CSE 421: Introduction to Algorithms

Terminology: Complexity

Yin-Tat Lee

Administrativia Stuffs

HW1 is out.
Please submit to Gradescope

Guidelines:

- You can collaborate, but you must write solutions on your own
- You CANNOT search the solution online
- See Ed for more guidelines.

Tips:

- Rewrite your proof.
- Make sure you use assumptions of the problem
- Make sure it is easy to understand

More

Algorithm:

😃😃😃😃😃😃😃😃😃
😃😃😃😃😃😃😃😃😃
😃😃😃😃😃😃😃😃😃
😃😃😃😃😃😃😃😃😃

Runtime:

😃😃😃😃😃😃😃😃😃

Correctness:

😃😃😃😃😃😃😃😃😃
😃😃😃😃😃😃😃😃😃
😃😃😃😃😃😃😃😃😃

Exceptions (not limited to):

- If possible, reduce the question into a solved problem,
  For example, for stable matching, explain
  - What “man” and “woman” are corresponding to
  - What their preference are.
  - How convert the stable matching to what we asked in the question.

- If the algorithm is similar to one in the class,
  Simply explain the difference.
  Make sure your description is not ambiguous.

Definition for Efficiency in This Course

Worst case complexity:

The worst case running time $T(n)$ of an algorithm is

max # steps algorithm takes on any input of size n.

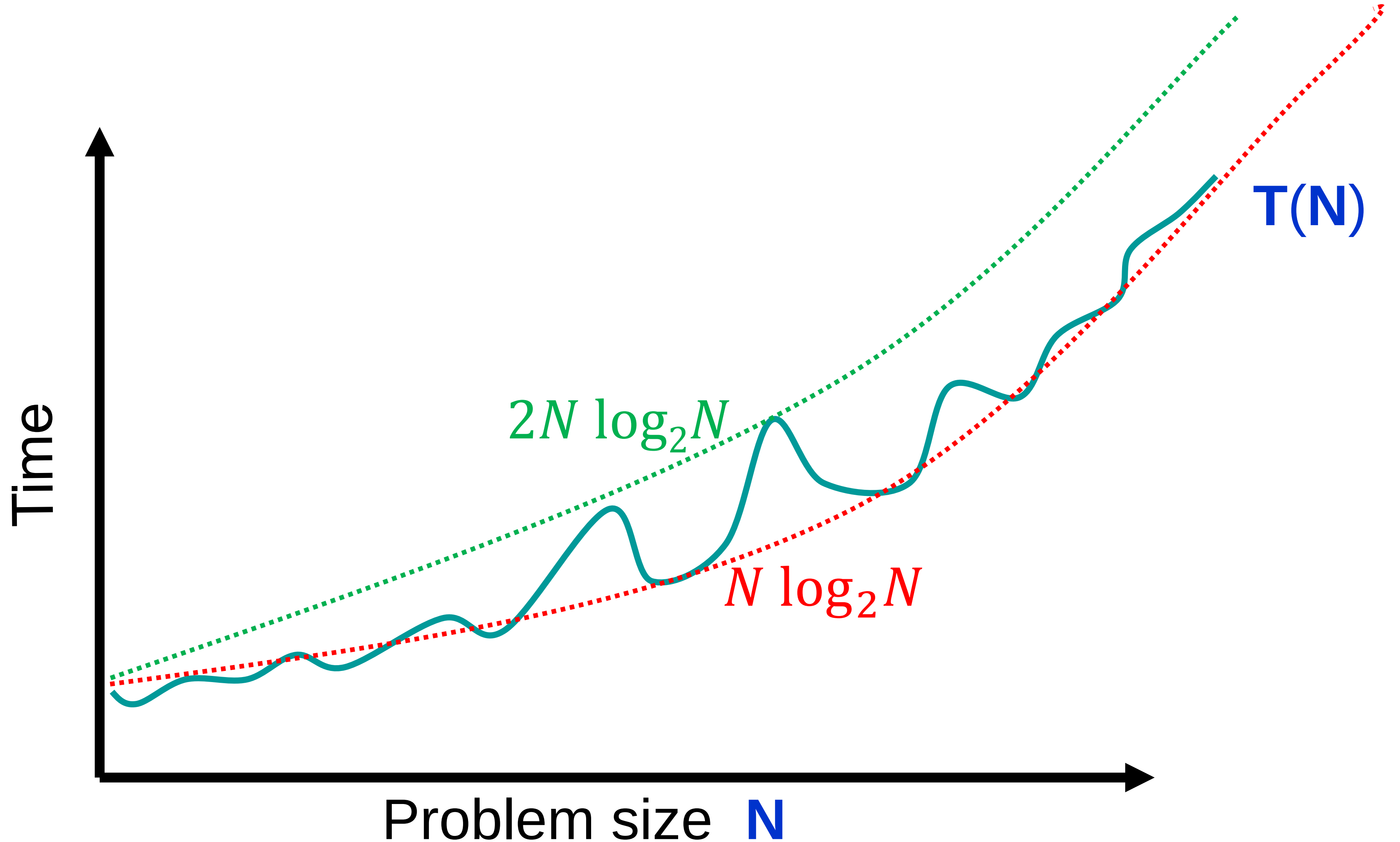
Definition of 1 step in this course:

- only simple operations (+,*,-,=,if,call,…).
- each operation takes one time step.
- each memory access takes one time step.
- no fancy stuff (add two matrices, copy long string,…).

Definition of efficiency in this course:

An algorithm is efficient if it has polynomial worst case runtime.

