CS2040S Cheatsheet AY23/24 —— @Jin Hang

Recurrence Relation

$T(n)=T(n-1)+O\left(n^k\right)$	$O\left(n^{k+1}\right)$
$T(n)=T(n-1)+O(\log n)$	$O(n\log n)$
$T(n)=T(n-1)+O(n\log n)$	$O(n^2\log n)$
$T(n)=T(n/k)+O(1)$	$O(\log n)$
$T(n)=T\left(\frac{n}{2}\right)+O(n)$	$O(n)$
$T(n)=2T\left(\frac{n}{2}\right)+O(1)$	$O(n)$
$T(n)=aT\left(\frac{n}{a}\right)+O(n)$	$O(n\log n)$
$T(n)=aT(n-1)+O(1)$	$O\left(a^n\right)$

Master Theorem: For $T(n)=aT(n/b)+f(n)$
1. $\exists\epsilon>0$ s.t. $f(n)=O(n^{\log_b a-\epsilon})\Rightarrow T(n)=\Theta(n^{\log_b a})$
2. $\exists k\geq 0$ s.t. $f(n)=\Theta(n^{\log_b a}\lg^k n)\Rightarrow T(n)=\Theta(n^{\log_b a}\lg^{k+1} n)$
3. $\exists\epsilon>0$ s.t. $f(n)=\Omega(n^{\log_b a+\epsilon})$
   if $\exists c<1$ s.t. $af(n/b)\leq cf(n)\Rightarrow T(n)=\Theta(f(n))$

Order of Big-O Notation: $O(1)<O(\log(\log n))<O(\log n)<O(\log^m n)<O(n^k)<O(2^n)<O(n!)<O(n^n)$

$\log n!=\sum_{i=0}^{n-1}\log(n-i)\leq O(n\log n)$

$n\log m+m\log n=O(n\log m+m\log n)$

- $T(n)=O(f(n))$ if $\exists c>0, n_0>0$ s.t. $n>n_0$: $T(n)\leq cf(n)$
- $T(n)=\Omega(f(n))$ if $\exists c>0, n_0>0$ s.t. $\forall n>n_0$: $T(n)\geq cf(n)$
  For $T(n)=O(n^2)\rightarrow T(n)=\Omega(n)$ or $\Omega(n^2)$
- $T(n)=\Theta(f(n))$ iff. $T(n)=O(f(n))\&T(n)=\Omega(f(n))$

Peak Finding (Find local maximum)

1-D Peak Finding $T(n)=T(n/2)+O(1)=O(\log n)$

Output a local maximum in A, where `A[i-1] <= A[i]` and `A[i+1] <= A[i]`. Assume that `A[-1] = A[n] = -MAX_INT`

```
if A[n/2] is a peak then return n/2
else if A[n/2+1] > A[n/2] then Search for peak in right half.
else if A[n/2-1] > A[n/2] then Search for peak in left half.
```

Property: Recurse in the right half → ∃ a peak in the right half.

Correctness: 1. There exists a peak in the range [begin, end]
2. Every peak in [begin, end] is a peak in $[1,n]$

2-D Peak Finding: Reduce-and-Conquer

Output: a peak in A[n,m] that is not < (at most) 4 neighbors.

```
Find MAX element on border + cross.
if found a peak, DONE.
else: Recurse on quadrant containing element bigger than MAX.
```

Running time $T(n,m)=T(n/2,m/2)+O(n+m)=n\sum_{i=1}^{k}\frac{1}{2^i}+\sum_{i=1}^{k}\frac{1}{2^i}\leq 2n+2m=O(n+m)$

sort	best	average	worst	稳定	S(n)
bubble	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
selection	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$	×	$O(1)$
insertion	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
merge	$\Omega(n\log n)$	$O(n\log n)$	$O(n\log n)$	✓	$O(n)$
quick	$\Omega(n\log n)$	$O(n\log n)$	$O(n^2)$	×	$O(1)$
heap	$\Omega(n\log n)$	$O(n\log n)$	$O(n\log n)$	×	$O(n)$

sort	invariant (after k iterations)
bubble	largest k elements are sorted
selection	smallest k elements are sorted
insertion	first k slots are sorted
merge	given subarray is sorted
quick	partition is in the right position

Tree **BST Impt. Property:**
- all in left sub-tree < key < all in right sub-right
- same keys ≠ same shape, order of insrertion determine the shape of tree [ways of insertion $(n!)$ > shapes of BST(about 4^n)]
- On a **balanced** BST, all operations run in $O(\log n)$ time
- A BST is **balanced** if $h=O(\log n)$
- Node $n\leq\sum_{i=0}^{h}2^i=2^{h+1}-1$ for all BST

Height: Number of edges on longest path from root to leaf.
- $h(v)=0$ (if v is a leaf)
- $h(v)=\max(\text{h(v.left), h(v.right)})+1$
- $\log n+1-1=O(\log n)\leq h\leq n$, for tree with total n nodes.

