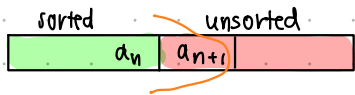


Sorting
 ① element preserving
 ② ordered

$A = [a_0, a_1, \dots]$
 $B = []$ add $\min(A_0), \min(A_1), \dots$

→ Insertion sort



for i in range $(1, n)$: # add a_{n+1} into sorted
 for j in range $(i, 0, -1)$: # insert into sorted
 if $A[j] < A[j+1]$:
 swap $(A, j, j+1)$

	Best	Average	Worst
Insertion	n	n^2	n^2
Merge	$n \log n$	$n \log n$	$n \log n$
Quick	$n \log n$	$n \log n$	n^2
Heap	$n \log n$	$n \log n$	$n \log n$

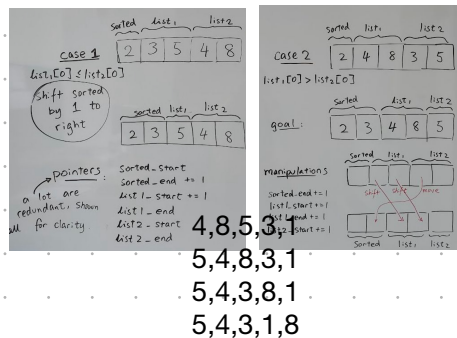
else:
 break # < then stop

→ Merge sort - $O(1)$ extra space complexity

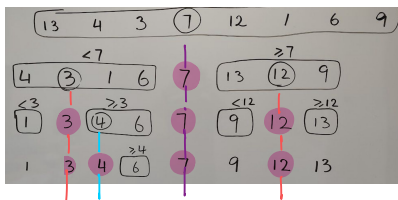
```
def merge(lst1, lst2):
    if lst1 == [] or lst2 == []:
        return lst1 + lst2
    sorted = []
    while lst1 != [] and lst2 != []:
        head1 = lst1[0]
        head2 = lst2[0]
        if head1 <= head2:
            sorted.append(head1)
            lst1 = lst1[1:]
        else:
            sorted.append(head2)
            lst2 = lst2[1:]
    return sorted + lst1 + lst2
```

```
def mergesort(lst):
    if len(lst) == 1:
        return [lst[0]]
    left = mergesort[:len(lst)//2]
    right = mergesort[len(lst)//2:]
    return merge(left, right) # n-1 worst case
    e.g. [1, 3, 5], [2, 4, 6]
```

* inplace merge sort * partition 1 list only!!



→ Quick sort: 'pivot'



worst case $O(n^2)$ pivot worst
 #com: $\frac{n(n-1)}{2}$ $(n-1) + (n-2) + \dots + 1$
 best case $O(n \log n)$ partition
 #com $T(n) = 2T(n/2) + Cn$ recursive
 $= Cn \log n$
 average case $O(n \log n)$
 #com $A(n) = n-1 + \left(\frac{1}{n}\right) \sum_{i=0}^{n-1} [A(i) + A(n-i-1)]$
 $= \theta(n \log n)$

def quicksort(A):

if $\text{len}(A) == 1$:
 return A

pivot = $A[\text{len}(A)//2]$

less = [i for i in A if i < pivot]
 mid = [m for m in A if m == pivot]
 more = [n for n in A if n > pivot]

return quicksort(less) + mid + quicksort(more)

* in place quicksort *** examinable

def qs_inplace(A, int n, int m):

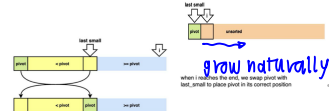
if $n \geq m$:
 return

pivot_pos = partition(A, n, m)

