 5 12 6 19 23 8 10
  - After: ?
- Q3:
  - Before: 23 5 12 6 19 10 8 17
  - After: ?

Problems of the Day (Cont.)

- Is Facebook friendship a directed or undirected graph?
- Is X(Twitter) followership a directed or undirected graph?
- Is Facebook friendship a simple or non-simple graph?
- Is Facebook friendship a sparse or dense graph?

Program Exercises (Moodle CodeRunner)

- Exercise 05 (close at 4/1 23:59)
  - Dutch National Flag Problem (Red-White-Blue Sort) :
    - *Given an array of elements that are either red, white, or blue, sort them **in-place** so that all reds come before whites, and all whites come before blues. For example, ['blue', 'white', 'red'] -> ['red', 'white', 'blue']*
  - Convert a Graph from Adjacent List to Adjacent Matrix :
    - *To convert a graph from adjacency list which is a list of **GraphVertex** class to adjacency matrix in type of **nested list** of int.*