Sorting

- Adaptive : is the algorithm faster if some elements are already sorted
- In-place : space complexity, how much mem at least do you need to sort n-things → n things
  - If it use O(1) extra memory → in-place
- Stable : do items w/ same value end up in same relative order

Ex 1 4 6 10 4 → 1 4 4 6 10 → stable

→ 1 4 4 6 10 → not stable

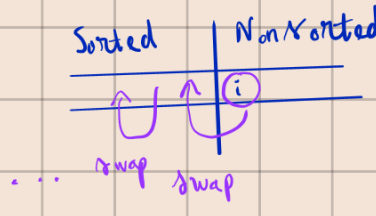
Bubble Sort :

per item : for each pair of items, if it out of order → swap

- keep track of last swap's index, the following iteration only goes up to this pos
- Run-time : $O(n^2)$
- stable, adaptive, and in-place

Insertion Sort :

- At iteration i :



- Stable, Adaptive, In-place, $O(n^2)$

:) Selection Sort : largest → last, 2nd-largest → 2nd-to-last, ...

- Stable : X
- Adaptive : X
- In place : ✓

Note : Take fewest actual swaps

:) Bubble Sort : partial adaptive

worse ↓

8 1 2 3 4 5 6 7 2 3 4 5 6 7 8 1

:) Cocktail Shaker Sort : Bubble Sort from left → right once
Bubble Sort from right → left once
⋮

- Still stops at no swap
- last indices on the left and right are the boundary for next iteration

:) Why use Insertion, Bubble, Cocktail Sort?

- stable + in-place + adaptive
- faster for small amounts of data
- used inside of more complex sorts

Note : • Worst (runtime ↓) is bubble sort

- Selection sort does fewer swaps
- Insertion sort does fewer compares

·) Can fix 1 item to be faster ?
- add all items a heap : $O(n)$
- repeatedly remove largest item : $O(\log n)$
  $\Rightarrow O(n \log n)$

- In-place (depends) ✗, adaptive ✗, stable (if you swap only adjacent elements : probably stable
  swap further elements
  → probably unstable

·) Divide & Conquer

·) Merge Sort

·) High-level plan : split data into small & large

How to choose pivot : many options

How to divide data ("partition")

4- step plan

① pivot
② Move pivot to index 0
③ Partition data into (< pivot) and (> pivot)
④ $\underbrace{A[1] \rightarrow}_{i} \; | \; \underbrace{\leftarrow A[n-1]}_{j}$

To find a num < pivot on the right
and a num > pivot on the left

→ Swap them

Stop when i > j

⑤ Return pivot to original pos
→ Do QuickSort on left A and right A left A | pivot | right A

Worst runtime : $O(n^2)$ → pivot = max / min

Avg runtime : $O(n \log n)$

- Stable Adaptive In place
  ✗ ✗ ✓ (don't use another Data structure)

- pivot strategies
  - Arr[0]
  - middle (Arr[0], Arr[size/2], Arr[size])
  - Random pivot
  - Guaranteed good pivot :
    - Median-of-Medians

i) Quick Select :

Stable Adaptive In-place
✗ ✗ ✓

·) Radix Sort :

Ex : Sorting n digit , sort by 1's place first
10's place second
⋮

• Sort by digit's pos :
  • 10 options : 0 - 9 → create 10 queues to put 1's place numbers
  • # loops = k
    • each loop put nums into buckets
    • get # outs from bucket 0 → 9

Runtime : O(kn)

Note : given a sorted arr ⇒ still O(kn)

Apaptive ✗ Stable ✓ In-place ✗

True Runtime : O(kn + kB)
- kn : k loop, put n digits into buckets
- kB : k loop, go through each bucket to get digits out

·) Pattern matching prob :

Input : input string s → length n
s's pattern s' → length m

Output : true if s contains s'

false otherwise

- Brute force : for each id in s
  check if s' is in there

:) Bayer Moore : go from right → left

:) KMP : Go from left → right

:) Rabin - Karp : Use hashing to check the whole pattern at each index

- If different hash code ⇒ pattern is not here
- hash (pattern) takes $O(m)$
  → "roll" - able hash, i.e. given one hash code already computed, compute the NEXT one in $O(1)$ time
  - new Hash = old Hash - old First + new Last
- hash $(a_1 a_2 \dots a_n) = a_1 B^{n-1} + a_2 B^{n-2} + \dots + a_n$ where B is a prime

  rolling : $(\text{hash} - \text{old first} \cdot B^{n-1})\, B + \text{new Last}$

- Running time :
  hash of pattern $O(m)$
  rolling hash : $O(n)$ times, $O(1)$ each

compare matching-hash patterns $O(m)$

$\rightarrow O(m+n)$

- Worst case : A lot of matching-hash pattern

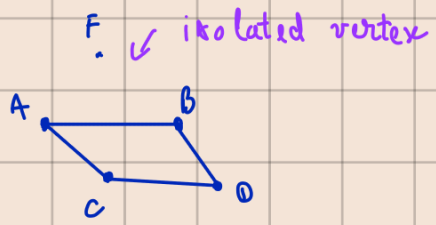
) Galil Rule :