Time Complexity on Worst Case Inputs



O-Notation

Given two positive functions f and g

- $f(n) = O(g(n))$ if there is a constant $C > 0$ and N st

  $f(n) \leq Cg(n)$ for all $n > N$

- $f(n) = \Omega(g(n))$ if there is a constant $C > 0$ and N st

  $f(n) \geq Cg(n)$ for all $n > N$

- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, namely

There is a constant $C_1, C_2 > 0$ and N st

$$C_1 g(n) \leq f(n) \leq C_2 g(n) \text{ for all } n > N$$

Common Asymptotic Bounds

- Polynomials:

  $a_0 + a_1 n + \cdots + a_d n^d$ is $O\left(n^d\right)$

- Logarithms:

  $\log_a n = O(\log_b n)$ for all constants $a, b > 0$

- Logarithms: log grows slower than every polynomial

  For all $x > 0$, $\log n = O(n^x)$

Running Time

An algorithm runs in polynomial time if $T(n) = n^{O(1)}$.

Equivalently, $T(n) = O(n^d)$ for some constant d.

Name	Complexity class	Running time ($T(n)$)	Examples of running times	Example algorithms
constant time		$O(1)$	10	Finding the median value in a sorted array of numbers Calculating $(-1)^n$
inverse Ackermann time		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic time		$O(\log^* n)$		Distributed coloring of cycles
log-logarithmic		$O(\log \log n)$		Amortized time per operation using a bounded priority queue[2]
logarithmic time	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Binary search
polylogarithmic time		$\text{poly}(\log n)$	$(\log n)^2$	
fractional power		$O(n^c)$ where $0 < c < 1$	$n^{1/2}$, $n^{2/3}$	Searching in a kd-tree
linear time		$O(n)$	n, $2n + 5$	Finding the smallest or largest item in an unsorted array, Kadane's algorithm, linear search
"n log-star n" time		$O(n \log^* n)$		Seidel's polygon triangulation algorithm.
linearithmic time		$O(n \log n)$	$n \log n$, $\log n!$	Fastest possible comparison sort; Fast Fourier transform.
quasilinear time		$n\,\text{poly}(\log n)$		
quadratic time		$O(n^2)$	n^2	Bubble sort; Insertion sort; Direct convolution
cubic time		$O(n^3)$	n^3	Naive multiplication of two $n \times n$ matrices. Calculating partial correlation.
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$	$n^2 + n$, n^{10}	Karmarkar's algorithm for linear programming; AKS primality test[3][4]
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem.
sub-exponential time (first definition)	SUBEXP	$O(2^{n^\varepsilon})$ for all $\varepsilon > 0$		Contains BPP unless EXPTIME (see below) equals MA.[5]
sub-exponential time (second definition)		$2^{o(n)}$	$2^{n^{1/3}}$	Best-known algorithm for integer factorization; formerly-best algorithm for graph isomorphism
exponential time (with linear exponent)	E	$2^{O(n)}$	1.1^n, 10^n	Solving the traveling salesman problem using dynamic programming
exponential time	EXPTIME	$2^{\text{poly}(n)}$	2^n, 2^{n^2}	Solving matrix chain multiplication via brute-force search
factorial time		$O(n!)$	$n!$	Solving the traveling salesman problem via brute-force search
double exponential time	2-EXPTIME	$2^{2^{\text{poly}(n)}}$	2^{2^n}	Deciding the truth of a given statement in Presburger arithmetic

Data Structures CSE332 (constant time through polylogarithmic time)

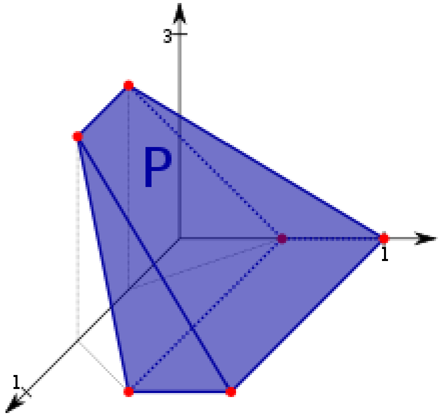
Sublinear Algorithms

This course (linear time through polynomial time)

Why "Polynomial"?

Point is not that n^{2000} is a practical bound, or that the differences among n and $2n$ and n^2 are negligible.

- "My problem is in P" is a starting point for a more detailed analysis
- "My problem is not in P" may suggest that you need to shift to a more tractable variant



Year/Authors	Steps
1989/Dyer-Frieze-Kannan [6]	n^{23}
1990/Lovász-Simonovits [18]	n^{16}
1990/Lovász [17]	n^{10}
1991/Applegate-Kannan [2]	n^{10}
1990/Dyer-Frieze [5]	n^8
1993/Lovász-Simonovits [19]	n^7
1997/Kannan-Lovász-Simonovits [11]	n^5
2003/Lovász-Vempala [20]	n^4
2020/YinTat,....	n^3
Github Repo	n^2

Other Complexities

Average Case Complexity:

avg # steps algorithm takes

Communication Complexity:

max # communication algorithm send between servers

Space Complexity:

max # space algorithm needs

Parallel Complexity:

max length of the longest series of operations that have to be performed sequentially due to data dependencies

CSE 351 Quiz

What is the cost of following operations? (in terms of cycle for CPU with 1 core)

- Compute a*b+c where a,b,c are float (throughput)

~1/16 cycles

- Cost of unpredictable if (latency)

~20 cycles

- Cost of reading 1 byte from a random location in memory (latency)

~300 cycles

- Cost of reading 1 byte from a random location in a M.2 SSD (latency)

~100k cycles

- Cost of reading 1 byte from a random location in a 7200RPM harddisk (latency)