Operation: **Search/Insert:** $O(h)$ **Traverse:** $O(n)$
- Successor Queries: $O(h)$

```
Search for key in the tree.
if (result > key), then return result.
if (result <= key), then search for successor of result.
node has a right child: successor(Node) = right.searchMin()
node has no right child: Find its next in order node
```

- Delete: $O(h)$
  1. **No children:** delete node directly
  2. **1 child:** Remove the node and link its parent and child node
  3. **2 children:** Find `successor(Node)` → swap it with its successor. → Delete node v

  Claim: successor of deleted node has at most 1 child

AVL Trees (* H.B. = height-balance for following)

Invariant: A node v is **H.B.** if |v.left.height − v.right.height| ≤ 1

Claim: A **H.B.** tree with n nodes has at most height $h<2\log(n)$.

Lemma 1: A **H.B.** tree with height h has at least $n>2^{\frac{h}{2}}$ nodes

proof. Let n_h be **min. num.** of nodes in a **H.B.** tree of height h.

$n_h\geq 1+n_{h-1}+n_{h-2}\geq 2n_{h-2}\geq\ldots\geq 2^k n_0$ where

$h-2k=0\Rightarrow k=\frac{h}{2}$. Hence, $h<2\log(n)$

Lemma 2: 高度为$h(h\geq 1)$的AVL树最少节点数递推公式
$S(h)=S(h-1)+S(h-2)+1\Rightarrow S(h)=\text{Fib.}(h+2)-1$

Delete: After deletion, for every ancestor of the deleted node
1. Check if it is height-balanced
2. If not, perform a rotation
3. Continue to the root
4. Up to $O(\log n)$ Rotations

Order Statistics Weight: size of the tree rooted at that node.
- w(leaf) = 1
- **rank** = $w_{左}+1$
- w(v) = $w_{左}+w_{右}+1$

Select(k): $O(h)\Leftrightarrow O(\log n)$

```
rank = m.left.weight + 1;
  if (k == rank) then return v;
  else if (k < rank) then return m.left.select(k);
  else if (k > rank) then return m.right.select(k−rank);
```

rank(node): $O(h)\Leftrightarrow O(\log n)$

```
rank = node.left.weight + 1;
while (node != null) do
  if (node is left child) then do nothing
  else if (node is right child) then
    rank += node.parent.left.weight + 1;
  node = node.parent;
return rank;
```

Insert/Delete
1. Insert/Delete
2. 节点→根遍历
3. 路径中所有节点v.weight+1($O(\log n)$)
4. 翻转调整AVL树
5. 翻转后更新节点weight ($O(1)$)

Maintain weight during rotations: $O(1)$ Time (翻转后只用改两个)

Interval Queries We need to maintain MAX after every rotation
1. **Search** for interval: $O(\log n)$
2. **Search** for all interval that overlap the node: $O(k\log n)$ for k overlapping intervals (Best Sol.: $O(k+\log n)$, Not Cover now)
3. **Insert / Delete**: After insert / delete, Conduct Rotation if the tree is out of balance ⇒ maintain MAX after every rotation.

Claims of Interval Search
1. If search goes right, then no overlap in left subtree.
2. If search in left subtree fails, then search also would fail in right subtree! ⇔ If search goes left and fails, then key < every interval in right sub-tree.
3. Either search finds key in subtree or it is not in the tree.

Orthogonal Range Searching $S(n)=O(n)$

Strategy: Preprocessing (buildtree): $O(n\log n)$
1. Use a binary search tree.
2. Store all points in the leaves of the tree. (Internal节点只存拷贝)
3. Each internal node v stores the MAX of any leaf in left subtree.

