COMP2123 Notes

Algorithms in O(1)

- Adding two numbers (as such finding average of n numbers is O(n))
- Return a value.
- Find a[i] (array)
- Following an object reference
- Calling method
- Assigning values

Abstract Data Types (ADT)

- Like an interface, it specifies the data type and the functions/methods, but users don't know what is behind the scenes.
- Code the tricky part once and people can just use the application.
- Data structure : concrete representation of data, point of view of an implementer (not user)
    - The algorithm
- Abstract base class (abc) in python
- Data structure implementation: inherits from the abc, provide code all required methods
    - If the operation of the method does not depend on the size of the list, it is most likely in constant time O(1)

Index Based Lists

- size(), isEmpty(), get(i), set(i,e) add(i,e), remove(i,e)

Array Based Lists

- Stored contiguously in memory, if you know the address of the start and the size, you can find address of each element
- Time complex for add(i,e) and remove(i,e) is O(n) as all elements after i are to shift back / forward
- Allow random access

size()	o(1)
isEmpty()	o(1)
get(i)	o(1)
set(i,e)	o(1)
add(i,e)	o(n)
remove(i)	o(n)

Positional lists

- Store elements as positions
- Unlike index, this keeps referring to the same entry even after insertion / deletion happens elsewhere.
- element() : return the element stored at the position instance.
- size(), isEmpty(), first(), last()
- before(p), after(p), insertBefore(p), insertAfter(p), remove(p)

Singly Linked List

- Sequence of node, each with reference to the next
- Randomly stored in mem
- Time complexity O(N):
    - insertBefore(p,e): insert e in front of p

size()	o(1)
isEmpty()	o(1)
first()	o(1)
last()	o(1)
before(p)	o(n)
after(p)	o(1)
insertBefore(p,e)	o(n)
insertAfter(p,e)	o(1)
remove(p)	o(n)

Double Linked List

- List captured by referencing its sentinel nodes
- All its method are in O(1) as we don't have to go through the list to find the position for insertbefore(p,e)/ insertAfter(p,e)

size()	o(1)
isEmpty()	o(1)
first()	o(1)
last()	o(1)
before(p)	o(1)
after(p)	o(1)
insertBefore(p,e)	o(1)
insertAfter(p,e)	o(1)
remove(p)	o(1)

- Both operate o(n) if user want to find an element by index (does not allow random access)

Linked list	Array
- Efficient insertion and deletion - Simpler behaviour as collection grows - Don't have maximum capacity (space not wasted) - Good match to positional adt	- No extra memory for storing pointers - Allow random access - Less distance for program counter to move to next one - Good match to index-based adt

Iterator

- Snapshot freezes the contents of the data structure OR
- Dynamically follows changes to the data structure (behaviour changes predictably)
- iter(obj) returns an iterator of the object collection.
- __iter__(self) returns an object having next()method
- next(): return next object, advance cursor or raise StopIteration()

Example :

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```

- -> prints 1 to 20
- Next method can also raise StopIteration() under given condition.

Stack
- LIFO (last in first out)
- push(), pop(), top(), size(), isEmpty()
- Keep track of a history that allows undo ,
- Keep track of chain of active method, hence allow recursion
- Parentheses matching (push opening and pop when closing appears)
- Space :O(n), operations : O(1)

push(e)	o(1)
pop()	o(1)
top()	o(1)
size()	o(1)
isEmpty()	o(1)

Queue
- FIFO (first in first out)
- enqueue(e), dequeue(), first(), size(), isEmpty()
- Waitinglist, printer, multiprogramming
- Start: index of first element, last: index of last element
  - End = (start + size) mod N (N is the maximum capacity of the line)
  - Enqueue: last increment
  - Dequeue: first increment
- Space used : O(N), Operation time : O(1)

Trees
- Following the parent relation always leads to the root
- Root : node without parent (parent is none value)
- Internal node: node with at least a child
- External node: node with no children
- Ancestor : following a path up from a node, all node it came across (parent, parent of parent...) ('uncles' and 'aunties' don't count)
- Decenstor : following a path down from a node, all node it came across (child, grandchildren)
  - Ancestor / dancestor relation are transitive
  - All nodes are decestor of root
  - Every air of nodes have at least one common ancestor
  - Lowest common ancestor (lca) of x and y is z such that, z is ancestor of x and y and no descenstor of z has that property
- Siblings : two nodes with the same parent
- Depth of node: number of ancestors not including itself.
- Level : set of node with give path ({root} is level 0)
- Height of a tree : maximum depth
- Subtree : tree made up of some node and its decenstors
- Edge : pair of node such that one is a parent of another (so there is a path between)
- Path: sequence of node such that 2 consecutive node in the sequence have an edge

Ordered trees
- There is a prescribed order for each node's children
- E.g sections in chapters

Tree adt
- Generic
  - size()
  - isEmpty()
  - interator() -iterator
  - Positions () - iterator
- Access method
  - root()
  - parent(p)
  - children(p)
  - numchildren(P)
- Query method
  - isInternal(p)
  - isExternal(p)
  - isRoot(p)
- Node object : contain its value, the parent and its children

Traversing
- Preorder
  - Visit the node before visiting its decenstors

```
def pre_order(v)
  visit(v)
  for each child w of v
    pre_order(w)
```

  - Follow the order of the left side of the node been touched

- Post order
  - Visit the node after visiting its decenstors

```
def post_order(v)
  for each child w of v
    post_order(w)
  visit(v)
```

- Follow the order of the right side of the node been touched

Binary trees
- Each internal node has at most two children (left child and right child)
- The tree is proper if every internal node has two children
- Uses: classification, comparison (yes / no path)
- Extends tree operations with additional method
  - leftChild(p)
  - rightChild(p)
  - sibling(p)
  - If a node has null on both left and right, then it is external
- In order traverse:
  - The node is visited after its left subtree and

    ```
    def in_order(v)
      if v.left ≠ null then
        in_order(v.left)
      visit(v)
      if v.right ≠ null then
        in_order(v.right)
    ```

    before its right subtree

  - Order is the order of touching bottom of the node

Complexity
- Method call itself on all children : O(n) where n is number of nodes
- Method call itself on at most one child (worst case is do one call at each level): o(h) where h is height of the tree (go through each node when we don't know where the leaf is)

Binary search tree
- A binary tree where for every node, its left has value smaller than itself and right has value bigger than itself.
- Inorder traversal is able to visit the nodes in increasing order
- Implementation
  - all external node are null value
  - Search (k, v) : search down recursively by comparing value k with the value of node v, keeps going down until it reaches the equal value or external (null) value (which is unsuccessful search)
    - O(h)time: worst-case: h=n-1, best-case: h<=log2(n)
  - Put(k,o): If value k is in the tree, (found by search) replace the value to o, otherwise make an external node k and assign the value o to it. (add extra two external node after this)
  - remove(k) : find the node holding k (w) and delete it.
    - Case 1: w has 1 external child
      - Remove w and z from the tree
      - Make the other internal child of w take w's place
    - Case 2 : w has two internal children
      - Find the smallest node y among all right subtree under w (go right then all the way left down)
      - Replace w with y
    - O(n) space used, O(h) time (for both put and delete)
- range_search(T, K1, K2) : find all keys in T that is in [K1, K2]
  - Let p1 and p2 be paths to k1 and k2
  - Boundary node : node in p1 or p2
  - Inside node : node in [k1,k2] but not in p1 and p2
  - Outside node: node not in p1, p2 nor [k1,k2]
  - This algorithm only visits boundary and inside nodes
  - | inside node| <= |output|
  - | boundary node | <= 2* h
  - Run time : O(|output| + tree height)
  - put(k,o)

Rank-balanced trees
- O(log n) to perform a search

AVL tree
- Rank-balanced trees, r(v) = height of subtree rooted at v
- Balanced constrain: ranks of two children of every internal nodes differ by at most 1
- Height : O(log n)
- Searching, insertion, remove : O(log n)
- Insertion : to maintain the avl property, we need to do a single rotation or double
- rotation (take O(1) each) O(logn) total
- Rotation: three node, change the pointer such that the middle node becomes parent of the other two

Priority queue
- Store key-value items and can only remove the smallest key

	Unsorted list	Sorted list
insert(k,v)	O(1)	O(n)
remove_min()	O(n)	O(1)
min()	O(n)	O(1)
size()	O(1)	O(1)
is_empty ()	O(1)	O(1)

- Application : stock matching engines, price time priority
- Sorted list implementation insert list by first iterate and find the positions, so that it is able to find the smallest straight away
- Unsorted list insert straight but iterate to get the minimum key
- To sort priority in a list, both take O(n^2)