~10M cycles

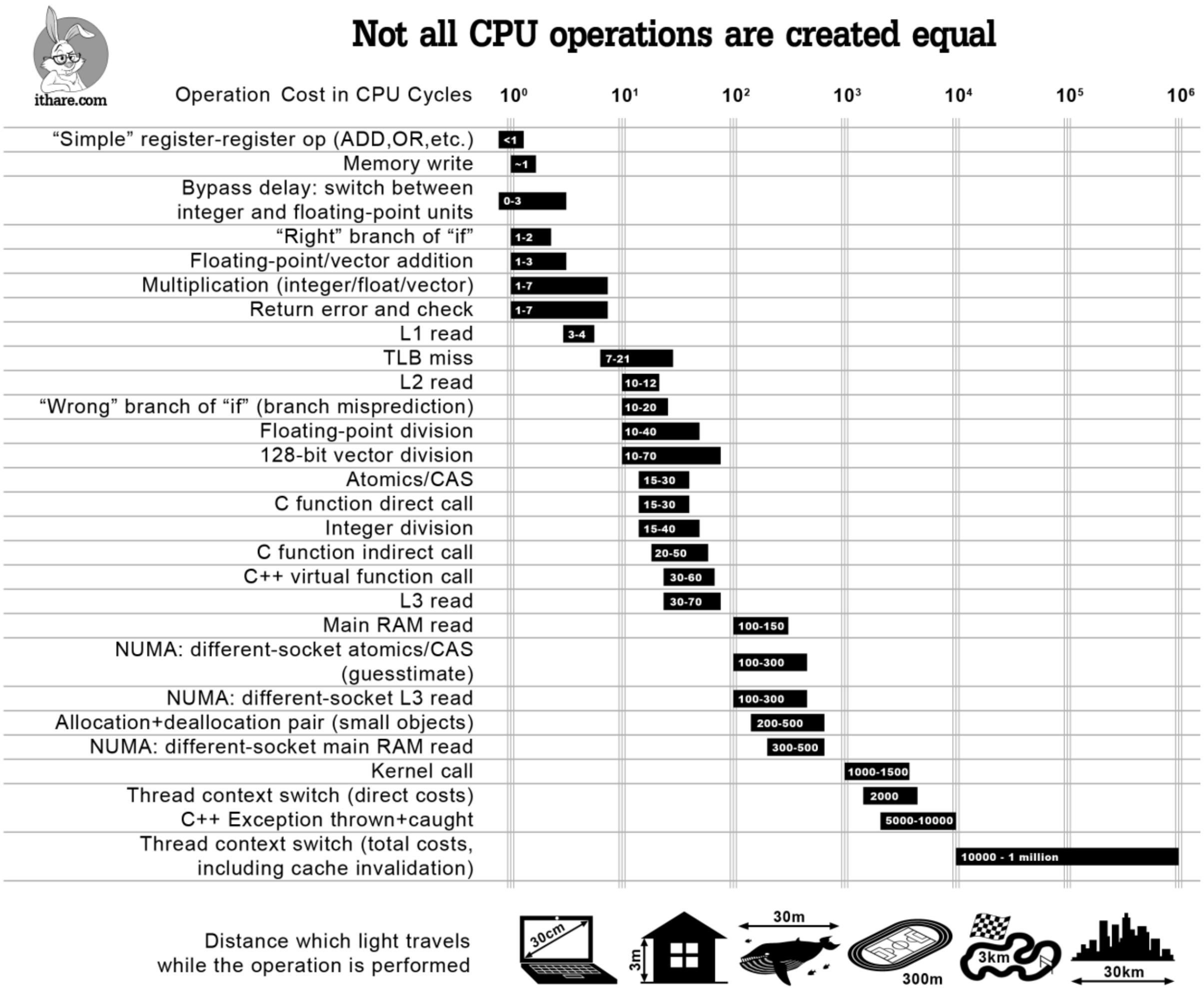
- Cost of Elon posting a Twitter from Mars (latency)

~10T cycles

Life is simple in 421.
Everything is O(1)

Warning

In real world, not all operations take same amount of time.



Warning

In real world, not all memory accesses take same amount of time.

Latency Numbers Every Programmer Should Know

latency.txt

```
Latency Comparison Numbers (~2012)
----------------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns                      20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us          ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD, 4X memory
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datacenter roundtrip
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms
```

Example:

Improving Google CPU usage by 0.5% via a better hash table

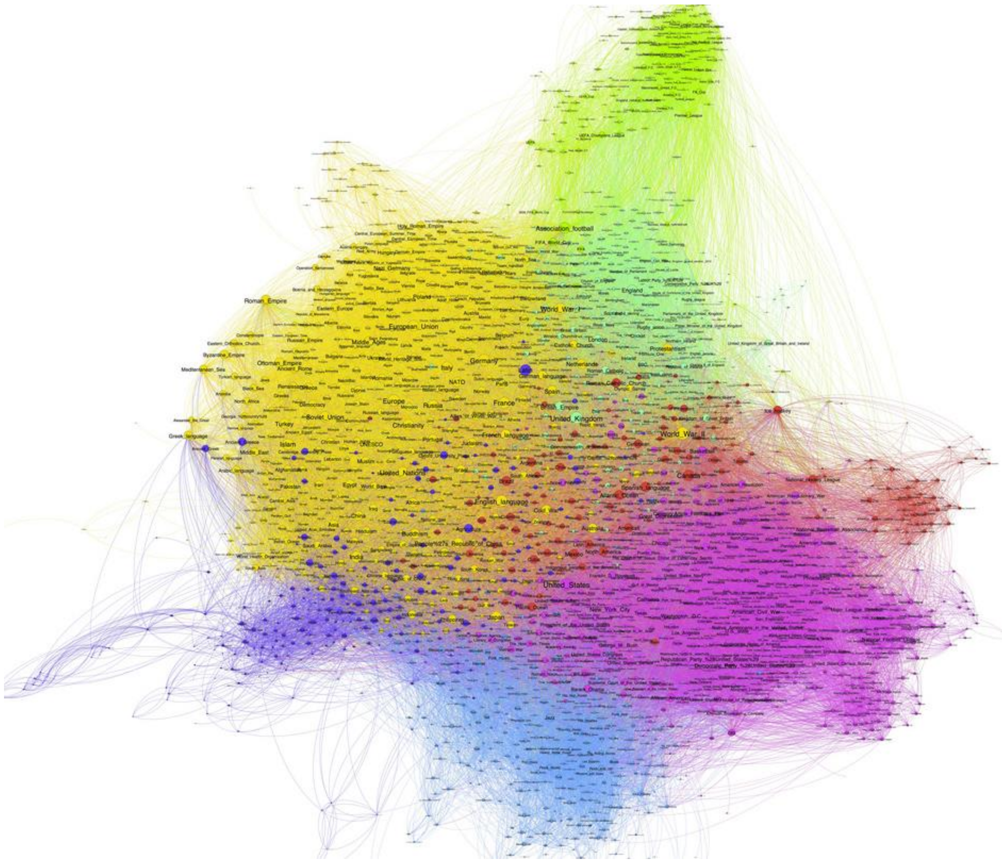
https://www.youtube.com/watch?v=ncHmEUmJZf4

CSE 421: Introduction to Algorithms

Terminology: Graph

Yin-Tat Lee

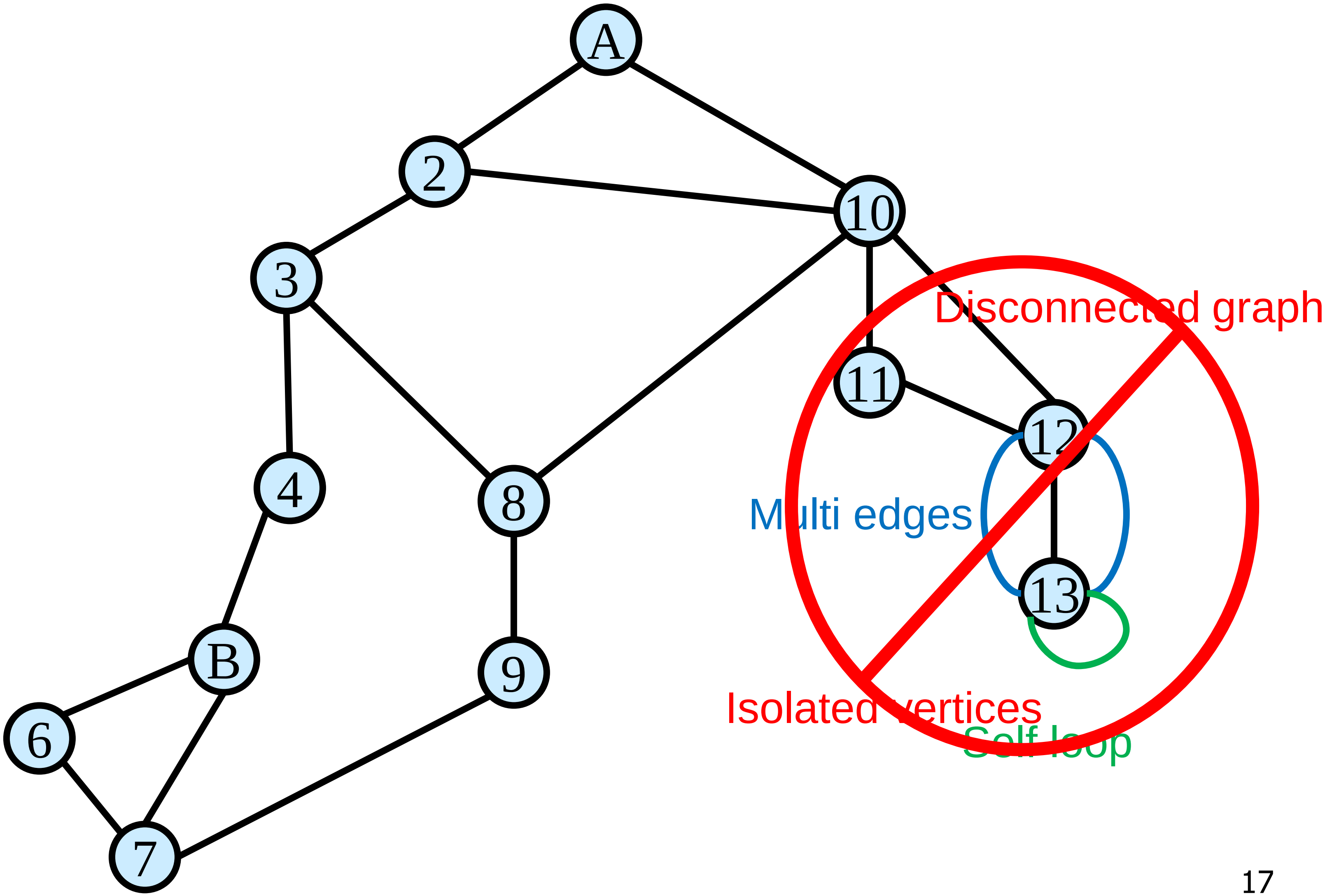
Graphs



Examples:

- Transportation networks
- Communication networks
- Internet
- Social networks
- Dependency networks