Operations
1. **FindSplit(low, high)**: $O(\log n)$, find split node.
2. **LeftTraversal(v, low, high)**(Also have RightTraversal): $O(\log n+k)$, Left Traverse. At every step, we either:
   - Output all right sub-tree and recurs left: $O(k)+O(\log n)$
   - Recurs right: $O(\log n)$

Invariant: The search interval for a left-traversal at node v includes the **maximum** item in the subtree rooted at v.

Dynamic: Need to fix rotations after insert and delete operations

(a,b)-Tree $2\leq a\leq(b+1)/2$ | **Rule 1: (a,b)-child policy**

Node type	#Keys Min	#Keys Max	#Children Min	#Children Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

Rule 3: Depth All leaf nodes must all be at the same depth.

Property An (a,b)-tree is balanced with $\log_b n\leq h\leq\log_a n$
1. **Search:** An (a, b)-tree with n nodes has $O(\log_a n)$ height. → Binary search for a key at every node takes $O(\log_2 b)$ time ⇒ $O(\log_a n\cdot\log_2 b)=O(\log n)$
2. **Split:** Find mediean v_m, split LHS($v<v_m$) and insert v_m to parent node. For split operation, copy $\frac{b}{2}$ elem. from one node to other and cost b to insert a key into a key list ⇒ $O(b)$.
3. **Insert:** $O(b\cdot\log_a n)=O(\log_a n)$
4. **Delete:** $O(\log_a n)$
5. **Merge and Share:** $O(b)$

kd-Tree Operations
1. **Search:** $O(\log n)$, If it is a horizontal / vertical split, then compare the x / y to the split value, and branch left or right.
2. **Build:** Use QuickSelect to find median of the data by the x or y as the split value, and then partition the points among the left and right children. $T(n)=2T(n/2)+O(n)=O(n\log n)$
3. **Find the minimum** x: $T(n)=2T(\frac{n}{4})+O(1)=O(\sqrt{n})$
   - horizontal split ⇒ recurse on the left child
   - vertical node ⇒ recurse on both children (minimum could be in either the top half or the bottom half)

Hashing **Hash Function** $h:U\rightarrow\{1\ldots m\}$
- $h(k)=k\mod p$, p 最好是质数,不接近2^i且与输入无关
- Fxed ⇒ probabilities based on the distribution of input
- **Worst Case** only happens when hash function is not good

Collisions: For $k_1\neq k_2$, if $h(k_1)=h(k_2)\Rightarrow$ collisions (Unavoidable as size$(U)>m$, pigeon hole principle)

A hash table takes more space than a simple list ⇒ has to store (Key, Value) [When $h(x_1)=h(x_2)$ useful to tell defference]

Simple Uniform Hashing Assumption
- Every key is equally likely to map to every bucket.
- Each key is put in a random bucket.
- Keys are mapped independently.
- Enough buckets ⇒ Not too many keys in one bucket.

Assume n items and m buckets
- Load(hash table) $\alpha=\frac{n}{m}$, $\alpha\uparrow$ P(collision) ↑
- P(X(i,j)=1)(i in j) $=\frac{1}{m}$
- E(X(i,j))$=\frac{1}{m}$
- E(collision) $=\sum_{i=0}^{n}\sum_{j>i}\frac{1}{m}=\sum_{i=0}^{n}\frac{n-i}{m}=\frac{n^2-n}{2m}$
- E(max chain length)$=O(\log n)$ or $\Theta(\frac{\log n}{\log\log n})$

Chaining
- **Insert:** $O(1+\text{cost}(h))$
- **Delete:** 双向链表$O(1)$ 单向$O(n)$
- **Search:** Worst: $O(n+\text{cost}(h))$ Expect: $O(1+\frac{n}{m})=O(1)$
- Using AVL Tree to store ⇒ Improve Worst Case and $S(n)\uparrow$

Open Addressing **Hash Function**
- For every bucket j, there is some i s.t.: $h(key,i)=j$
- The hash function is permutation of $\{1..m\}$

Double husing: $h(k,i)=(h_1(k)+ih_2(k))\mod m$

$h_2(k)$ relatively prime to m ⇒ hit all buckets, for example
1. let $m=2^n$ and make $h_2(k)$ prime to m
2. let m prime and make $0<h_2(k)<m$