- As it needs to iteratively insert list to a queue and iteratively get minimum from the queue to the list
- Selection sort : scan through the list, find the minimum key after ith to swap it to the ith

i	A	s
0	7, 4, 8, 2, 5, 3, 9	3
1	2, 4, 8, 7, 5, 3, 9	5
2	2, 3, 8, 7, 5, 4, 9	5
3	2, 3, 4, 7, 5, 8, 9	4
4	2, 3, 4, 5, 7, 8, 9	4
5	2, 3, 4, 5, 7, 8, 9	5
6	2, 3, 4, 5, 7, 8, 9	6

```
def selection_sort(A):
    n ← size(A)
    for i in [0, n) do
        # find s ⩾ i minimizing A[s]
        s ← i
        for j in [i, n) do
            if A[j] < A[s] then
                s ← j
        # swap A[i] and A[s]
        A[i], A[s] ← A[s], A[i]
```

position. O(n^2), use unsorted list implementation

- Insertion sort : A[0,i) is the priority queue, A[i,n) is yet to be inserted O(n^2)
    - Each number moves forward until it hits the smaller key
    - I.e, each number, if bigger than x move forward one spot while leave its duplicate in the old spot, and replaced by the previous if previous is still bigger than x (previous is already smaller that next)
    - Insert to array

i	A	j
1	7, 4, 8, 2, 5, 3, 9	0
2	4, 7, 8, 2, 5, 3, 9	2
3	4, 7, 8, 2, 5, 3, 9	0
4	2, 4, 7, 8, 5, 3, 9	2
5	2, 4, 5, 7, 8, 3, 9	1
6	2, 3, 4, 5, 7, 8, 9	6

```
def insertion_sort(A):
    n ← size(A)
    for i in [1, n) do
        x ← A[i]
        # move forward entries > x
        j ← i
        while j > 0 and x < A[j-1]
            A[j] ← A[j-1]
            j ← j - 1
        # if j>0 ⇒ x ≥ A[j-1]
        # if j<i ⇒ x < A[j+1]
        A[j] ← x
```

Heap data structure

- Store key value as a node
- Not a binary search tree
- Heap order: all children of the node is larger than the node
- Complete binary tree: every level except for the last is full, last level (level h) nodes take leftmost position (last node is the rightmost node of maximum depth)
- Root always hold the minimum key
- Upheap: when insert a value, it start from the bottom and compare to move up to its position O(log n) if it is smaller than the root,update root, new level open
- remove_min() removes root
    - Swap the root key with the key of last node
    - Delete the last node (which was the root)
    - Restore heap order by swapping the root downwards
    - Set printer to new last node (O(log n))
    - O(log n)
- Heap sort is the version of priority-queue sorting that implements the priority queue with a heap runs in O(nlogn)
- Heap in array :
    - For the node at index i:
    - Its left child is at 2i +1
    - Its right child at 2i + 2
    - Its parent is at (i-1)/2 (ground)
- Summary

Size, isEmpty	O(1)
insert	O(log n)
min	O(1)
removemin	O(log n)
remove	O(log n)
replaceKey	O(log n)
replaceValue	O(1)

- Comparator(a,b) : if a occurs before b, return i < 0
- List based map
    - put(k,v) : O(1) if we know the key doesn't exist and insert and the beginning or end
    - Put, get, remove : O(n) worst case (we must travers to find the element or check existence)
    - Restricted keys
        - Uses and array of N size with keys in range 0 to N-1
        - Use keys as address (index) to get items
        - O(1) operation
        - Downside : takes big space
- Hash function and hash tables
    - Use hash function h to map keys to corresponding indices in an array
    - H is mathematical function and is efficient to compute
        - E.g h(x) = x mod N
        - h(x) is hash value of x
        - Non reversible : you cannot get x from h(x)
    - A hash table for a given key type K consists of
        - Hash function h: k → [0, N-1]
        - Array of size N
        - Ideally item (x,o) is at A[h(x)]
    - Hash function is composition of two functions:
        - Hash code : mapping key to integers
            - H1 : keys → integers
        - Compression function
            - H2 : integers → [0, N-1]
        - h(x) = h2(h1(x))
    - One way of hashing a string of elements is to use the sum
        - Horner's algorithm

Used on keys $k = (x_1, x_2, \ldots, x_d)$. For a given value of a we define

$$h(k) = x_1 a^{d-1} + x_2 a^{d-2} + \ldots + x_{d-1} a + x_d$$

        - So that different permutations of elements have different hash value
        - O(d) time to evaluate
    - Modular division
        - h(k) = k mod N for some prime number N
        - If keys are randomly distributed in [0, M] where M >> N then probability of two colliding is 1/N
    - Universal hash functions
        - Let h be a function based UAR from 2-universal family. Then the expected

number of collision for a given key k in a set of n keys is at most n/N
  - Randomly linear hash function
    - h(k) = ((a k + b) mod p) mod N
    - P is prime, a,b are chosen from [1,p-1]
    - If the keys are in the range [0,M] and p > M then the probability that two keys collide is 1/N
- Collision handling
  - Separate chaining
    - If there is a collide, append the new key at the back of existing one
    - Space O(n + N)
  - Linear chaining
    - If there is a collide, go the next cell, until it find an empty cell
    - If all full, make a new array with bigger size
    - Space O(n)
    - Open addressing : colliding item is put in another cell of the table
  - Cuckoo hashing

Graphs
- Vertices : a set of nodes
- Edges : collection of pairs of vertices
- Directed edge : ordered pair of vertices (u,v)
  - U is the origin / tail (where arrow is from)
  - V is the destination / head
- Undirected edge : unordered pair of vertices

Undirected graph
- Endpoints : points connected by edge
- Incidents on endpoints are edges that they connect to
- Adjacent : vertices that are connected
- Degree of vertices number of edges connect to it
- Parallel edges : share same endpoints
- Self-loop: have only one endpoint
- Simple graph : graph with no parallel or self loops

Directed graph
- Edges go from tail to head
- Out degree : number of edges out of a vertex
- In degree : number of edges into a vertex
- Parallel edges : edges share same tail and head (including self loop)
- Self-loop : tail and head are the same vertic
- Simple graph : no parallel or self-loops but allow anti-parallel loops (edges go opposite direction)
- Path : a sequence of vertices such that every pair of consecutive vertices is connected by an edge
  - Simple path : no repeated vertices
- Cycle : a path that starts and ends a the same vertex
  - Simple cycle : all vertices are distinct (except for the start as it also ends there)
- Subgraph : S is a subgraph of G when S has vertices and edges that are subsets from G
  - If you add an edge to S[e], you add both endpoints of that edge to S[v]

- Connectivity : a graph is connected if there is a path between every pair of its vertices
  - A connected component of a graph G is a maximal connected subgraph of G
  - Maximal connected : with max vertices and still can connected in that subgraph

Trees and forests
- Tree is connected graph with no cycles
  - Every tree on n vertices has n-1 edges
- Forest is a graph with its connected components been trees
- A spanning tree is a connected subgraph(tree) with the same vertices as the graph
  - a spanning tree is not unique if the graph is not a tree

Properties
- Sum of all degrees = 2m (m= # of edges)
- Simple Undirected : m <= n(n-1)/2 (n = # vertices)
- Simple Directed : m <= n(n-1)

Edge list structure
- Vertex sequence holds
  - Sequence of vertices
  - Vertex object keeps track of its position in the sequence
- Edge sequence
  - Edge object keeps track of its positions in the sequence
  - Edge object points at the two veteice it connected

Adjacency list

Each vertex also keeps a sequence of edges they incident on

Adjacency matrix

2d array adjacency matrix

Contain reference to edge object for adjacent vertices

Null for no adjacency

Summary

Asymptotic performance

▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n + m)$	$O(n + m)$	$O(n^2)$
incidentEdges(v)	$O(m)$	O(deg(v))	$O(n)$
getEdge(u, v)	$O(m)$	O(min(deg(u), deg(v)))	O(1)
insertVertex(x)	O(1)	O(1)	$O(n^2)$
insertEdge(u, v, x)	O(1)	O(1)	O(1)
removeVertex(v)	$O(m)$	O(deg(v))	$O(n^2)$
removeEdge(e)	O(1)	O(1)	O(1)

Graph traversal
- Depth first search (DFS)
  - Follow outgoing edge leading to unvisited vertices (otherwise backtrack)
  - DFS edge : edge used to visied a new vertex, otherwise it is a backedge

- If all edges from that vertex is visited we back to where it is from
- O(m+n)
- {(u,parent[u]) : u in Cv} form a spanning tree of Cv
  - Cv is the connect component of v in graph G
  - Parent[u] set of vertices that can reach u
- Identifying cutting edge
  - And edge is cutting edge if we remove this edge and the graph is not connected
  - Can Only test edges in a dfs tree of G O(nm)
  - Down and up method
    - For every vertex v, compute the highest level it can reach by taking one back edge up (and edges it yet visited)
    - If this level <= level of its parent u, then (u,v) is nota cutting edge
    - o(n+m)