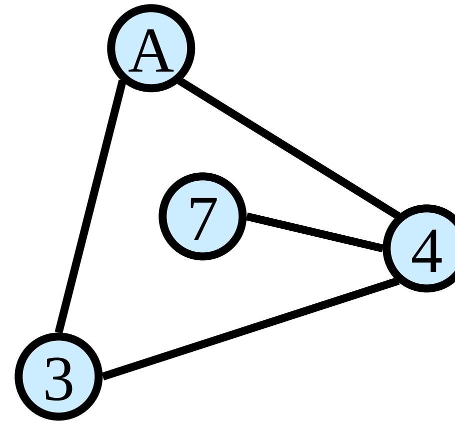
Undirected Graphs G=(V,E)

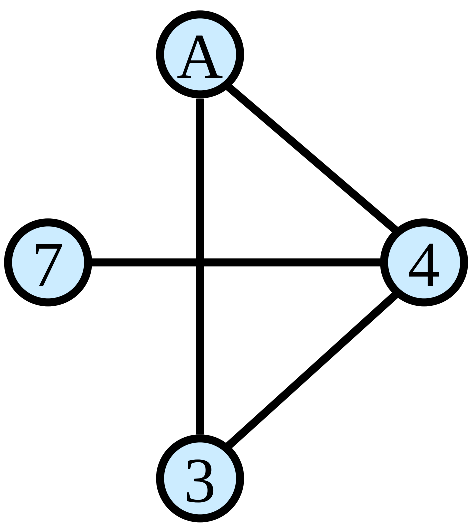


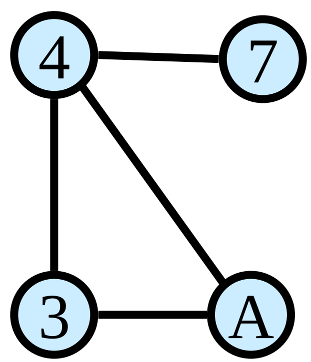
Graphs don’t Live in Flat Land

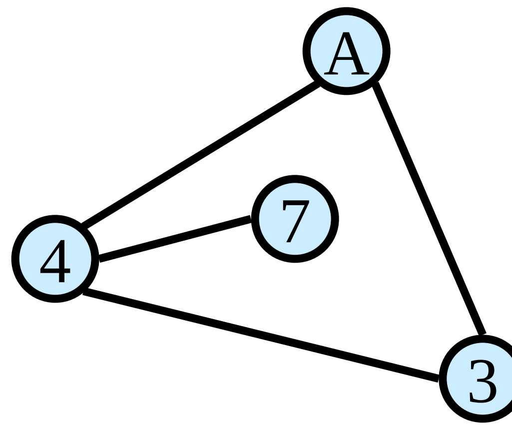
Geometrical drawing is mentally convenient, but mathematically irrelevant:

4 drawings of a single graph:

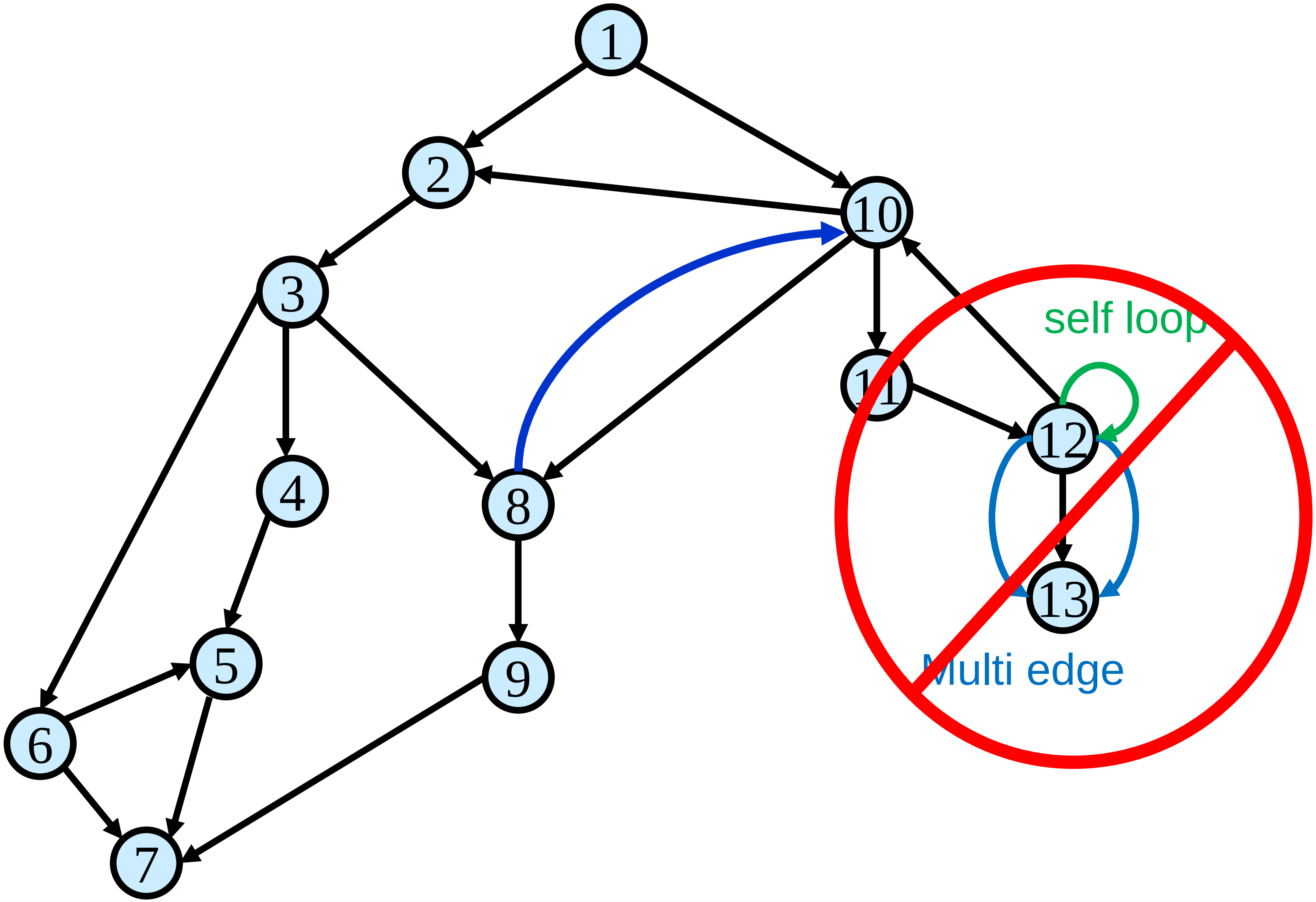








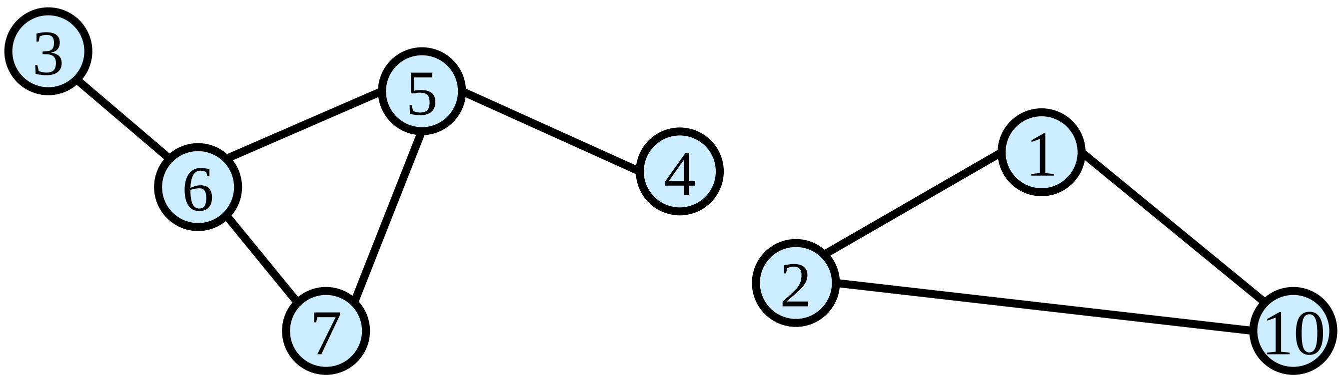
Directed Graphs



Terminology

- Degree of a vertex: # edges that touch that vertex

deg(6)=3

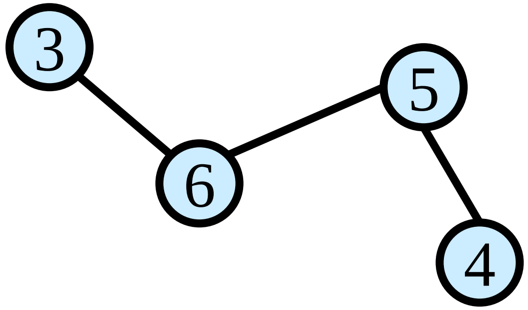


- Connected: Graph is connected if there is a path between every two vertices

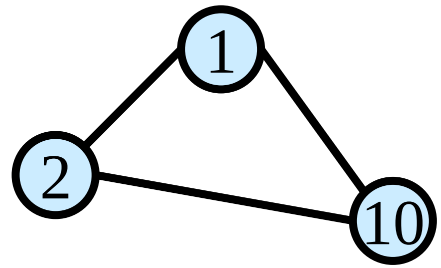
- Connected component: Maximal set of connected vertices

Terminology (cont'd)

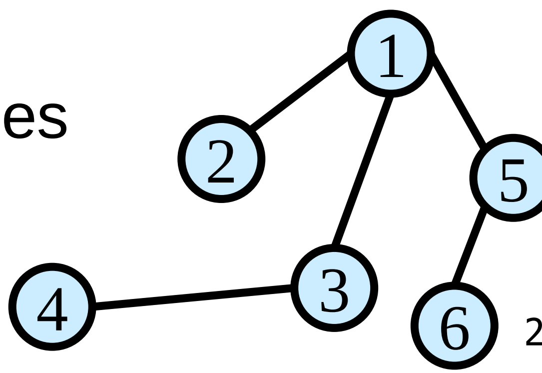
- Path: A sequence of distinct vertices s.t. each vertex is connected to the next vertex with an edge



- Cycle: Path of length > 2 that has the same start and end



- Tree: A connected graph with no cycles



Exercise: Degree 1 vertices

Claim: If G has no cycle, then it has a vertex of degree ≤ 1

(Every tree has a leaf)

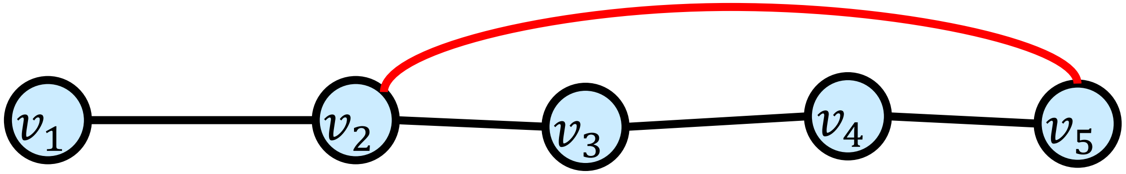
Proof: (By contradiction)

Suppose every vertex has degree ≥ 2.