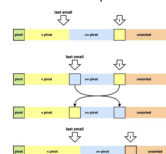
qs_inplace(A, n, pivot_pos-1)

qs_inplace(A, pivot_pos+1, m)

partition function: place the pivot



partition function: compare unsorted stuff with pivot



partition - place the pivot

def partition(A, n, m):

mid = $(n+m)//2$ } choose pivot
 pivot = $A[\text{mid}]$

swap(A, n, mid) # swap with

last_small = n first pivot itself
 for i in range(n+1, m+1):

less_than_pivot = $A[i] < \text{pivot}$

if (less_than_pivot):

last_small += 1

swap(A, last_small, i)

swap(A, n, last_small)

return last_small, pivot

n-1 iteration

bestcase all sorted $O(n)$ n
 worstcase reversed $O(n^2)$ $\frac{(n-1)n}{2}$
 average case $O(n^2)$

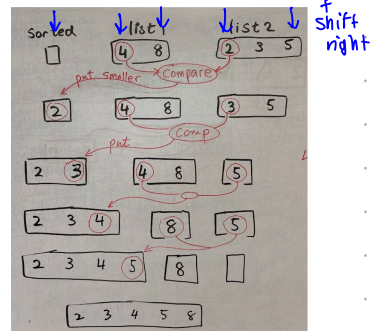
$$\left(\frac{1}{1}\right)1 + \left(\frac{1}{2}\right)2 + \dots + \left(\frac{1}{n}\right)n = \left(\frac{1}{n}\right)\frac{n(n+1)}{2} = \frac{n+1}{2}$$

note: $i = 1, \dots, n-1$ $\frac{2}{4}(n-1) + \frac{1}{2}\frac{(n-1)(n)}{2} = \frac{(n-1)(n+1)}{4}$

* worstcase $O(n \log n)$ Best $2W(\frac{n}{2}) + \frac{n}{2}$
 $W(n) = 2W(\frac{n}{2}) + \frac{n}{2}$

$$\begin{aligned} W(1) &= 0 \\ W(n) &= W(n/2) + W(n/2) + n-1 \quad \text{Or} \\ W(2^k) &= 2W(2^{k-1}) + 2^k - 1 \\ &= 2(2W(2^{k-2}) + 2^{k-1} - 1) + 2^k - 1 \\ &= 2^2W(2^{k-2}) + 2^k - 2 + 2^k - 1 \\ &= 2^2W(2^{k-2}) + 2^k - 2 + 2^k - 1 \\ &= 2^k W(2^{k-k}) + k \cdot 2^k - (1 + 2 + \dots + 2^{k-1}) \\ &= k \cdot 2^k - (2^k - 1) \\ &= n \lg n - (n - 1) \\ &= O(n \lg n) \end{aligned}$$

pointers
 + shift
 right



[2, 3, 1, 4]

→ Heap sort

① remove max + fix heap

def fix_heap(self, i): starting to fix from index i

if self.is-leaf(i): } leaf then do nothing
return

left = self.get_left_tree(i)

right = self.get_right_tree(i)

if not self.is-node(right): is right is in the heap } if the right child doesn't exist
bigger-child = left

else:

self.A = A

bigger-child = left if self.A[left] > self.A[right] else right pick larger val

if self.A[bigger-child] < self.A[i]: } Already bigger than both 2 children

return

swap(self.A[i, bigger-child])

self.fix_heap(bigger-child)

def heapify(self, i):

if not self.is-node(i): } in the heap
return

self.heapify(self.get_left_tree(i))

self.heapify(self.get_right_tree(i))

self.fix_heap(i) → root node

priority queue

pg.put((2, content))

weight (↓ = priority)

priority, task = pg.get()

↓ is priority
outfirst

no need to know much

graph traversal * assume all nodes are reachable in graph traversal

① depth first

visited = []

define DFS(node):

visited.append(node)

for x in neighbors(node):
if x not in visited:
DFS(x)

A B D E F C

A

B

D

E

F

C

② breadth first

visited = []

define BFS(start_node):

q = Queue()

q.put(start_node)

while not q.empty():

node = q.get()

visited.append(node)

for node_neighbor in neighbors(node):

if node_neighbor not in visited:

q.put(node_neighbor)