DFS pseudocode

```
def DFS(G):

  # set things up for DFS
  for u in G.vertices() do
    visited[u] ← False
    parent[u] ← None

  # visit vertices
  for u in G.vertices() do
    if not visited[u] then
      DFS_visit(u)

  return parent
```

```
def DFS_visit(u):

  visited[u] ← True

  # visit neighbors of u
  for v in G.incident(u) do
    if not visited[v] then
      parent[v] ← u
      DFS_visit(v)
```

- Finding back edge : perform dfs, all edges left unvisited are back edges

Breadth-first search
- Visits all vertices at distance k from start vertex s

  Let C_v be the connected component of v in our graph G

  Fact: BFS(G, s) visits all vertices in C_s

  Fact: Edges { (u, parent[u]): u in C_s } form a spanning tree T_s of C_s

  Fact: For each v in L_i there is a path in T_s from s to v with i edges

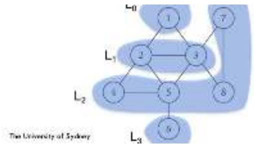
  Fact: For each v in L_i any path in G from s to v has at least i edges

  before visiting vertices at distance k+1

- o(n+m) for adjacency list
- Fact: A DFS edge (u, v) where u = parent[v] is not a cut edge if and only if down_and_up[v] ≤ level[u]
- o(n^2) for adjacency matrix

Summary

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	




```
# update current & next layers
current ← next
next ← []

return layers, parent
```

Weighted graphs
- Edge with numbers associated to it

Shortest path : minimum total weight of the edges from one point to another
1. A subpath of a shortest path is itself a shortest path
   a. If that subpath is not the shortest subpath, we can always replace it with one that is shorter
2. There is a tree of shortest paths from a start vertex to all the other vertices called shortest path tree

Dijkstra's algorithm
- Outputs distance from s to v (for all v in V) and shortest path rooted at s
- Assumption : graph is connected, undirected and all edge weights are positive

Idea
- Maintain in array D the upper bound distance from s to each v
- Keep track of a subset S such that for all v in S, d contain actual shortest path from s to v (D[v] lowered to its possible minimum)

Initialisation
- D[s]=0
- D[v]=+inf

Iteration
- Add u to S for the u with the smallest D[u]
- Update D values adjacent to u
- Edge relaxation :
  - e = (u, z) where u is the last vertex added in S and z is not yet in S
  - Relaxation of e updates D[z] to min{D[z], D[u] + w(u,z)}
- So that next shortest path is always sum of previous shortest paths
- Code:

```
def Dijkstra(G, w, s):

  # initialize algorithm
  for v in V do
    D[v] ← ∞
    parent[v] ← ∅
  D[s] ← 0
  Q ← new priority queue for { (v, D[v]) : v in V }

  # iteratively add vertices to S
  while Q is not empty do
    u ← Q.remove_min()
    for z in G.neighbors(u) do
      if D[u] + w[u, z] < D[z] then
        D[z] ← D[u] + w[u, z]
        Q.update_priority(z, D[z])
        parent[z] ← u
  return D, parent
```

- To follow a specific shortest path from v to s, follow parent reference back to s from v
- If all edges are weighted 1, we are running a bfs
- Run time
    - Initialisation : O(n) +
    - Iteratively add v to S : O(deg(v)) for each v
    - For connected graph :
    - m>=n-1
    - Run time : O(m) m is total edge (with out pq) Pq operation
    - Inserts: n
    - Decrease key : m
    - Remove min : n
    - Using heap for pq, algo runs in O(mlogn)
    - Using fibonacci heap with pq can carry decrease key opp to O(1), so O(m + n logn) in total
- Correctness

Dijkstra's Algorithm Correctness

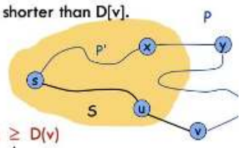
Invariant: For each $u \in S = V \setminus Q$, we have $D[u] = dist_w(s, u)$

Proof: (by induction on $|S|$)

Base case: $|S| = 1$ is trivial since $D[s] = 0$

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S and u=parent[v]
- The shortest s-u path plus (u, v) is an s-v path of length D[v]
- Consider any s-v path P. We'll see that it's no shorter than D[v].
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S:



$$w(P) \geq w(P') + w(x,y) \geq D(x) + w(x,y) \geq D(y) \geq D(v)$$

inductive hypothesis | Def of D(y) | Dijkstra chose v instead of y

The University of Sydney Page 31

Minimum spanning tree

- A spanning tree is a graph whose sum of edge weights is minimised
- Cut property : let S be a subset of nodes. And e be the min cost edge with exactly one endpoint in S, then MST contain e
    - Cutset : Subset of edges with exactly one endpoint in S
- Cycle property : let C be a cycle in the graph, and f be the edge with maximum cost, then f is not in MST
- A cycle and a cutset intersects an eve number of edges
    - If there is a way out of the subset, there must be a way back in

Prim's algorithm IDEA:

Prim's Algorithm

```
def prim(G, c):
  u ← arbitrary vertex in V
  S ← { u }
  T ← ∅
  while |S| < |V| do
    (u, v) ← min cost edge s.t. u in S and v not in S
    add (u, v) to T
    add v to S
  return T
```

- While tree don't span the entire graph, find the minimum edge in cutset and inset the outside endpoint to S
- Implementation :

Implementation: Prim's Algorithm

```
def prim(G, c) {

  for v in V do
    d[v] ← ∞
    parent[v] ← ∅
  u ← arbitrary vertex in V
  d[u] ← 0
  Q ← new PQ with items { (v, d[v]) for v in V }
  S ← ∅

  while Q is not empty do
    u ← delete min element from Q
    add u to S
    for (u, v) incident to u do
      if v ∉ S and c_e < d[v] then
        parent[v] ← u
        decrease priority d[v] to c_e
  return parent
```

Main idea: for every v in V \ S we keep
- d[v] = distance to closest neighbor in S
- parent[v] = closest neighbor in S

- O(mlogn) using heap
- O(m + n logn) using fibonacci heap
- Correctness : everytime we add an edge, we followed the cut property

Kruskal's algorithm

- Line up the edge in ascending order by their costs
- If adding e to T creates a cycle, discard e and move on (cycle property)
- Otherwise put e = (u,v) into T
- Sorting edges take O(m log m)
- Using dfs to check cycle takes O(mn)

Union find adt

- make_set(A): turn A into singleton sets with elements in A
- find(a) : returns an id for the set element a belongs to
- union(a,b) union the set a and b belongs to

Implementing union find in kruskal's algorithm

- If e = (u,v) does not make a cycle, that means u and v are in different sets
- Once we added e in T, union u and v in one set, meaning they are now one connected component

```
def Kruskal(G,c):

  sort E in increasing c-value
  answer ← [ ]
  comp ← make_sets(V)
  for (u,v) in E do
    if comp.find(u) ≠ comp.find(v) then
      answer.append( (u,v) )
      comp.union(u, v)
  return answer
```

- find(a) : 2m calls
- union(a,b) : n -1 calls
- Time: make_set : O(n)
- find(u) : O(1)
- union(u,v) : O(n)
- Total O(n^2) time

Tie Breaking

- Remove assumption that all edges are distinct
    - Add i/n^2 to each edge value or
    - Lexicographical order by index of edges

Greedy algorithm

- Build a solution one time at a time
- Make locally optimal choice every time and hope to get the the global optimal solution at the end
- Can be v hard to proof

```
def generic_greedy(input):
   # initialization
   initialize result

   determine order in which to consider input

   # iteratively make greedy choice
   for each element i of the input (in above order) do
     if element i improves result then

       update result with element i
   return result
```

Proofs

1. Exchange argument
2. Structural

Knapsack problem

- Given a set of n items, each item has a weight (+) and a benefit(+)
- Goal: to have maximum benefit in the bag after taken the maximum weight capacity
- Each step: identify the 'best' item and put it in the bag
    - 'Best' can be defined by weight, benefit, or weight/benefit
    - In this scenario, w/b is clearly the best

```
def fractional_knapsack(b, w, W):

 # initialization

x ← array of size |b| of zeros curr ← 0

 # iteratively do greedy choice

for i in descending b[i]/w[i] order do

 x[i] ← min(w[i], W - curr)

curr ← curr + x[i]

return x
```

- Proof : exchange argument
    - Consider some feasible solution (x) that is different to the one by greedy
    - Then there must be items i and k such that
        - Xi < wi , Xk > 0 and bi / wi > bk/wk
        - In another word, a more valuable item, i , that is not fully added in the bag, yet a less valuable item k is already in the bag
    - If we replace k with some i, we get a better solution'
    - The amount we replace depends on the amount we have for i and k
        - Min{wi-xi, xk}
        - Either we replace all item k, or we use up all item i
    - Thus there is no better solution than greedy
- O(n logn) : sort items, then O(n) to process the for loop