Graph Theory

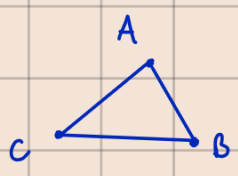
(in this class)

) Equal : G_1 and G_2 are equal if they have the same vertices and edges

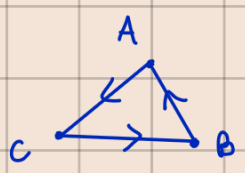
) Isolated vertex :



) Undirected graph :

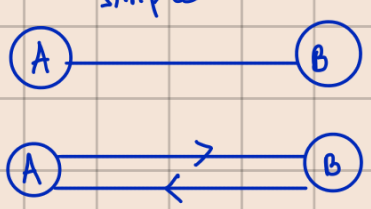


) Directed graph :



) Simple graph : no self loops & no repeated edges

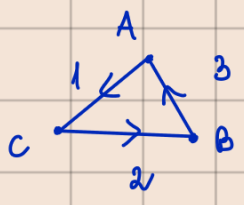
Simple



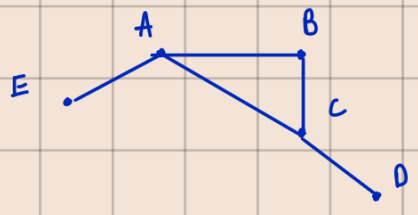
$\vec{AB} \neq \vec{BA}$

not Simple

- Weighted edges: edges can have "weight" (stands for cost, distance, ...)



- Graph traversal:



- Path: sequence of vertices, where 2 vertices next to each other are adjacent vertices
  - No repeated vertices
  - No repeated edges

- Trail: can repeat vertices but can't repeat edges

- Walk: can repeat vertices and edges

- Cycle: no repeated edges
  no repeated vertices except start = end

- Circuits: can repeat vertices
  can't repeat edges

- Connected: 2 vertices are connected if $\exists$ a path that connects them

:) Clique : a graph with maximum # edges
(must be simple graph)

$$\text{max \# edges} = \frac{|V|(|V|-1)}{2} \sim O(|V|^2)$$

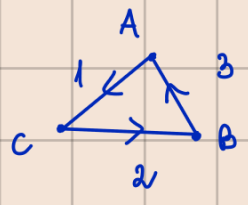
:) Sparse : few edges (trees)

$\sim O(|V|)$

:) Dense : many edges (cliques)

$\sim O(|V|^2)$

:) Graph - representation : $|V| \times |V|$ matrix, each entry corresponds to edge connect row vertex with col vertex



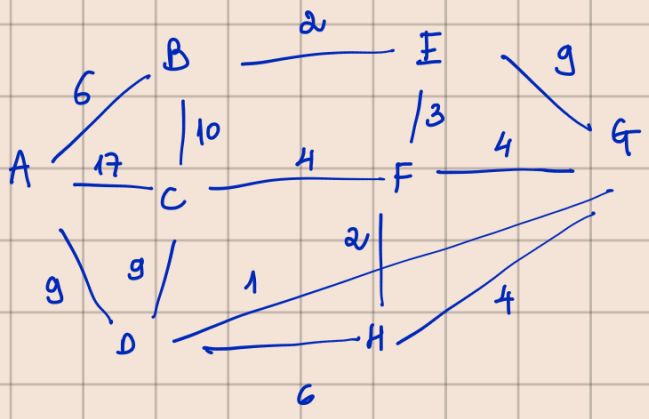
	A	B	C
A	3	—	1
B	—	—	—
C	—	2	—

- Map < vertex, [adjacent vertices] >

:) DFS

:) BFS

- Dijkstra fail with negative edges



Dist

A : 0
B : 6
E : 8
D : 9
G : 10
F : 11
H : 13
C : 15

6	17	9	8	11	17	10	15	18	14	14	15	13
~~A~~	~~B~~	C	~~D~~	~~E~~	~~F~~	G	~~G~~	H	C	F	H	C	~~H~~

76 2 2 1

3 5

- Minimum Spanning Tree (MST):

- "greedy" algo: trying to grab the min cost edge whenever possible (Also called Prim)

- goal: minimize total edge cost

- plan: start somewhere, track edge seen visited, always grab the cheapest one

How do we create the nodes and edges? Use node and adjacent list

- Plan: Use a heap (similar to Dijkstra)

· Note : if an edge that contains 2 visited nodes
→ pick that edge make a cycle

⇒ Kruskal : greedily choose the lightest edge in the entire graph that doesn't make a cycle

· Plan :

① sort the edges, choose the smallest edge in the graph which doesn't make a cycle

② How to know when a cycle appear ?
- Cannot use the strategy of Prim
- Label distinct connected components resulted from chosen edges with distinct names
- After choose edge that connect 2 C° component → update label

→ Actually, this can be done using Union - Find data structure

⇒ Union - Find data structure :

· find(x) : return the label of x
· union(x, y) : update labels after adding (x, y)

· runtime of find / union :

· Ackermann function = a function that grows very fast
· $A(n)$ ↑ fast
· $A^{-1}(n)$: inverse function of Ackermann func
→ ↑ slow

. Runtime of union / find : $O(A^{-1}(|V|))^{*}$