⇒ Produce $\Theta(m^2)$ probing sequences

Linear Probing: $h(key,i)=h(k,1)+i\mod m\rightarrow\{1\ldots m\}$
- Produce m distinct probing sequence
- **Cluster:** $\frac{1}{4}$ full ⇒ $\Theta(\log n)$ size

Operation Analysis: Search/Probing/Delete
- P(i^{th} is full) $=\frac{n-i+1}{m-i+1}$
- $P(X\geq i)=\prod_{k=1}^{i}P(i)\leq(\frac{n}{m})^{i-1}=\alpha^{i-1}$
- E(X)$=1+\alpha(1+\alpha(1+\ldots))=\sum_{i=0}^{\infty}\alpha^i=\frac{1}{1-\alpha}$ [$\alpha\uparrow$, Perform. ↓]

Grow the table: For old table of m_1 with $n>m$ elem.
1. Choose a new size m_2 and hash func. h
2. For old elem. Compute $h(k)$ ⇒ Copy to new bucket

Total cost: $O(m_1+m_2+n)$
- +1 each time: $T(n)=\sum_{i=0}^{n}(m+i)=O(n^2)$
- ×2 each time: $T(n)=\sum_{i=0}^{\log i}O(m\times 2^i)=O(n)$
- m^2 each time: **Resize:** $O(n^2)$ **Insert:** $O(n)$ [inefficient]

Claim [If $(n==m)$, then $m=2m$. If $(n<m/4)$, then $m=m/2$]
1. double a table of size m ⇒ at least $\frac{m}{2}$ new items were added.
2. shrink a table of size m ⇒ at least $\frac{m}{4}$ items were deleted.

Amortized Analysis: Operation has amortized cost $T(n)$ if for every k, the cost of k operations is $\leq kT(n)$ [**NOT Average**]
- amortized cost always ≤ worst-case cost
- **hash table resizing**: $O(k)$ for k insertions ⇒ $O(1)$

Fingerprint Hash Table: Use fingerprint 0/1 ⇒ store m bits

Bloom filter: n elem. m buckets, using k hash functions
1. Only false positives, $n/m\uparrow,p\downarrow$
2. **All Operation:** $O(k)$
- P(a given bit is **0**) $=(1-\frac{1}{m})^{kn}\approx(\frac{1}{e})^{kn/m}$
- P(false positive) $=(1-(\frac{1}{e})^{kn/m})^k$
- $k=\frac{m}{n}\ln 2\Rightarrow$P(error)$=2^{-k}$
- delete operation: store counter instead of 1 bit
- intersection (bitwise AND), union (OR) ⇒ $O(m)$

Heap (Binary Heap): Heap Order + Binary Tree
1. Biggest items at root. Smallest items at leaves
2. Every level is full, except possibly the last
3. All nodes are as far left as possible.

Analysis Height starts from 0!
1. height(max) = $\lfloor\log n\rfloor$
2. Basic Operation: $O(h)$

3. insert/modfiyKey: Modify → bubbleup/down(down首选**left**)
4. delete(k): k和最底层最右侧元素互换→删k →Bubbledown
5. extractMax(): `Node v = root; delete(root);`
6. successor(k): $O(n)$ i.e. To find successor(leaf) ⇔ Search in leaf
7. Convert min-heap to max-heap: $O(n)$, Just heapify min-heap

Store in Array: 层序遍历后储存在数组里
- left$(x) = 2x+1$ • right$(x) = 2x+2$ • parent$(x) = \left\lfloor \frac{x-1}{2} \right\rfloor$

Heap Sort: All case $T(n) = O(n\log n) + O(n\log n) = O(n\log n)$
- Faster than MergeSort; A little slower than QuickSort
- Unstable i.e. $\{1,\mathbf{2},2\} \to \{\mathbf{2},1,2\} \to \{1,2,\mathbf{2}\}$
- Ternary (3-way) HeapSort is a little faster

1. **Unsorted Array → Heap**
   Keep insert elem.from Array to a empty binary heap
2. **Heap → Sorted Array**
   Keep `extractMax()` and add to the back from right to left

Heapify(v2): Recursivly **bubble down** all nodes [Max-Heap]

$T(n) = \sum_{h=0}^{\log n} \frac{n}{2^h} O(h) \le c \cdot n \left(\frac{\frac{1}{2}}{(1-\frac{1}{2})^2} \right) \le 2 \cdot O(n)$