Task scheduling

- Given a set of n lectures and their start and finish time
- Goals: find the minimum no. of classrooms needed so that classrooms don't collide

```
def interval_partition(S):

   # initialization

sort intervals in increasing starting time order d ← 0 # number of allocated classrooms

   # iteratively do greedy choice
   for i in increasing starting time order do
     if lecture i is compatible with some classroom k then

       schedule lecture i in classroom 1 ≤ k ≤ d

  else

     allocate a new classroom d+1

     schedule lecture i in classroom d+1

     d ← d+1

return d
```

- n logn
- Keep the classrooms in priority queue
- Proof : structural
    - Let d be the number of classrooms the greedy algorithm allocates
    - This, by the definition in algo means that, classroom d is open due to having incompatible job with all other d-1 classrooms
    - This means we have d-1 lectures starting before si(start time of this lec)
    - Thus we have now d lecture overlapping at some point

- All schedules use >= d classrooms
- Thus the algorithm is indeed correct

Text compression
- Given a string x
- Goal: encode it into smaller bits
- Huffman encoding
  - Encode higher frequency characters with shorter bits
  - No encode is prefix of another (no encoding is the start of another) – achieved by a tree
  - Only the leaf (external node) hold a character
  - o(n + dlogd) where d is the number of distinct characters (|C|)
  - O(|C| log |C|) if use heap
  - String length

$$\sum_{c\ in\ C} f(c) * depthT(c)$$

```
def huffman(C, f):
 # initialize priority queue
Q ← empty priority queue for c in C do
T ← single-node binary tree storing c Q.insert(f[c], T)
 # merge trees while at least two trees
while Q.size() > 1 do
f1, T1 ← Q.remove_min()
f2, T2 ← Q.remove_min()
T ← new binary tree with T1/T2 as left/right subtrees f ← f1 + f2
Q.insert(f, T)
 # return last tree
f, T ← Q.remove_min() return T
```

- Pick two least frequent tree, merge them together, add the tree frequent value to the sum of the two, and put new tree back to queue
- Every encoding tree has a pair of leaves that are siblings
- If depth of character a is less than b (a is closer to the root) , then a is more frequent than b
- If we combine the two least frequent character (e and f) and make a new tree T', THEN expanding T' will also give optimal T and the different in length is f(e) + f(f)
- Proof :

Proof (by induction):
- If |C| = 1 then the encoding is trivially optimal
- If |C| > 1 then let (C', f') be the contracted instance
- By inductive hypothesis, the encoding tree T' constructed for (C', f') is optimal
- Recall that

$$\sum_{c\ in\ C} f(c) * depth_T(c) = \sum_{c\ in\ C'} f'(c) * depth_{T'}(c) + f(i) + f(k)$$

- Since f(i) and f(k) are the minimum amount, we can extend after expanding the tree, T is therefore also optimal

"Divide and conquer"

1. Divide : solve if it is a base case, otherwise divide problem up into parts
   a. Typical base case are when the size of input is 0 or 1
2. Recur / delegate : recursively solve each problem
   a. Similar to induction hypothesis
   b. We assume it works by applying the recursion
3. Conquer : combine the solved solution of each parts into an overall solution

- Time complexity:
  - For n > 1: T(n) = recur + divide and conquer
  - For n=1:T(n)=basecase
  - Divide step : time in terms of n
  - Recur step : time in terms of T (and HOW MANY TIMES WE RECURSE)
  - Conquer step : time in terms of n

Searching sorted array
- Given a sorted array A, find if number x is in this array
- **Binary search**
  - Base case : if array is empty, return no
  - Compare x to A[n/2], if they are equal return yes
  - Else if A[n/2] > x, recursively search [0, n/2] in the array, otherwise, recursively search [n/2, n]
- **Correctness**
  - The correctness follows that , if x is in A before divide, x is in A after divide
    - And if x is greater than the middle point, then x must be the later half
    - Or vice versa
  - Every divide step leads to smaller array
    - Is x is in A, we'll eventually reach x due to invariant and return yes
    - Otherwise, we'll reach an empty array and return no
- Time complexity

  T(n)=T(n/2)+O(1) for n>1
  T(n)=O(1) for n=1

  - There will be logn amount of time we iterate T(n) - > T(n/2^k) to T(1)
  - Each time, we add a cost of 1 (as the dived and conquer takes O(1)
  - We have to add logn amount of those
  - So final cost is O(LOGn)
- Linked list application
  - Catch : we now can not call the A[n/2] place in O(1) AS for linked list, we have to iterate the list to find a position since they don't have a index
  - In this case

- Divide : O(n) \
- Recur : T(n/2)
- Conquer : O(1) (returning answer from recursion)

- T(n) = T(n/2) + O(n) , T(n) = O(1)
- Which solves to O(n) where expanding above gives:
  - n + n/2 + n/4 + ... + 1

Merge sort

1. Divide the array into halves
2. Recursively sort each half
3. Conquer two sorted halves together to make one

```
def merge_sort(S):
  # base case
  if |S| < 2 then

    return S

# divide

mid ← ⌊|S|/2⌋

left ← S[:mid]  # doesn't include S[mid]

right ← S[mid:] # includes S[mid]

# recur

sorted_left ← merge_sort(left)

sorted_right ← merge_sort(right)

# conquer

return merge(sorted_left, sorted_right)
```

array (merge)

Merge :

- Given two sorted arrays
- Iterate through each item in the two array, compare and put the smaller one in during each iteration
- Repeat until one array is empty
- Insert the left over item in the next array in

```
def merge(L, R):

  result ← array of length (|L| + |R|)

  l,r ← 0,0

while l + r < |result| do
    result[index] ← L[l]

    l←l+1                        < R[r]) then

  else

    result[index] ← R[r]

    r←r+1

return result
```

Merge correctness

- Inductive hypothesis : After the ith iteration, our result has the ith smallest element in the final array
- Base case: after o iteration, array is empty, so it contains the 0 smallest elements in sorted order
- Prove true for k + 1 th iteration
  - Since both arrays are sorted, adding the smallest element that is not yet in the array indeed gives the K +1 th largest item (assume inductive hypothesis)
  - correctness follows from the two given array are sorted

Merge sort correctness

- Merge sort correctly sorted an array of size i
- If array is of size 0 or 1. It is already sorted
- Prove true for array size K + 1
  - Splitting the arrays give each at most a size of k (inductive hypothesis)
  - By inductive hypothesis, those are all sorted
  - As proved above, merge will merge correctly for the two splitted arrays
  - Hence, running the merge on the two halves will indeed sort the array

Time cost

- Divide : O(n)
- Recur : 2*T(n/2) (since we are dividing two arrays and working on both)
- Conquer O(n) (merging step)
- T(n) = 2T(n/2) + O(n) ,
- T(n)=O(1) for n=1
- Final cost : O(nlogn)
  - In this case, the recurrence relation can be rewritten as:
    T(n) = 2T(n/2) + O(n).
    The work done at each level of recursion is O(n), and the recursion depth is log(n) because we divide the input size by 2 at each level until we reach the base case of n = 1.
    Therefore, the total work done can be calculated as:
    Total work = O(n) * log(n) = O(nlog(n))

Solving recurrence by unrolling

- Analyse and identify pattern from first few levels of expansion
- Use the pattern to sum up over all levels
- Cheat sheet: (need to prove if use)

Recurrence	Solution
2T(n/2) + O(n)	O(nlogn)
2T(n/2) + O(log n)	O(n)
2T(n/2) + O(1)	O(n)
T(n/2) + O(n)	O(n)
T(n/2) + O(1)	O(log n)
T(n-1) + O(n)	O(n^2)
T(n-1) + O(1)	O(n)

Quick sort

- Divide : choose a random pivot partition x from the array, divide the array to three sub arrays : less than x, equal to x , greater than x O(n)
- Recursively sort the three arrays T(nL) + T(nR)
- Conquer : join the three arrays together O(n)
- E[T(n)] = E[T(nL) + T(NR)] + O(n)
- Expected time : O(nlogn)
- Worst case : all value selected is the smallest value in the array during each recurring
    - T(n) = T(n-1) + O(n)
    - Worst time = O(n^2)

Comparison sorting lower bonding

- By performing pairwise comparisons between two elements we are trying to sort
- Like all we had above
- Such algo can be seen as a decision tree
    - Internal node : compares two indices of the input array
    - External node : permutation of {1..n}
- The height of such decision tree is the lower bound of the run time (cannot be faster
- than the height)
- Ω(nlogn)
- Since the decision tree has n! External nodes, thus the height is logn!
- and logn! = nlogn

Maxima set

Goal : find the point that all other points have either a smaller x coordinate or smaller y coordinate than the point
Idea : check every point, one at a time to see if the other point have either smaller x or smaller y