Start from a vertex v_1 and follow a path, $v_1, \ldots, v_i$ when we are at v_i we choose the next vertex to be different from v_{i-1}. We can do so because $\deg(v_i) \geq 2$.

The first time that we see a repeated vertex ($v_j = v_i$) we get a cycle.

We always get a repeated vertex because G has finitely many vertices



Exercise: Trees and Induction

Claim: Every tree with n vertices has $n - 1$ edges.

Proof: (Induction on n.)

Base: $n = 1$, the tree has no edge

Induction: Let T be a tree with n vertices.

So, T has a vertex v of degree 1.

Remove v and the neighboring edge, and let T' be the new graph.

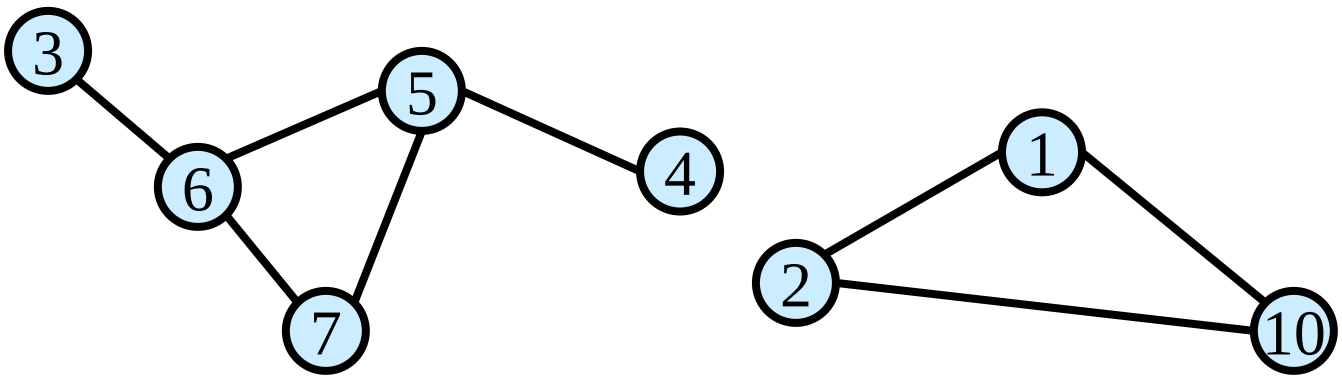
We claim T' is a tree: It has no cycle, and it must be connected.

So, T' has $n - 2$ edges and T has $n - 1$ edges.

Exercise: Degree Sum

Claim: In any undirected graph, the number of edges is equal to $(1/2)\sum_{\text{vertex } v} \deg(v)$

Pf: $\sum_{\text{vertex } v} \deg(v)$ counts every edge of the graph exactly twice; once from each end of the edge.



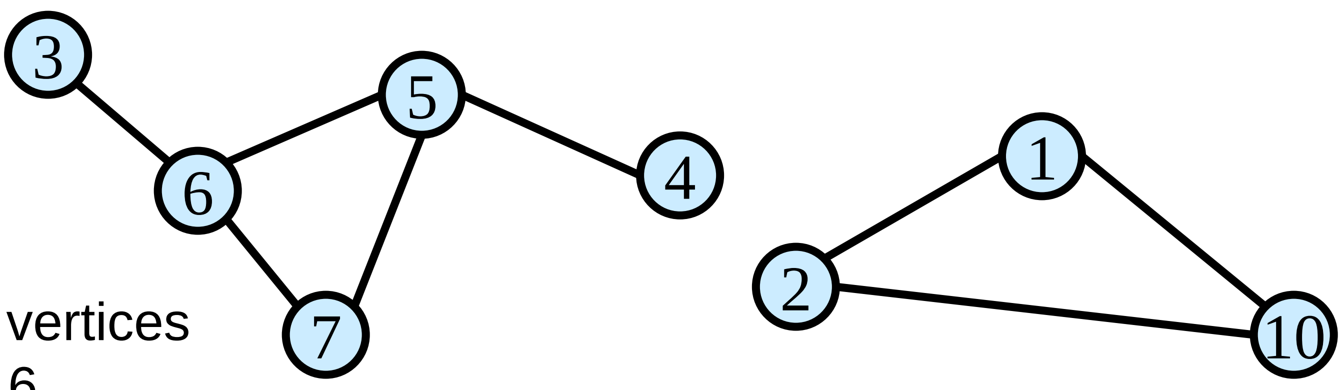
|E|=8

$$\sum_{\text{vertex } v} \deg(v) = 2 + 2 + 1 + 1 + 3 + 2 + 3 + 2 = 16$$

Exercise: Odd Degree Vertices

Claim: In any undirected graph, the number of odd degree vertices is even

Pf: In previous claim we showed sum of all vertex degrees is even. So there must be even number of odd degree vertices, because sum of odd number of odd numbers is odd.



4 odd degree vertices
3, 4, 5, 6

Exercise: #edges

Let $G = (V, E)$ be a graph with $n = |V|$ vertices and $m = |E|$ edges.

Claim: $0 \leq m \leq \binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$

Pf: Since every edge connects two distinct vertices (i.e., G has no loops)

and no two edges connect the same pair of vertices (i.e., G has no multi-edges)

It has at most $\binom{n}{2}$ edges.

Sparse Graphs

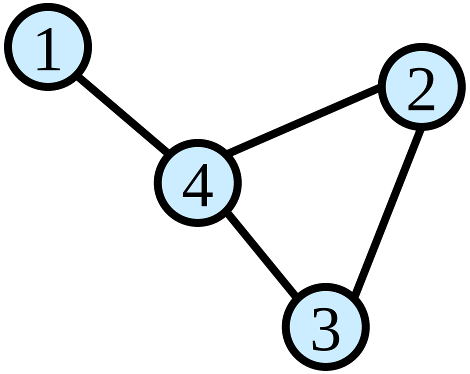
A graph is called sparse if $m \ll n^2$ and it is called dense otherwise.