- **Base:** 1. $> \frac{1}{2}$ Nodes are leaves 2. cost(bubbleDown) $= h$
- **Invariant:** After ith loop, A[i]..A[n-1] is a root of a max-heap

Priority Queue Implement: AVL Tree (indexed by priority)
- **insert:** $O(\log n)$
- **extractMax:** $O(\log n)$: Find maximum → delete

Graph **Edge:** $\forall e_1, e_2 \in E : e_1 \ne e_2$

Path: Intersects each node 最多一次. [Line] $(deg = 2,\ dia = n-1)$
Cycle: "Path" where first and last node are the same.
Tree: Connected graph with no cycles. n vertices ⇔ $n-1$ edges
Forest: Graph with no cycles.
Diameter: Maximum distance between two nodes
Degree of a graph: Maximum number of adjacent edges
Star: One central node, 其余节点与中心相连 $(deg = n-1,\ dia = 2)$
Clique: 完全图, All pairs connect by edges. $(deg = n-1,\ dia = 1)$
Cycle: $deg = 2$, $dia = \frac{n-1}{2}$ or $\frac{n}{2}$, with **even nodes → bipartite**
Bipartite Graph: Nodes divided into two sets with **no edges** between nodes in the same set. $(deg_{\max} = dia_{\max} = n-1)$
- **Tree, Star** and **2D grid graph** are bipartite

Strong Connect Component: $\forall v, w \in G, \exists \mathrm{P}(v,w), \mathrm{P}(w,v)$
Graph of strongly connected components is **acyclic**!
Connected Components: Undirected graph G=(V,E), u, v are connected if there is a path between them
- Use DFS to find number of (strong) connnected component in graph
- The number of CC. $\ge V - E$ ⇒ Connected G: $E \ge V-1$
- Number of CC. on a cycle=$(n-1)$!

Representing a Graph: Nodes and Edges for G(V,E)
Adjacency List: Nodes in array, Edges: linked list per node
- **Memory:** $O(V+E)$ Better in Memory!
- 列举邻居 $O(V)$ • **查找u,v关系:** $O(\min(|V|,|E|))$

Adjacency Matrix: A[v][w] = 1 iff. $(v,w) \in E$
- Matrix A^n represents len n path, $A[c][d] = A[c][x] \times A[x][d]$
- **Memory:** $O(V^2)$ • **列举邻居:** $O(E)$ • **i和v的关系:** $O(1)$

Base rule: if graph is **dense** $(|E| = \Theta(V^2))$ i.e. clique, use **adjacency matrix**; else use an adjacency list for **sparse** graph.

BFS (Queue)[邻接表] $T(n) = O(V) + O(E) = O(V+E)$
1. Build levels ⇒ Calculate level[i] from level[$i-1$]
   Explore all **outgoing edges** of v. $(O(E)$, each edge **twice**)
2. Add all unvisited neighbors of v to the queue
3. Skip already visited nodes.(Visit **only once**) $(O(V))$
4. Produce a Tree with root s

Shortest path is a tree, 可能high deg(Star), 可能high dia(Path)
BFS finds minimum number of **Steps** not minimum **Distance**.
- For Queue $\{v_1, \ldots, v_i\}$(v_1 is head) ⇒ $v_i.d \le v_{i+1}.d$

DFS (Stack)[邻接矩阵] $T(n) = O(V) \times O(V) = O(V^2)$
1. Visit v, Explore all outgoing edges of v.$(O(E))$, follow a path
2. Stack ⇒ Backtrack until find a new edge, recursively explore
3. Push all unvisited neighbors of v on the front of the stack
4. Don't re-visit a vertex $(O(V)$ to visit all nodes)

Parent Graph: Tree, **Don't** contains shortest paths
- BFS and DFS visit every node and edges once, but not every path
- Too expensive: some graphs have an **exponential** number of paths

Shortest Paths Not a unique answer!
Triangle inequality: $\delta(S,C) \le \delta(S,A) + \delta(A,C)$, δ is 最短距离
Lemma 1: $\delta(s,v) \le \delta(s,u) + 1$ for $\forall(u,v) \in E$
- Only chosen source vertex and not any destination vertex ⇒ Computes shortest paths from source to all other vertices
- **multiply** constant $C > 0$ on every weight SSSP remain the **same**
- **add** constant $C > 0$ **may change** SSSP
- Subpaths of shortest paths are shortest paths