Pre-processing : sort all the points by its x coordinate
Divide : split the sorted array in halves
Recur : recursively find the max set on each halves
Conquer : compute the union of left and right max set

Observation

- Every point in right ms is in the whole set, because although it ignores the left, all points in the left (sorted by x coord) are smaller than itself so satisfies at least one of the condition already
- Every point in left ms is either in the whole set or is dominated by the highest point in the right

Conquer

- Find the highest point in the right ms : p
- Compare every point in left ms to q
- Add only the once that are higher than p

Time complexity

- T(n) = 2T(n/2) + O(n) = O(nlogn)

Unrolling

- For each step / level, calculate time taken for each

  Let r be a positive real and k a positive integer then
  $$1 + r + r^2 + ... + r^k = (r^{k+1} - 1)/(r-1)$$

  Consequently if r > 1 then
  $$1 + r + r^2 + ... + r^k < r^{k+1} / (r-1)$$

  and if r < 1 then
  $$1 + r + r^2 + ... + r^k < 1 / (1-r)$$

  level and then add them up

Master theorem

T(n) = aT(n/b) + f(n),
where,
n = size of input
a = number of subproblems in the recursion
n/b = size of each subproblem. All subproblems are assumed to have the same size.
f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

T(n) = aT(n/b) + f(n)

where, T(n) has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$.

3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n))$.

ϵ > 0 is a constant.

Solved example

T(n) = 3T(n/2) + n2
Here,
a = 3
n/b = n/2
$f(n) = n^2$

$\log_b a = \log_2 3 \approx 1.58 < 2$

ie. $f(n) < n^{\log_b a+\epsilon}$, where, ϵ is a constant.

Case 3 implies here.

Thus, $T(n) = f(n) = \Theta(n^2)$

Algorithm Analysis

Primitive Operations - $O(1)$

Common Running Times

1. 1 - constant
2. logn - logarithmic
   - ↳ $\log_2 n > \log_8 n$
3. $\log^2 n = \log(\log n)$
   - ↳ polylogarithmic
4. $\sqrt{n}$ - square root
5. n - linear
6. nlogn - quasilinear (linearithmic)
7. n^2 - quadratic
8. $n^2 \log n$
9. n^3 - cubic
10. 2^n - exponential
11. $2!$ - factorial

Big-Oh Notation => upper bound on RT
- n^x is $O(a^n)$ for any fixed $x>0$ + $a>1$.
- $\log n^x$ is $O(\log n)$ for any fixed $x>0$
- $\log^x n$ is $O(n^y)$ for any fixed con. $x, y>0$

Big-Omega Notation (Ω) => lower bound

Big-Theta Notation (Θ) => asymptotically tight bound

Transitivity:

If $f = O(g)$ + $g = O(h)$, then $f = O(h)$

Sums of Functions:

If $f = O(g)$ + $g = O(h)$, then $f(g) = O(h)$

Log Properties

- $\log_b a = \frac{1}{\log_a b}$
- $\log_b c = \frac{\log_a c}{\log_a b}$
- $a = b^{\log_b a}$
- $b^{\log_c a} = a^{\log_c b}$

Lists

- Abstract Data Type: desired behaviour

Data Structure - concrete rep.

Index-based List (List ADT) isEmpty()
=> size(), get(i), set(i,e), add(i,e), remove(i,e)

Array-based List element stored at A[i]
set(), get() => O(1) ind. of size
add(), remove() => O(n) => shifting n elem.
space: O(N) ↱ change size as you add

Dynamic Array Space: O(n)

Positional List - points to element
=> first(), last(), before(p), after(p), insertBefore(p,e), insertAfter(p,e), remove(p)

Singly Linked List - reference to first node
=> insertBefore O(n)

Doubly Linked List - link to element, prev, next
↳ has a header + trailer => Space: O(n)
↳ all ops - O(1)

Stack - Last in, first out ↗ push() ↘ pop() - remove last inserted
↳ top(), size(), isEmpty()
Based on Arrays -> space O(n); ops O(1)

Queue - First in, first out -
↳ enqueue(e): insert at end; dequeue() remove at front; first(), size(), isEmpty()
↳ based on arrays: end = (start+size) mod N

Double-ended queue (Deque) space: O(N)
↳ allow insertions + deletions @ both ends
↳ getFirst/Last, addF/L, removeF/L => O(1)

Trees (Tree ADT)

- node has at most 1 parent
- Root: node w/o parent
- Internal node: node w/ at least 1 child
- External/Leaf node: node w/o children
- Ancestors, Descendants, Siblings
- Depth of a node: # of ancestors not incl. itself
- Height: max depth
- Level: set of nodes w/ a given depth
- Edge: pair (u,v) where one is the parent

Ordered Tree: has prescribed order

Pre-order: visit node before descendants

Post-order: visit node after descendants

Binary Tree - each node has at most 2 ch.
↳ proper BT: every internal node has 2 ch.

Inorder: node visited after L, before R

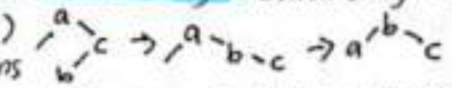
Euler Tour Traversal: visit each 3x
- on the left (pre), from below (in), on the right (post)

Linked Structure for BT
- node: element, pointers to parent, L, R

Linked Structure for general trees
- node: element, pointer to parent, seq. of children

Binary Search Tree (BST)
=> get(), put(k,v), remove(k)
=> **Sorted Map ADT**: keys have sorted order
=> key(u) < key(v) < key(w) u=L w=R
=> inorder visits BST in ↑ order
=> internal nodes store key-value pairs
=> external nodes do not store items
=> **search**: compare key stored at node to given key to decide whether go L/R
↳ runs in O(h); worst case O(n)
O(logn) for balanced trees
=> **insertion**: if present, replace value. oth., expand node by replacing external node w/ new key-value pair

=> Deletion: if node has 1 child, promote the child and replace the node.
If node w has 2 children, find node y following w in an inorder traversal. y would have no children. replace w w/ y and remove y.
=> complexity: space $O(n)$ ops: $O(h)$
=> Duplicate keys: $key(L) \le key(node) < key(R)$
↳ use list to store duplicates
↳ Range queries: search all keys K such that $K_1 \le K \le K_2$
> $key(v) < K_1 \to R$ > $key(v) > K_2 \to L$
> $K_1 \le key(v) \le K_2 \to L$, add v, R
> running time: $O(|output| + h)$

Trinode Restructuring → Balancing Trees
- (a,b,c)
2 rotations

- takes $O(1)$ since you just have to update 3

Rank-balanced Trees - keep a rank for every
AVL Trees $r(v)$ is height of subtree rooted at v; ranks of 2 children of every internal node differ by at most 1
> height: $O(\log n)$; space $O(n)$; ops $O(\log n)$

Priority Queue: can only remove the smallest key
> insert(k,v), remove-min(), min()

Sequence-based PQ

	Unsorted	Sorted (by priority)
size, isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
remove-min, min	$O(n)$	$O(1)$

Priority Queue Sorting - insert keys - remove-min to sort
- n insert, n remove-min
- so $O(n^2)$

Selection sort - sort end first
> sort unsorted sequence > can be in-place
> $A[0,i)$: sorted; $A(i,n]$: PQ
> n inserts - $O(n)$; n remove-min - $O(n^2)$
> look at the smallest after given index + then swap places

Insertion sort: sort front first
> Best: sorted asc; worst: sorted desc.
> n insert → $O(n^2)$; n remove-min → $O(n)$
> $A[0,i)$: PQ (sorted); $A[i,n)$: rest
> find if there's value greater before given index and the insert index before greater value

Heap: BT where pointers are only at root + last inserted node
> heap-order: $key(m) \ge key(parent(m))$
> complete BT: every level $i < h$ is full
↳ remaining nodes take leftmost pos.
> root has smallest key, height is $\log n$
> Upheap: restore h-o by swapping keys along upward path from insertion pt.
↳ $O(\log n)$ => find position: go up then right
> remove-min: replace root key w/ last inserted; restore h-o by downheap
↳ swap keys along downward path from root. → $O(\log n)$

Heap PQ - min $O(1)$; insert, removemin $O(\log n)$
Heap sort - $O(n \log n)$ => PQ sorting w/ heap
Heap-in-array - root at index 0 of array
- last node at index $(n-1)$
- node at index i: $L \to 2i+1$; $R \to 2i+2$
↳ parent → $\lfloor (i-1)/2 \rfloor$