A → 0

B → 1

D → 2

E → 3

F → 3

A B C D E F

del-max = root-on + 7A leaf + fixheap

Worst/avg/best: $O(n \log n)$

import heapq
make a heap sort algorithm using the heap library
def heapsort(list):
create a heap from the list
heapq.heapify(list) $O(n)$ time
sorted_list = []
while list: h
pop the smallest element from the heap
smallest = heapq.heappop(list) $\log(n)$ time
sorted_list.append(smallest)
return sorted_list

delete-max

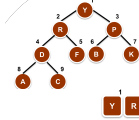
heapq.heappop()

① pop root on + return

② move last to root

③ fixheap

implement abstraction
Heap: an array viewed as binary tree



for any node with children
node at index i
left child at index 2i
right child at index 2i+1

starts index = 1

binary tree T with height h: fills by layer left → right

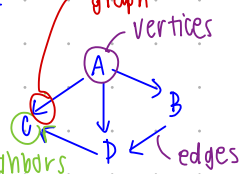
- left subtree of i: 2i
- right subtree of i: 2i+1
- parent node i: $\lfloor \frac{i}{2} \rfloor$ or $i // 2$
- node i is a leaf? $2i > n$?

max heap: parent > children $A[i] \geq A[2i]$
 $A[i] \geq A[2i+1]$

Graph + Graph traversal

graph

directed graph



E * smaller for edge V
Adjacency list E entries
{ 'A': ['B', 'D', 'C'],
'B': ['D', 2], ...
'C': []
'D': ['C'] }

vertices

edges

Adjacency matrix

to

A B C D

A 0 1 1 1

B 0 0 0 1

C 0 0 0 0

D 0 0 1 0

from

* undirected = 2 ways
⇒ symmetric adjacency mat

weighted edges
A B C
A 0 1 3
from B ∞ 0 ∞
C ∞ 2 0

1

BFS

Dijkstra's algorithm - returns (d, pi)

• set d = ∞; unvisited
• visited node: S (T/F)

* visit the new node with smallest d + update the distance

def dijkstra(V, E, start): adjacency list w/ weights

initialize

d = { node: float('inf') for node in V }

pi = { node: None for node in V }

S = set()

d[start] = 0 pq = [(0, start)]

u = heapq.heappop(pq)

while len(S) < len(V):

unvisited = V - S # unvisited nodes

u = min(unvisited, key=lambda node: d[node])

S.add(u) → closest unvisited node

for (v, w) in E[u]:

if d[u] + w < d[v]: → distance from start = update

d[v] = d[u] + w

pi[v] = u

heapq.push(pq, (d[v], v))

return d, pi

visit new node

update distance

post-traversal

→ Topological sorting • sorting with dependencies, linearization all arrow → (pre-requisite) + meet all prereq.

$O(|V| + |E|)$

(each $v \in V$ is visited once)

- not unique (many solutions)
- only DAG (Directed Acyclic Graph) → no cycle
 - ↳ rooted trees

- steps ① pick 1 unvisited node $[n]$
- ② DFS for reachable unvisited nodes visited
- ③ recursive call back $visited = [n] + visited$

subgraph

• $V' \subseteq V, E' \subseteq E$

spanning tree

- 'tree' subgraph (no cycle!)
- connect all nodes in the graph



minimum spanning tree (MST)

- spanning tree w/ smallest total weight

★ Greedy Algorithm

- best atm to find MST
- ↳ move towards the closest
- optimal for MST

visited = []

def DFS(node):

for x in neighbors(node):
if x not in visited:
DFS(x)

visited = [node] + visited

def topSort(graph):

call unvisited node

★ MST property

check 1 possible ok alt → 'local'