Sparse graphs are very common in practice

- Friendships in social network
- Planar graphs
- Web graph

$O(n+m)$ is usually much better runtime than $O(n^2)$.

Storing Graphs



Vertex set $V = \{v_1, \ldots, v_n\}$.

Adjacency Matrix: A

- For all, $i, j, A[i, j] = 1$ iff $(v_i, v_j) \in E$
- Storage: n^2 bits

	1	2	3	4
1	0	0	0	1
2	0	0	1	1
3	0	1	0	1
4	1	1	1	0

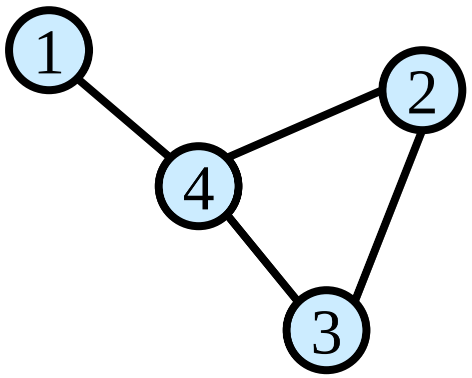
Advantage:

- $O(1)$ test for presence or absence of edges

Disadvantage:

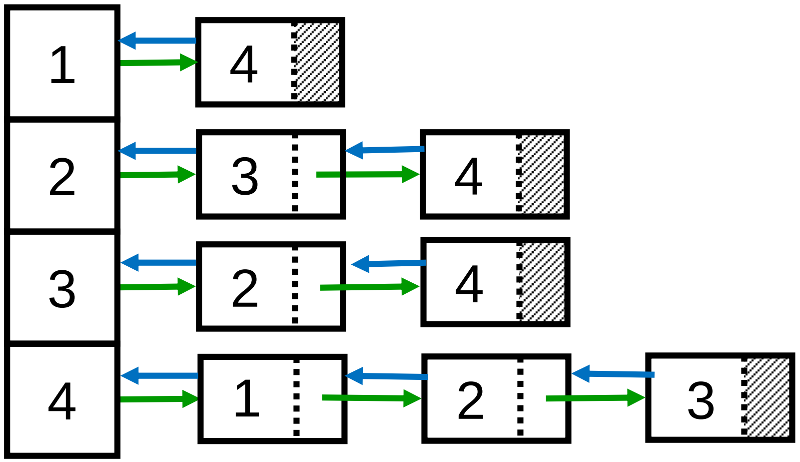
- Inefficient for sparse graphs both in storage and edge-access

Storing Graphs



Adjacency List:

O($n + m$) words



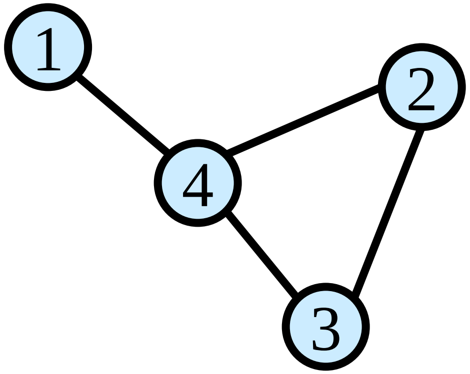
Advantage

- Compact for sparse
- Easily see all edges

Disadvantage

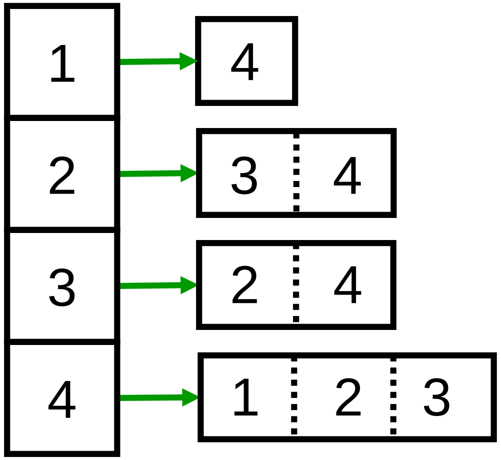
- Bad memory access
- Not good for parallel algorithms.

Storing Graphs



Adjacency Array:

$O(n + m)$ words



Advantage

- Compact for sparse
- Easily see all edges
- Better for memory access
- Better for parallel algorithms.

Disadvantage

- Difficult to update the graph

Storing Graphs

Implicit Representation:

$f(v)$ outputs an iterator of neighbor of v.

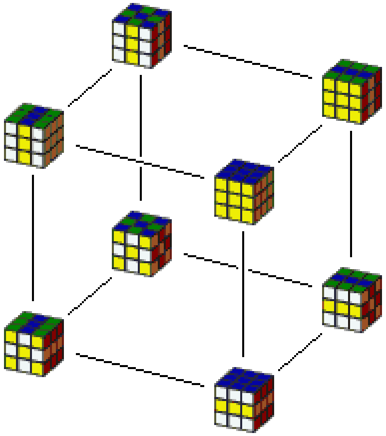
Aka, f(v)->next()->next()->next()->next()

Advantage

- No space is required

Disadvantage

- Mainly work for abstractly defined graph



2,125,922,464,947,725,402,112,000 states.

Storing Graphs

In practice, pick the representation according to the algorithm (depends how we want to access the graph).

In this course, we focus on asymptotic runtime.

We can simply do this:

- For each vertex, use a hash table to store its neighbors
- This gives O(1) time for many operations

    Insert, Delete, Find, Next, …