Hash Tables
Map - searchable collection of k,v pairs
=> get(), put(), remove, size()
=> put: if key already present, replace value
List-based map - based on doubly linked
=> put - $O(1)$; get, remove $O(n)$
Map w/ restricted keys - use keys as index; map w/ n items + N keys
ops are all $O(1)$ but N can be big (space)
Hash Tables - use a hash function h to map keys to corresponding indices in array A w/ fixed interval $[0, N-1]$
- integer $h(x)$ is the hash value of key x
- ideally, there should be no collisions (items stored at the same hash value)
- hash function h is usually a composite of 2 functions: $h(x) = h_2(h_1(x))$
> hash code h_1: keys → integers
> compression func. h_2: integers → $[0, N-1]$
- Probability of collision: $1/N$ where N is prime
Separate chaining → add. space
↳ create a list w/in hash value
↳ Load factor $\alpha = n/N$
↳ expected: $O(1+\alpha)$; worst-case: $O(n)$ when all items collide to a single chain

Open Addressing using Linear Probing

open addressing: colliding item placed in diff cell of the table

Linear probing: place colliding item in the next (circularly) available cell
↳ colliding items lump together

search: start at cell h(k); probe consecutive locations until an item w/ key k is found or empty cell is found or N cells have been probed.
↳ DEFUNCT replaces deleted elements
get(k) must pass over cells w/ DEFUNCT and keep probing

↳ put: If k is found, replace value. otherwise, store it at index i which is the index w/ the first DEFUNCT

Performance: wc: get, put, remove $O(n)$
If randomly distributed, expected # of probes is $1/(1-\alpha)$ where $\alpha = n/N$
If α is constant < 1, expect rt is $O(1)$

Cuckoo hashing - use 2 hash function and 2 hash tables
↳ get, remove → $O(1)$
↳ evict previous item + insert new; the evicted goes to its other possible place
Eviction cycle: keep counter/put flag

Set - unordered collection of elements w/o duplicates; ops are traditional set operations: union, contains, etc.

Set implemented via map
- map to store keys; ignore value
- contains(k) answered by get(k)

Graph - consists of a pair (V, E)

Edge (E): directed (u,v) → v origin/ u→v tail
undirected (u,v) u–v v–head/dest.

Undirected Graphs - edges connect endpts
- edges are incident on endpts
- adjacent vertices are connected w/ an e
- degree = num edges on a vertex
- parallel edges share same endpts
- self-loop: only 1 endpt
- simple graphs: no parallel/self-loops

Directed Graphs - edges: tail to head
- out degree: num edges out of vertex
- in degree: num edges into a vertex
- parallel: share tail + head
- self-loop: same tail + head
- anti-parallel: same endpt but opp. tails or heads
- simple graphs: no parallel or self-loop but can have anti-parallel edges

Path: seq. of vertices
↳ simple path: all vertices are distinct

Cycle: path that starts + ends at the same vertex; can revisit same vertex but not same edge
↳ simple cycle: all vertices are distinct

Properties of a graph - $\sum_{v \in V} \deg(v) = 2m$
- n (# of v); m (# of e); Δ max degree
- simple undirected - $m \le n(n-1)/2$
- simple directed: $m \le n(n-1)$

Subgraph: Let $G = (V, E)$ be a graph. $S = (U, F)$ is a subgraph of G if $U \subseteq V$, $F \subseteq E$

Subset: subset $U \subseteq V$ induces a graph $G[U] = (U, E[U])$ where $E[U]$ are edges in E w/ endpts in U. subset $F \subseteq E$ induces a graph $G[F] = (V[F], F)$ where $V[F]$ are endpts of edges in F.

Edge List structure v: a E: a
- vertex List: seq. of vertices, vertex objects keep track of its pos. in the seq; points to the vertices
- Edge list: seq edges; edge objects keep track of its position in the seq; points to the edges + the endpts

Adjacency List
- each vertex keeps a seq. of edges adj. to it
- edge objects keep ref to pos in the incidence seq. of its endpts
- good for sparse graphs

Adjacency Matrix
- 2D array: ref to edge object for adj vertices; null for nonadj vertices
- good for dense graphs

Performance	edge list	Adj. List	Adj. Matrix
space	n+m	n+m	n^2
incidentEdges(v)	m	deg(v)	n
getEdge(u,v)	m	min(deg(u), deg(v))	1
insertVertex(x)	1	1	n^2
insertEdge(u,v,x)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

Connectivity - connected if there is a path btw. every pair of vertices in G
- connected component of G: maximal connected subgraph of G

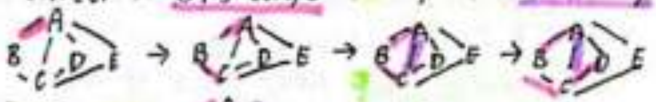
Tree (T): T is connected; T has no cycles

Forest: graph w/ no cycles, its connected components are trees

* Every tree on n vertices has n-1 edges

Spanning Tree + Forest
- Subset of graph G which has all the vertices covered w/ min edges; no cycles
- can't be disconnected

Depth First Search - follows outgoing edges leading to yet unvisited vertices
- If edge discovers a new vertex, it's called a DFS edge. Oth., it's a back edge



Performance assuming adj list rep

: $O(m+n)$
- main DFS function: visits all vertices $O(n)$
- DFS-visit(u): $O(deg(u))$ - depends on deg. of u
  - called u times so $O(\sum_u deg(u)) = O(m)$

Prop. of DFS. Let C_v be the conn. comp of v
- DFS-visit(u) visits all vertices in C_v
- Edges {(u, parent[u]) : u in C_v} form a span tree of C_v
- Edges{(u, par[u]) : u in V} form a span forest of G

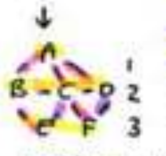
Cut Edges: In a connected graph, $G=(V,E)$, edge (u,v) in E is a cut edge if $(V, E\setminus\{(u,v)\})$ is not connected
- $O(m^2)$: For each edge, remove (u,v), check if DFS is still connected, put back (u,v)
- $O(nm)$: only test edges in a DFS tree of G
- $O(n+m)$: compute DFS tree of G, for every v in V, compute level[v]

down-and-up[v]: height of vertex v that can be reached by taking DFS tree down the and then one back edge up

* DFS edge (u,v) is a cut edge where u is a parent[v] $\Leftrightarrow$ down-and-up[v] $\geq$ level[u]

Breadth First Search - identify all objects in each layer first: $L_0 = \{s\}$; L_1 = vertices one hop away from S

discovery edge; cross edge



Properties: BFS visits all v in C_s
1. Edges {(v, parent[v]) : v in C_v} form a span tree T_s of C_v
2. For each v in L_i, there is a path in T_s from s to v with i edges.
3. For each v in L_i, any path in G from s to v has at least i edges.

Performance: setting things up $O(n)$

processing each layer $O(\sum deg(u)) = O(m)$
- adj. list - $O(m+n)$; adj matrix $O(n^2)$

Applications: DFS + biconnected comp

BFS: shortest paths; Both: cycles, paths

Weighted Graph: each edge has a weight

Greedy algo: build a soln one step at a time; making locally optimal choices at each stage in the hope of finding a global optimal soln. → sum is the min.

Shortest Path: subpath of s.p is a s.p
- there is a tree of shortest paths from a start vertex to all other vertices

Dijkstra's Algo: G is connected + undirected
- edge weights are nonnegative
- maintain a distance estimate + keep track of the actual distance (sum of weights)
- Initially D[s] = 0; D[v] = ∞ for all v in V - s
- In each iteration, add to S vertex u in V\S with smallest D[u]; update D-values for vertices adj. to u

Performance: $O(m)$ on everything except PQ operations: $O(m + n\log n)$

heap as PQ: $O(m\log n)$, Fibonacci heaps: $O(m + n\log n)$

Minimum Spanning Tree - tree whose sum of edge weights is minimised.

Properties: all edge costs c_e are distinct
- Cut property: Let S be any subset of nodes and let e be the min cost edge w/ exact 1 endpt. in S. Then, the MST contains e
- Cycle property: Let C be any cycle and let f be the max cost edge belonging to C. Then, f must not be in MST.
- Cut: nonempty $S \subseteq V$; cutset D is the subset of edges w/ exactly 1 endpt
- Cycle + cutset intersect in an even # of edges. → Every time, we add an edge, we follow cut property.

Prim's Algo We add the min cost edge (u,v) s.t u in S and v not in S. Start w/ any node and update distances of adj. selected nodes. Select min + repeat (use table)

- For every v in V\S, we keep the distance to closest neighbour and the closest neighbour in a table
- similar time complexity as Dijikstra

Kruskal's Algo - consider edges in asc. order of weight.
- If adding e to T creates a cycle, discard e. Otherwise, insert e.
- Choose edges based on order of weights → no need for lists/tables

Lexicographic Tiebraking: assuming all costs are integral, if we add i/n^2 to each edge e_i then any MST under the perturbed weights is still an MST under orig. weights.