• T is a spanning tree of G



adding edges (u,v) will create cycles
& $(u,v) = \text{max-weight edge of the cycle}$
add (u,v)

if u,v already: partition = confirm no cycle

$|V| = |E| + 1$ MST = $[(A,B,1), (B,C,2), \dots]$

while (edge heap not empty) $O(E)$

& $(len(MST) < |V| - 1)$:

pop the min $W, u, v = \text{heappop}, \text{heappop}(\text{edge_heap})$

if not uf. connected (u,v) : $O(\log V)$

uf. union (u,v)

MST.append $((u,v,W))$ $O(\log V)$

if u,v already: partition = confirm no cycle

if not connected: $< |V| - 1$: disconnected

if connected: $> |V| - 1$: cycle

important update since last class

stop if MST already has $|V| - 1$ edges (this tree is already formed)

if u,v already: partition = confirm no cycle

if not connected: $< |V| - 1$: disconnected

if connected: $> |V| - 1$: cycle

important update since last class

stop if MST already has $|V| - 1$ edges (this tree is already formed)

if u,v already: partition = confirm no cycle

if not connected: $< |V| - 1$: disconnected

if connected: $> |V| - 1$: cycle

important update since last class

stop if MST already has $|V| - 1$ edges (this tree is already formed)

if u,v already: partition = confirm no cycle

if not connected: $< |V| - 1$: disconnected

if connected: $> |V| - 1$: cycle

important update since last class

stop if MST already has $|V| - 1$ edges (this tree is already formed)

if u,v already: partition = confirm no cycle

if not connected: $< |V| - 1$: disconnected

if connected: $> |V| - 1$: cycle

important update since last class

stop if MST already has $|V| - 1$ edges (this tree is already formed)

if u,v already: partition = confirm no cycle

if not connected: $< |V| - 1$: disconnected

if connected: $> |V| - 1$: cycle

important update since last class

stop if MST already has $|V| - 1$ edges (this tree is already formed)

if u,v already: partition = confirm no cycle

Kruskal's Algorithm

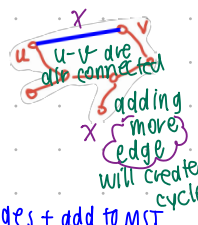
↓ shortest edge

↓ 1 edge at a time

enumerate all edges + add to MST priority queue

do not make cycle

'connectivity': add the edge u,v when they are not already connected in MST



★ Union find algorithm for connectivity

(i) record a set of connected components

(ii) update the set when we add edges

→ 3 functions

① find(x); find the representative element x belongs to

② connected(x,y); x & y connected? find(x) == find(y)?

③ union(x,y); union the sets x,y belongs to

find(4) → 1

Quick Union: representative element = root node

• array 'id' → every node points to its parent, root points to itself

union find operations

def parent(node):

return id[node]

def root(node):

return parent(node) == node is the node root?

def find(node):

return the root node if root(node): (representative)

return node

return find(parent(node)) → recursively call parent()

def connected(x,y):

return find(x) == find(y) x,y same partition? aka connected?

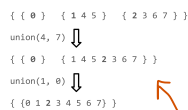
def union(x,y):

repr1 = find(x)

repr2 = find(y)

id[repr2] = repr1

root node y via root node x



union(4, 7)

{ {0} {1 5} {2 3 6 7} }

union(1, 0)

{ {0 1 2 3 4 5 6 7} }

union(1, 0)

{ {0 1 2 3 4 5 6 7} }

union(1, 0)

{ {0 1 2 3 4 5 6 7} }

union(1, 0)

{ {0 1 2 3 4 5 6 7} }

union(1, 0)

{ {0 1 2 3 4 5 6 7} }

union(1, 0)

{ {0 1 2 3 4 5 6 7} }

union(1, 0)

{ {0 1 2 3 4 5 6 7} }

union(1, 0)

{ {0 1 2 3 4 5 6 7} }

union(1, 0)

{ {0 1 2 3 4 5 6 7} }

union(1, 0)

{ {0 1 2 3 4 5 6 7} }

union(1, 0)