Bellman-Ford: Relax all edges when visit a node, $T(n) = O(EV)$

```
relax(int u, int v) for e : graph // relax every edge
if(est[v] > est[u]+weight(u,v)) then est[v] = est[u]+weight(u,v)
```

Terminate: When an entire sequence of $|E|$ `relax` have no effect
Invariant: After k iterations, all nodes u whose shortest paths are within k hops have correct updated distance [est[u] = distance(s,u)]
Invariant 2: Suppose D is destination node. After k iterations, the kth node along the shortest path to D has correct updated distance.
- **Negative weight cycles (N.W.C):** Run $|V|+1$ iterations. If an estimate **changes** in the last iteration, then ∃ N.W.C
- Negative weight edge → correct answer! N.W.C → false
- If **all** weights are the **same**, use BFS

Dijkstra [Priority Queue] $O(E\log V)$ $(w(u,v) \ge 0$ for $(u,v) \in E)$
1. Maintain distance estimate for every node.
2. Begin with empty shortest-path-tree, Repeat:
   - Consider node with minimum estimate.
   - Add node to shortest-path-tree.$(O(\log V))$
   - Relax all outgoing edges$(O(E))$.

Properties: $T(n) = O(V)$ `extractMin` $+ O(E)$ `decreaseKey`
1. Relax each edges **Only Once**! 2. Terminate: $u.d = \delta(s,u)$

Topological Order (DFS) $T(n) = O(V+E)$
1. Sequential total ordering(Not Unique!) of all nodes
   - Pre-Order: Process each node when first visited
   - Post-Order: Process each node when last visited
2. Edges only point **forward**(Cycle ⇒ No Topo. Order)
3. Topological order of a tree ⇔ **Pre-Order** traversal of the tree.

Topological Sort Kahn's Algorithm $(T(n) = O(V+E))$
1. S = all nodes in G that have no **incoming edges**, (choose one)
2. Add nodes in S to the topo-order
3. Remove all edges adjacent to nodes in S
4. Remove nodes in S from the graph
- Maintain a Priority Queue: $O(E\log V)$
- **Invariant:** At each step, at least one node must have in-degree of 0
- After Kahn's, can use **topo.-order** relaxation to find SSSP

SSSP in DAG $(O(V+E))$ 1.Topo. Sort ⇒ 2. Relax in Order
Longest P. Negate weight to find LP in DAG
×Solved efficiently. Graph with positive weight cycles → impossible

Union Find Union(u,v)-Find `isConnected`(u,v) **Quick Find** ↓
- **find(int p, int q)**: return(compId[p] == compId[q]); $[O(1)]$
- **union(int p, int q)**: $[O(n)]$

```
updateComponent = componentId[q]
for (int i=0; i<componentId.length; i++)
    if (componentId[i] == updateComponent)
        componentId[i] = componentId[p];
```

Quick Union: Implement a flat tree using a parent array
General idea: Find/Union the root the a tree

```
while (parent[p] != p) p = parent[p];
while (parent[q] != q) q = parent[q]; then find/union:
```

1. **find(int p, int q)**: Compare root, `return (p == q);` $[O(n)]$
2. **union(int p, int q)**: Join the root, `parent[p] = q;` $[O(n)]$

Weight Union: Root of big tree be the root of smaller tree
- tree T_1 **one level deeper** after union iff. size$(T_2) >$ size(T_1)
- size$[T_i] >$ size$[T_j]$ ⇒ size$[T_i]$ + size$[T_j] >$ 2size$[T_j]$ ⇒ size$[T_n] > 2^h = n$
- Both **find** and **union** become $O(\log n)$
- weight/rank/size/height of subtree doesn't change except root
- weight/rank/size/height only increases when tree size doubles

Path Compression: After finding the root, set the parent of each traversed node to the **root**.
- **Alternative:** Make every other node in path point to grandparent
- Find: $O(\log n)$; Union: $O(\log n)$

Complexity: Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m,n))$ time. $(\alpha(n) \le 5)$

MST Input: Connect **undirected** Graph with **weight**
Spanning Tree: Acyclic subset of edges that connects all nodes
Cut: A partition of the vertices V into two disjoint subsets. ⇒ An edge crosses a cut if it has one vertex in each of the two sets.