Time Complexity: sorting edges : $\Theta(m\log n)$
- Test if cycle occurs : $\Theta(mn)$

Union Find ADT - keep track of an evolving partition of A.
- Ops: make-sets(A), find(a) union(a,b)

- Simple union-find
- make-sets(A) $\Theta(n)$ where $n = |A|$
- find(u) $\Theta(1)$; union(u,v) → $\Theta(n)$
- Kruskal's algo : $O(n^2)$ ⇒ find : 2m calls, union : n-1 calls

Better Union-Find
- keep track of cardinality of each set. When taking union of 2 sets, change the smallest. Element can change sets $O(\log n)$; seq. of n union ops $O(n\log n)$
- Kruskals : $O(m\log n)$

Greedy Algo

Fractional Knapsack - given a set S of n items w/ each item i having b_i (+ve benefit) + w_i (+ve weight), choose items w/ max total benefit of weight at most W
- best : items w/ highest b_i/w_i ratio.
- complexity : $O(n\log n)$ for sorting (use PQ heap so each removal takes $O(\log n)$) + then $O(n)$ to process in for loop

Task Scheduling - given a set of n lectures. Lecture i starts at s_i and finishes at f_i, find min # of classrooms to sched all lectures s.t. no 2 occur @ same time + place

Interval Partitioning
- Sort intervals by starting time
- when space is available, put it in a classroom w/o caring about which is best. Oth, open new classro.
- $O(n\log n)$ - for each room k, maintain the finish time of last job added. Keep classrooms in PQ

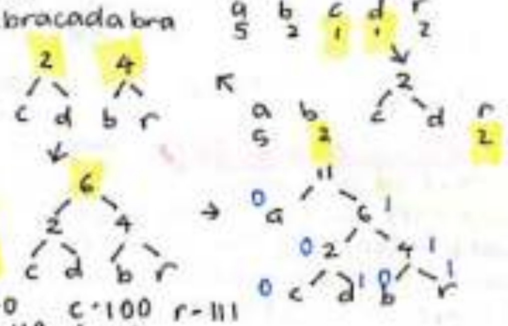
Text compression - given string X, efficient encode it to smaller Y

Runlength encoding - encode based on # of characters (eg. 12W1B8C)

Huffman encoding - Let C be the set of characters in X. Compute freq. f(c) for each character c in C. Encode high freq. char. w/ short code words. No code words is a prefix for another code.

Encoding Tree - code : mapping of each char. to a binary code word
- each external node stores a char.
- code word : path from root to external node (0→L; 1→R)
- put smaller tree as left child when combining trees
- Huffman's Tree : $O(n + d\log d)$ where n is the size of X and d is the # of distinct char. of X
- builds tree from bottom up
- $\sum_{c \in C} f(c) \times \text{depth}_T(c)$

X - abracadabra



a - 0 c - 100 r - 111
b - 110 d - 101

- Every tree encoding has a pair of leaves that are siblings
- For any a + b in C, if $\text{depth}_T(a) < \text{depth}_T(b)$ then $f(a) \geq f(b)$.
- 2 siblings furthest from root have lowest freq.
- If we combine 2 lowest freq. char. to get a new instance (C', f'), an optimal tree for C' can be expanded to get optimal tree for C
- In each op, 2 remove, 1 insert. TC dominated by PQ ops which take $O(\log n)$ if PQ; oth, calculating freq. is $O(n)$.

Divide and conquer - divide, recurse, conquer OR divide, conquer, merge

Binary search - If array empty, "No". Otherwise compare x to middle element $A[\lceil \frac{n}{2} \rceil]$. If $A[\lceil \frac{n}{2} \rceil] > x$, search $L \rightarrow A[0]$ to $A[\lceil \frac{n}{2} \rceil - 1]$. Else, if $A[\lceil \frac{n}{2} \rceil] < x$, recur. search $R \rightarrow A[\lceil \frac{n}{2} \rceil + 1] \rightarrow n$

Recurrence - Divide $\rightarrow O(1)$, Recur $\rightarrow T(\frac{n}{2})$, conquer $\rightarrow O(1)$ $T(n) = \begin{cases} T(\frac{n}{2}) + O(1) & n>1 \\ O(1) & n=1 \end{cases}$

$T(n) = O(\log n)$

$0: c$
$1: T(n) = T(\frac{n}{2}) + c$
$2: T(n) = T(\frac{n}{4}) + 2c$
$i: T(n) = T(\frac{n}{2^i}) + ic$

$\frac{n}{2^i} = 1 \Rightarrow i = \log n$
$\log n: T(n) = 1 + c\log n$
$T(n) = O(\log n)$

* Recurrence formula includes recur + conquer

If linked list used: $T(n) = \begin{cases} T(\frac{n}{2}) + O(n) & n>1 \\ O(1) & n=1 \end{cases}$ $\hookrightarrow O(n)$

Merge Sort

- Divide into 2 halves, recurse on both + merge
- keep track of smallest element in each sorted half. insert smallest of 2 elements into array. Repeat until both lists are merged
- divide: $O(n)$; recur: $2T(\frac{n}{2})$; conquer $O(n)$

$T(n) = \begin{cases} 2T(\frac{n}{2}) + O(n) & n>1 \\ O(1) & n=1 \end{cases} \Rightarrow T(n) = O(n\log n)$

Sample Recurrence Formulae

$T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow O(n\log n)$
$= 2T(\frac{n}{2}) + O(\log n) \Rightarrow O(n)$
$= 2T(\frac{n}{2}) + O(1) \Rightarrow O(n)$
$= T(\frac{n}{2}) + O(n) \Rightarrow O(n)$
$= T(\frac{n}{2}) + O(1) \Rightarrow O(\log n)$
$= T(n-1) + O(n) \Rightarrow O(n^2)$
$= T(n-1) + O(1) \Rightarrow O(n)$ $\Uparrow O(n)$

Quick sort 1. Divide. Choose random element as pivot. Partition into 3: (i) < (ii) = (iii) >
2. Recursively sort < + > lists $- T(n_L) + T(n_R)$
3. Conquer. Join 3 lists together $\Rightarrow O(n)$ in place

$E[T(n)] = \begin{cases} E[T(n_L) + T(n_R)] + O(n) & n>1 \\ O(1) & n=1 \end{cases} \Rightarrow O(n\log n)$

* comparison-based sorting takes $\Omega(n\log n)$

Maxima-set - A pt is max if all other pts in the set either have a smaller x or y coor.
1. Preprocessing. sort pts by $\uparrow x$ and break ties using y. Store in an array
2. Divide sorted array into 2 halves
3. Recursively find MS in both halves
4. Conquer. Compute MS of $MS_L \cup MS_R$
$\hookrightarrow$ Find highest pt. p of MS_R. Compare every pt q in MS_L to p. If $q_y > p_y$, add q to merged MS. Add every pt in MS_R to merged MS. $T(n) = 2T(n/2) + O(n) = O(n\log n)$

① $O(n\log n)$ ② $O(n)$ ③ $2T(\frac{n}{2})$ ④ $O(n)$
R.T $= O(n\log n)$

Integer multiplication Given 2 n-digit integers $x + y$, compute the product xy.

ATTEMPT 1: Compute by making 4 recursive calls on $\frac{n}{2}$ digit numbers + combining it

$x = x_1 2^{n/2} + x_0$; $y = y_1 2^{n/2} + y_0$

$xy = x_1y_1 2^n + x_1y_0 2^{n/2} + x_0y_1 2^{n/2} + x_0y_0$

$T(n) = \begin{cases} 4T(\frac{n}{2}) + cn & n>1 \\ c & n=1 \end{cases} \Rightarrow O(n^2)$

ATTEMPT 2: compute by making 3 rec. calls

$xy = x_1y_1 2^n + (x_1y_0 + x_0y_1) 2^{n/2} + x_0y_0$

$(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$

$T(n) = \begin{cases} 3T(\frac{n}{2}) + O(n) & n>1 \\ O(1) & n=1 \end{cases} \Rightarrow O(n^{\log_2 3})$

$0: O(1)$
$1: 3T(\frac{n}{2}) + n$
$2: 3(3T(\frac{n}{4}) + \frac{n}{2}) + n = 9T(\frac{n}{4}) + \frac{3n}{2} + n$
$i: 3^i T(\frac{n}{2^i}) + \sum_{i=0}^{i} (\frac{3}{2})^i n$

$\frac{n}{2^i} = 1 \Rightarrow i = \log n$

$\frac{3}{2} = 2^{\log_2(\frac{3}{2})}$

$\log n: 3^{\log n} + \sum_{i=0}^{\log n} (\frac{3}{2})^i cn = O(3^{\log n} + (\frac{3}{2})^{\log n} cn)$
$= O(3^{\log_2 n} + 2^{\log_2(\frac{3}{2}) \cdot \log_2 n} n)$
$= O(3^{\log_2 n} + 2^{\log_2 n^{\log_2(\frac{3}{2})}} n)$
$= O(n^{\log_2 3} + n^{\log_2 3/2} n) = O(n^{\log_2 3} + (n^{\log_2 3} \cdot \frac{1}{n} \cdot n))$
$= O(n^{\log_2 3})$