{ {0 1 2 3 4 5 6 7} }

equivalence relation: reflexive, symmetric, transitive

↳ dynamic: relationship changes overtime

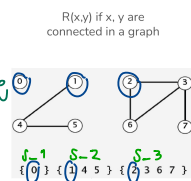
partition / disjoint union

subset of a relation (R)

• $\forall (x,y) \in S-i, R(x,y) \text{ true}$

• $S-1 \cup \dots \cup S-n = R$

• $S-i \cap S-j = \emptyset$



• can have representative elements

↳ 'weighted' Quick Union: small tree via big tree union

def size(root_node):

return sz[root_node]

def union(x,y):

repr1, repr2 = find(x), find(y)

if size(repr2) <= size(repr1):

id[repr2] = repr1

sz[repr1] = sz[repr1] + sz[repr2]

else:

id[repr1] = repr2

sz[repr2] = sz[repr1] + sz[repr2]

update the size

find & union

find & union

find & union

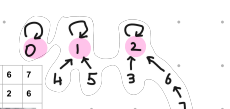
find & union

find & union

find & union

find & union

find & union



sz[0] = 1, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 2, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 3, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 4, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 5, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 6, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 7, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 8, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 9, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 10, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 11, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 12, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 13, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 14, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 15, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 16, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 17, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 18, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 19, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 20, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 21, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 22, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 23, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 24, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 25, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 26, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 27, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 28, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 29, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 30, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 31, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 32, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 33, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 34, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 35, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 36, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 37, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 38, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 39, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 40, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 41, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 42, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 43, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 44, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 45, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 46, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 47, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 48, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 49, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 50, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 51, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[0] = 52, sz[1] = 1, sz[2] = 1, sz[3] = 1, sz[4] = 1, sz[5] = 1, sz[6] = 1, sz[7] = 1

sz[

W8 - Analysis Technique

Recursive functions

Recurrent relations

① substitution method

(i) guess the correct ans $\Rightarrow$ want lowest upper bound

- recursion tree

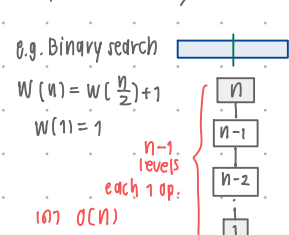
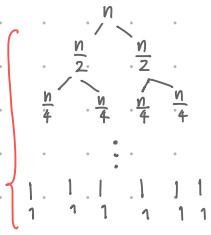
e.g. Quicksort (bestcase)

$$W(n) = 2W(\frac{n}{2}) + n - 1$$

$$W(1) = 0$$

$$\text{for } O(n \log n)$$

$$(2n-1 \text{ comp}) \leq n \text{ ops.}$$



e.g. Binary search
 $W(n) = W(\frac{n}{2}) + 1$
 $W(1) = 1$
 $\text{for } O(n)$

e.g. Quicksort (worstcase)
 $W(n) = W(n-1) + (n-1)$
 $W(1) = 0$
 $\text{for } O(n^2)$

NOTE: smooth function: non-negative $f(n) \geq 0$
 $f(2n) = \Theta(f(n))$
 equal complexity level!
 e.g. $f(n) = n^2 = \Theta(n^2)$, $f(2n) = 4n^2 = \Theta(n^2)$
 $f(n) = 2^n = \Theta(2^n)$, $f(2n) = 2^{2n} \neq \Theta(2^n)$
 $f(n) = n \log n$, $f(2n) = 2n \log 2n$
 $= n \log 2n + n \log 2n$
 $= 2n \log 2 + 2n \log n$
 $= 2n + 2n \log n = \Theta(n \log n)$
 if $f(n)$ is smooth,
 $f(n) = \Theta(g(n)) = O(g(n)) \forall n$
 and $\lceil \cdot \rceil, \lfloor \cdot \rfloor$ don't affect growth

(ii) mathematical induction proof

base case: $P(1)$ true
 inductive step: $P(k) \rightarrow P(k+1)$

e.g. Quicksort (worstcase) guess: $O(n^2)$
 ① base case: $W(1) = 0 \leq n^2 = 1(1^2) = C(n^2)$; try pick $C=1$ first