Property
1. MST ≠ Shortest Path, MST guarantee the heaviest edge on a path is **minimised** ⇒ **total weight** is minimal
2. Cut an MST ⇒ two pieces are both MSTs.
3. For every **cycle**, the **maximum** weight edge is not in the MST; the **minimum** weight edge **may or may not** be in the MST.
4. For every partition of the nodes, the **minimum** weight edge across the **cut** is in the MST.
5. For every vertex, the **minimum** outgoing edge is always part of the MST; the **maximum** outgoing edge may be part of MST
6. **Variant:** An (M)ST contains exactly $(V-1)$ edges

Generic MST: Apply Red/Blue rule until cannot ⇒ MST = $E_{蓝}$
1. **Cycle** with no red arcs ⇒ color **max-weight** edge red.
2. **Cut** with no blue arcs ⇒ color **min-weight** edge blue.

Termination: Every edge is colored, No blue cycles
Prim: $T(n) = O(V\log V) + O(E\log V) = O(E\log V)$
Basic idea: 1.Priority Queue 2.随机选起始点
$S = \{A\}$ (set of nodes connected by blue edges), **Repeat**
- Identify cut: {S, V–S}
- Find minimum weight edge on cut (`extractMin`$[O(\log V)]$)
- Add new node to S

Kruskal: $O(E\log V)$, Keep adding lightest edge
1. Initialize V using Union-Find ($|V|$ trees for each node)
2. Sort edges by weight.$(O(E\log E) \le O(E\log V^2) = O(E\log V))$
3. (**Variant**) Consider edges in **ascending** order: **If** both endpoints **find** in the same blue tree, color the edge **red**. **Otherwise**, color the edge **blue** and **union**. ($|E|$ Times)

Implement: Union-Find to determine whether two nodes in the same tree. For E edges, Find and Union $[O(\alpha(n))$ or $O(\log V)]$
Boruvka's Algo.: $T(n) = O(V+E) \times O(\log V) = O(E\log V)$
Initially: Create n connected components, one for each node
One "Boruvka" Step: $O(V+E)$, $k \to \frac{k}{2}$ components
1. For each connected component, search for the minimum weight outgoing edge. (BFS/DFS, $O(V+E)$)
2. Add selected edges.
3. Merge connected components. (Scan every node, update)

Special Case: All edge have same weight ⇒ BFS/DFS
Directed MST: Every Node Reachable on the path from the root
For every node except root, add **min** weight incoming edge $[O(E)]$
Re-weight: Add/Times k will not change relative order!
Max. Spanning Tree: Negate and Run MST or Reverse Krus.
TSP-MST Algorithm $T(n) = O(E\log V) + O(V+E)$
1. Find MST using MST algo. $[O(E\log V)]$
2. DFS on MST, every time visit a node ⇒ Every node appears at least twice in DFS walk $d_0 d_1 d_2 \ldots d_{2n-1}$ $[O(V+E)]$
3. Take short-cuts by skip visited city to avoid revisiting cities (triangle inequality ⇒ only len(tour)↓) $[O(V)]$

DP Overlapping sub-problems + Optimal Substructure
Strategy: using recursion and memoization may be asymptotically **faster** than a bottom-up implementation.
A.P.S.P and Floyd Algo.: $O(V^3)$
- Negative weight edge OK, cycle Not OK
- $d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\} & \text{if } k \ge 1 \end{cases}$

Longest Common Subsequence: $LCS(A(n), B(n)) =$
$\begin{cases} LCS(A(n-1), B(n-1)) + 1, A[n] == B[n] \\ \max(LCS(A(n), B(n-1)), LCS(A(n-1), B(n))), A[n] != B[n] \end{cases}$

Abstract Data Structure
symbol table operation: insert/delete/search/contains/size
- Impossible to **search** in fewer than $\Theta(\log n)$ comparisons