Geometric series Let $R \in \mathbb{R}^+$, $K \in \mathbb{Z}^+$

$1 + r + r^2 + \dots + r^k = (r^{k+1} - 1)/(r-1)$

If $r > 1$, then $1 + r + r^2 + \dots + r^k < \frac{r^{k+1}}{r-1} = \frac{a(r^k - 1)}{r-1}$

If $r < 1$, then $1 + r + r^2 + \dots + r^k < 1/(1-r)$

MASTER THEOREM

$T(n) = \begin{cases} aT(\frac{n}{b}) + f(n) & n>1 \\ c & n<d \end{cases}$

① If $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. ($T(n)$ is dominated by the last level)

② If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$. (T.C same for all levels)

③ If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ + $af(\frac{n}{b}) \leq \delta$ for $\varepsilon > 0$ + $\delta > 0$, then $T(n) = \Theta(f(n))$. ($T(n)$ is dominated by last level)

Step 1: compare $f(n)$ to $n^{\log_b a}$

If $n^{\log_b a} > f(n)$, ① If $n^{\log_b a} < f(n)$, ③

If $n^{\log_b a} = f(n)$, ②

Selection Given unsorted array A holding n numbers + an integer K, find K^{th} smallest in A.

First Attempt:

- Divide. Find median, and split on ≤ and > than the median $\Rightarrow O(n)$
- Recur: If $k \le n/2$, find k^{th} element in L. If $k > \frac{n}{2}$, find $(k-\frac{n}{2})^{th}$ element in R
- Conquer. return value $\Rightarrow O(1)$ ↳ $T(\frac{n}{2})$

$T(n) = \begin{cases} T(\frac{n}{2}) + O(n) & n>1 \\ O(1) & n=1 \end{cases} \Rightarrow O(n)$

Attempt 2: Approx. the median

$|A|/3 \le \text{rank}(A, x) \le 2|A|/3$

$T(n) = \begin{cases} T(\frac{2n}{3}) + O(n) & n>1 \\ O(1) & n=1 \end{cases} \Rightarrow O(n)$

Median of 3 Medians

↳ Partition A into 3. For each group find the median. Let x be median of the medians.

$|A|/3 < \text{rank}(A, x) < 2|A|/3$

$T(n) = T(2n/3) + T(n/3) + O(n) \Rightarrow O(n\log n)$

↳ recursive call on $n/3$ elements. Get rid of $2n/3$ elements in each call.

Median of 5 medians

$3|A|/10 < \text{rank}(A, x) < 7|A|/10$

$T(n) = T(7n/10) + T(n/5) + O(n) \Rightarrow O(n)$

↳ recursive call on $n/5$ elements.

Quick Selection

- choose random element as a pivot and partition into 3: (i) < (ii) = (iii) > $O(n)$
- Recur select right element from list $T(n)$
- Return soln. $O(n)$

$E[T(n)] = \begin{cases} E[T(n')] + O(n) \\ O(1) \end{cases} \Rightarrow O(n)$

Randomisation

Generating Random Permutations

Input: integer n; Output: perm of $\{1, 2, \dots, n\}$ chosen uniformly at random (UAR)

- If every execution is equally likely, then we want any 2 permutations to be generated by the same # of execut.

Fisher-Yates Shuffle

$j \in \{i, \dots, n-1\}$

- swap A[i] with A[j] where ↑

↳ element will either stay in place or shuffle w/ something after

# of executions = $1 \times 2 \times \dots \times n = n!$.

# of permutations = $1 \times 2 \times \dots \times n = n!$.

↳ Every execution happens w/ probability $\frac{1}{n!}$ where each execution leads to a diff outcome.

Finding Prime numbers

Distribution of Primes

Let $\pi(n)$ be the # of primes $\le n$, then $\pi(n) = \Theta(\frac{n}{\ln(n)})$. Probability of n to be prime is $\frac{n}{\ln N}$ where $n \in \{1, \dots, N\}$

2 functions: find-prime() + main is-prime()

- helper is-prime runs $T(n)$ so find-prime runs in $O(T(N)\log N)$ ~~a~~ has a bounded error

Rabin-Miller - testing primality

Given n + k. If n is prime, RM(n,k) → T. Else, if n is composite, RM(n,k) returns True w/ prob $1/4k$; False otherwise

```
def witness(x,n):  # check if n is comp.
  write n-1 as 2^k m for m odd
  y <- x^m mod n
  if y mod n = 1, return True # prime
  for i in {1,...,k-1}
    if y mod n = n-1 return True
    y <- y^2 mod n
  return False
```

* If $n > 2$ is comp., there are $\le \frac{n-1}{4}$ values of x s.t. witness(x, n) = True.
* If we call witness(x,n) w/ k diff values of x, probability $\le 1/4k$

main + helper = $O(k\log n)$

Treap

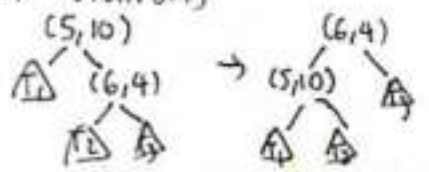
Given $\{(v_i, p_i)\}_{i=1}^{n}$, BST w.r.t v_i and heap property w.r.t p_i

↳ If p_i is chosen UAR from [0,1] then for a treap on $\{(v_i, p_i)\}_{i=1}^{n}$,

$E[\text{Treap height}] = \Theta(\log n)$

Treap Insert: $O(\log n)$

- Do regular BST insertions and restore heap property by doing local rotations



Treap Height: Suppose we sorted values so that $v_1 \le v_2 \le \dots \le v_n$

Pr[v_i is the root] = $1/n$ Pr[root $\in \{v_{\frac{n}{4}}, \dots, v_{\frac{3n}{4}}\}$] = $\frac{1}{2}$

(v_i, p_i)

$(v_1, \dots, v_{i-1})$ $(v_{i+1}, \dots, v_n)$ ⇒ Most nodes are fairly balanced ⇒ $O(\log n)$ height

FROM TUTORIALS

Finding lowerbound: use $\frac{n}{2}$ → split into odd+even ∧ cases

Reverse traversal
- add pointer to the end $O(n^2)$
- add pointer every $\sqrt{n}$ $O(n)$

Traversals:

Pre-order: 12, 14, 18, 23, 16, 11, 13, 34, 31

Post-order: 18, 16, 11, 23, 14, 34, 31, 13, 12

In-order: 18, 14, 16, 23, 11, 12, 34, 13, 31

INDUCTION: (divide-and-conquer)

BinSearch(A, v):
- If |A| = 0 return False
- mid = $\left\lfloor \frac{|A|}{2} \right\rfloor$
- If A[mid] < v: BinSearch(A[mid:], v)
- If A[mid] > v: BinSearch(A[:mid], v)
- If A[mid] = v, return True

Proof:

Base case: |A| = 0 which returns False
∴ Algo is correct

IH: BinSearch will return the correct result for arrays of size |A| < k
use IH to prove BinSearch for arrays of size |A| = k.

Case 1: A[mid] = v returns True ∴ correct

Case 2: A[mid] < v
we'll do BinSearch on array of size $\left\lfloor \frac{|A|}{2} \right\rfloor$
By IH, this must return correct result for that sub-array. If present in sub-array, it must be present in array. ∴ correct
otherwise, if not present in sub-array then it must not be present in the array since the given value must be in the right half of the array as it's sorted and A[mid] < v.
∴ correct

EXCHANGE ARGUMENT:

a b input

a^* b^* be optimal soln

a' b' in sorted order ⇐⇒ returned by algo

assume $a^* \neq a'$, $b^* = b'$ ∴

there must exist i such that $a_i > a_{i+1}$

$$|a^*_i - b^*_i| + |a^*_{i+1} - b^*_{i+1}|$$

$b^*_i \leq b^*_{i+1}$ ∴

$$\underbrace{|a^*_i - b^*_i| + |a^*_{i+1} - b^*_{i+1}|}_{\text{nonswapped}} \geq \underbrace{|a^*_{i+1} - b^*_i| + |a^*_i - b^*_{i+1}|}_{\text{swapped}}$$

After swapping, a^* is now similar to a' without reducing optimality.
∴ By exchange argument, algo is correct

Divide step
- We don't need to make a new array within each recurrence. You can simply make note of the start and end index of the smaller array. This makes it run in $O(1)$.

Pigeonhole principle
- If items are put into containers, then atleast one container contain more than 1 item. (majority)

Graphs:
- Bipartite: vertex set can be partitioned into 2 sets A + B s.t. E ⊆ A × B
- intra-layer: edge w/in layer
- inter-layer: edge bet. layer