② inductive step: assume $W(n) \leq C(n^2)$ for $n=1, 2, \dots, k$

consider $W(k+1)$,

try $C=1$; assume $W(n) \leq n^2$ for $n=1, 2, \dots, k$

$$\begin{aligned} \therefore \text{from ① \& ②,} \\ W(n) &\leq n^2 \text{ for } n \geq 1 \\ \text{So } W(n) &= O(n^2) \end{aligned}$$

$$\begin{aligned} W(k+1) &= W(k) + k \text{ want } \leq (k+1)^2 \\ &\leq k^2 + k \\ &\leq k^2 + k + (k+1) \quad k \geq 1 \\ &\leq (k+1)^2 \end{aligned}$$

e.g. Quicksort bestcase guess: $O(n \log n)$

① base case $W(2) = 1 \leq 2 = 2 \log 2$

② inductive $W(2^k) \leq 2^k \log 2^k$; $n = 2^k$

$$\begin{aligned} \text{consider } W(2^{k+1}) &= 2W(2^k) + 2^{k+1} - 1 \\ &\leq 2(2^k \log 2^k) + 2^{k+1} - 1 \\ &= 2^{k+1} \log(2^k) + 2^{k+1} - 1 \\ &= 2^{k+1}(k) + 2^{k+1} - 1 \\ &= 2^{k+1}(k+1) - 1 \leq 2^{k+1}(k+1) = 2^{k+1}(\log 2^{k+1}) \end{aligned}$$

we proved $W(n) = O(n \log n)$; $n = 2^k$ $\therefore$ same apply for all n !
 $n \log n$ = smooth function

③ iteration method: extend to initial conditions

④ master method: find asymptotic tight bound

$$W(n) = aW(\frac{n}{b}) + f(n) \Rightarrow \begin{cases} a \geq 1 \\ b > 1 \end{cases} \Rightarrow \begin{cases} \text{tree height} \\ \log_b n \end{cases} \Rightarrow n^{\log_b a} \text{ vs } f(n)$$

Master method (upper bound)
 proof: $f(n) = O(n^{\log_b a - \epsilon})$ $f(n)$ is small
 $\Theta(n^{\log_b a})$
 Master method (lower bound)
 proof: $f(n) = \Omega(n^{\log_b a + \epsilon})$ $f(n)$ is large
 $\Theta(f(n))$
 n is big and positive
 $n \rightarrow \infty$

Wd - Greedy Algorithm: optimization problems considering knowledge atm.

✓ individual choice not ex (small info)

✓ can't go back $\therefore$ not guarantee optimal sol

coin changing: n -max x optimal $D = \{1, 10, 25\}$, $n=30$ $O(n)$
 $\lfloor \cdot \rfloor$ x optimal $O(1)$

interval scheduling problem

FCFS x optimal

SJF x optimal

Earliest-finish-time-first (EFTF) $O(n \log n)$ $\checkmark$ optimal!!

knapsack problem

highest profit/weight first $O(n \log n)$ $\checkmark$ optimal

Dynamic programming

- recursive calls with

memorize solution + recompute

- solve optimality problems that satisfy the principle of optimality

$\Rightarrow$ dictionary store + retrieve + member

normal: $2^{\frac{n}{2}+1} - 1 < \# \text{ recursive calls} < 2^n - 1$
 $n = \# \text{ nodes}$

dynamic programming

member(Dictionary, j);
 return Dictionary[j] $\leftrightarrow$ 1
 store(Dictionary, j, s);
 Dictionary[j] = s
 retrieve(Dictionary, j);
 return Dictionary[j]
 All these operations are $O(1)$ time complexity

each $n-1, n-2, \dots, 1$
 compute 1 time each

$$O(n)$$

Topdown

(vs)