Dictionary: Symbol Table + find successor/predecessor
1. Sorting with a dictionary: $O(\log n)$
   - Insert every item into the dictionary.
   - Search for the minimum item.
   - Repeat: find successor
2. **Cannot** be implemented with a hash table(successor(key))

data structure	search	insert
sorted array	$O(\log n)$	$O(n)$
unsorted array	$O(n)$	$O(1)$
linked list	$O(n)$	$O(1)$
tree (kd/(a, b)/bst)	$O(\log n)$, $O(h)$	$O(\log n)$, $O(h)$
trie	$O(L)$	$O(L)$
heap	$O(n)$	$O(\log n)$, $O(h)$
dictionary	$O(\log n)$	$O(\log n)$
symbol table	$O(1)$	$O(1)$
chaining	$O(n)$	$O(1)$
open addressing	$\frac{1}{1-\alpha} = O(1)$	$O(1)$
priority queue	(depends) $O(\log n)$	$O(\log n)$
queue	$O(n)$	$O(1)$

Special Case in SSSP

Condition	Algorithm	$T(n)$
No Negative Weight Cycle	Bellman-Ford	$O(VE)$
Unweighted/Equal Weight	BFS	$O(V+E)$
No Negative Weight	Dijkstra	$O(E\log V)$
Any Tree	BFS/DFS	$O(V+E)$
DAG	Topological Sort	$O(V+E)$

- In a tree, ∃only one path between two vertices ⇒ BFS/DFS find

Supplyment

Mathematical

- $\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$
- $\frac{\log_b n}{\log_a n} = \log_b a$
- $E[X] = \frac{1}{p}$

Queick Sort

After every partition, the pivot should be in the correct position

1. 找选项中第一个元素在他应该在的位置p
2. 检查是否符合$e_{上} > p \quad \& \quad e_{下} < p$

Partition with pIndex

1. swap(pIndex, 0)
2. start after pivot in A[0]
3. Define: A[n+1] = $+\infty$
4. Partition like before

Duplicates Elem.

one-way Q.S. $O(n^2)$, Every partition arr divided to $[1 : n-1]$

Partition

```
partition(A[1..n], n, pIndex)
pivot = A[pIndex];
swap(A[1], A[pIndex]);
low = 2;
high = n+1;
while (low < high)
    while (A[low] < pivot) and (low < high) do low++;
    while (A[high] > pivot) and (low < high) do high-- ;
    if (low < high) then swap(A[low], A[high])
swap(A[1], A[low-1]);
return low-1;
```

Tree

Perfectly balanced: Both children of each node have an equal number of nodes and are perfectly balanced.

<u>**Order Statistics**</u> **Weight:** size of the tree rooted at that node.

- w(leaf) = 1
- **rank** $= w_{左} + 1$
- w(v) $= w_{左} + w_{右} + 1$

Select(k): $O(h) \Leftrightarrow O(\log n)$

```
rank = m.left.weight + 1;
    if (k == rank) then return v;
    else if (k < rank) then return m.left.select(k);
    else if (k > rank) then return m.right.select(k-rank);
```

rank(node): $O(h) \Leftrightarrow O(\log n)$

```
rank = node.left.weight + 1;
while (node != null) do
    if (node is left child) then do nothing
    else if (node is right child) then
        rank += node.parent.left.weight + 1;
    node = node.parent;
return rank;
```

Insert/Delete
1. Insert/Delete
2. 节点→根遍历
3. 路径中所有节点 v.weight+1($O(\log n)$)
4. 翻转调整AVL树
5. 翻转后更新节点 weight ($O(1)$)

Maintain weight during rotations: $O(1)$ Time (翻转后只用改两个)

String

- Compare 2 Strings: $O(L_{\max})$
- Append 2 Strings: $O(L_1 + L_2)$

```
PriorityQueue pq = new PriorityQueue();
for (Node v : G.V()) { pq.insert(v, INFTY); }
pq.decreaseKey(start, 0);
HashSet<Node> S = new HashSet<Node>();
S.put(start);
HashMap<Node,Node> parent = new HashMap<Node,Node>();
parent.put(start, null);
while (!pq.isEmpty()){
    Node v = pq.deleteMin();
    S.put(v);
    for each (Edge e : v.edgeList()){
        Node w = e.otherNode(v);
        if (!S.get(w)) {
            pq.decreaseKey(w, e.getWeight());
            parent.put(w, v);
        }
    }
}
```

-*-*-*-*-*-*- PLEASE DELETE THIS PAGE! -*-*-*-*-*-*-

Information
Course: CS2040/S
Type: Final Cheat Sheet
Date: July 3, 2024
Author: QIU JINHANG
Link: https://github.com/jhqiu21/Notes

-*-*-*-*-*-*- PLEASE DELETE THIS PAGE! -*-*-*-*-*-*-