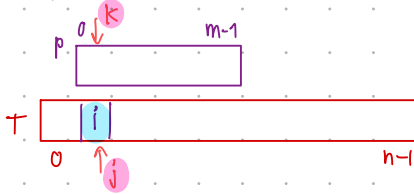
Bottom up

recursion tree
 dictionary to store & solve subproblems
 before solving sub problems, check
 $\rightarrow$ have: retrieve sol
 $\rightarrow$ no hv: recursive call + store in dict
 store(), member(), retrieve() $\rightarrow O(1)$
 $O(n)$

subproblem graphs
 draw subproblem graphs + find dependencies
 dictionary to store solutions for subproblems
 small $0, 1, 2, \dots, 6$ big
 Sol $\rightarrow$ need $\rightarrow$ need $\rightarrow$ need
 $O(n)$

String matching

1) Straight-forward solution

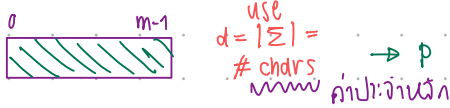


```
while (i < n) {
    if (T[i] == P[0]) {
        j++; k++;
        if (k == m) return i;
    }
    else {
        k = 0;
        j = ++i;
        if (j > n - m) break;
    }
}
```

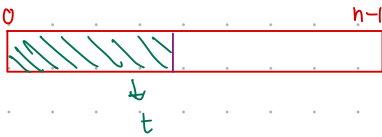
Worst case: $O(mn)$

2) Rabin-Karp Algorithm

1) Convert pattern length m to number p



2) the first m chars of the text to t



3) $P == t$?
Yes → return i

No → $t = (t - T[i] * d^M) * d + T[i+m]$

Hash

```
hash (T, m, d) {
    int h = T[0] % q;
    for (int i = 1; i < m; i++)
        h = (h * d + T[i]) % q;
}
```

$\Theta(m)$

3) $hp == ht$ & equal-string (P, T, 0, i, m)

No → shift
Yes → return j

```
int RKScan(P, T)
{
    m = Length(P);
    n = Length(T);
    dM = 1;
    For j = 1 to m-1 dM = dM * d % q; // d = |Σ|
    hp = hash(P, m, d); // O(m)
    ht = hash(T, m, d);
    for (j = 0; j <= n - m; j++) {
        if (hp == ht && equal_string(P, T, 0, j, m))
            return j;
        if (j < n - m) ht = rehash(T, j, m, ht);
    }
    return -1; // pattern not found
}
```

```
ht = rehash(T, j, m, ht);
int rehash(T, i, m, ht) {
    oldest = (T[i] * d^M) % q;
    oldest_removed = ((ht + q) - oldest) % q;
    return (oldest_removed * d + T[i+m]) % q;
}
```

Worst case $\Theta((n-m+1)m)$

expected $O(n \times m) + O(m) + O(n)$
very small hash for loop

spurious hits kept low by ↑ q

3) Boyer-Moore

- minimize alignments, maximize shifts
- Better with long patterns



```
int BMScan(char P[], int m, int T[], int n, int[] charjump, int[] matchjump)
{
    int i, j, k;
    j = m; k = m;
    while (j <= n) {
        // match found
        if (k < 1) return j - 1;
        if (T[j] == P[k]) { j--; k--; }
        else { j += max(charjump[T[j]], matchjump[k]); k = m; }
    }
    return -1; // match not found
}
```

charjumps

- (I) d in P : w rightmost d charjump
- (II) d not in P : shift m-i

matchjumps : from pattern



mismatch minimizes shift

matchjump = #match + #slide

Computing the jumps for all the characters:

```
void computeJumps(char P[], int m, int alpha, int[] charjump)
{
    for (ch = 0; ch < alpha; ch++)
        charjump[ch] = m;
    for (k = 1; k <= m; k++)
        charjump[P[k-1]] = m - k;
}
```

E.g. for P = "abacab", m = 6, charjump = [5, 4, 5, 1, 5, 1]

