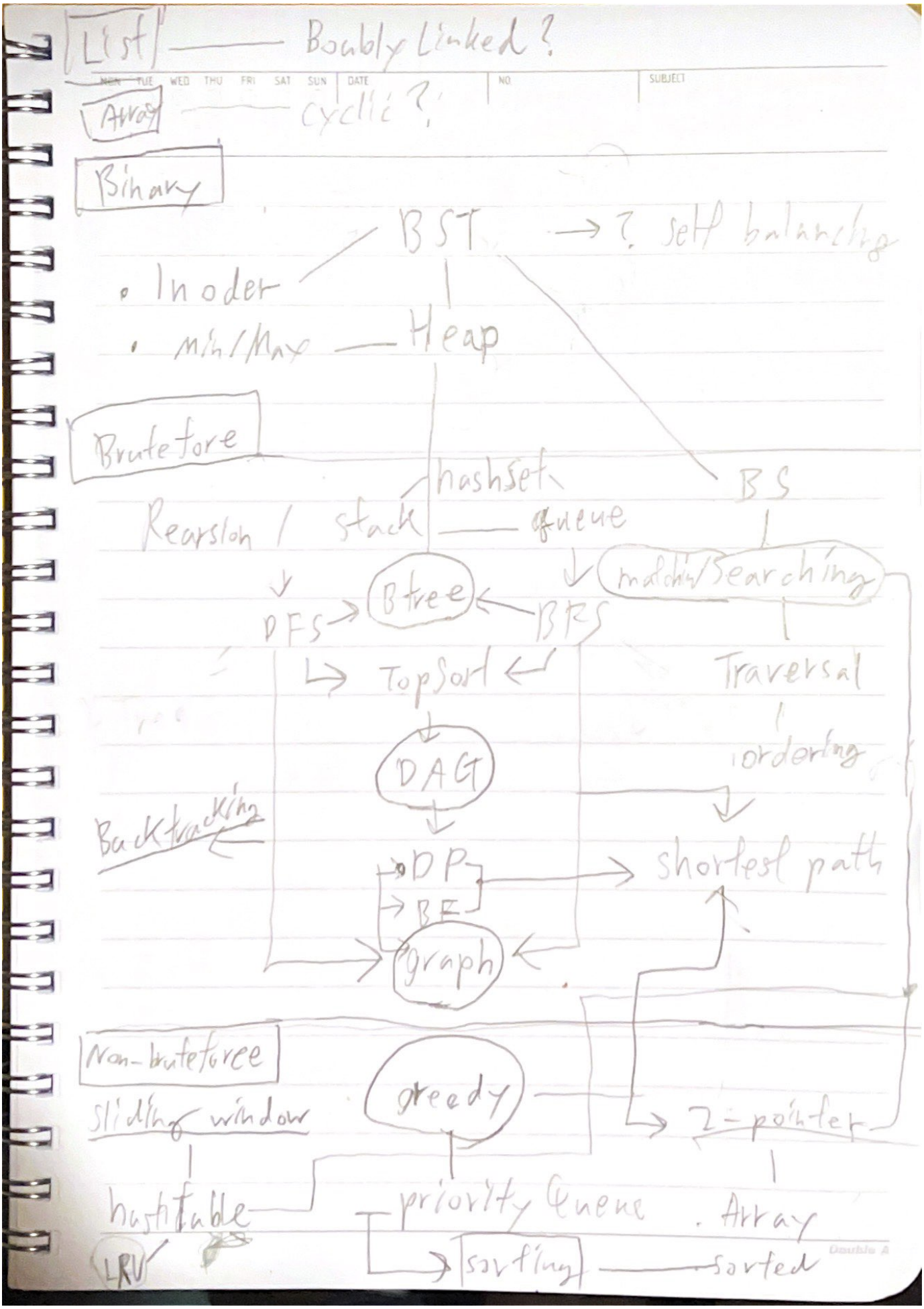

List
Doubly Linked?
Array
cyclic?
Binary
BST
? self balancing
Inoder
min/Max
Heap
Brutefore
hashset
BS
Rearsion / stack
queue
matching/Searching
Btree
DFS
BFS
TopSort
Traversal
ordering
DAG
Backtracking
DP
BF
shortest path
graph
Non-bruteforce
greedy
sliding window
2-pointer
hashtable
priority Queue
Array
LRU
sorting
sorted

Do Nothing

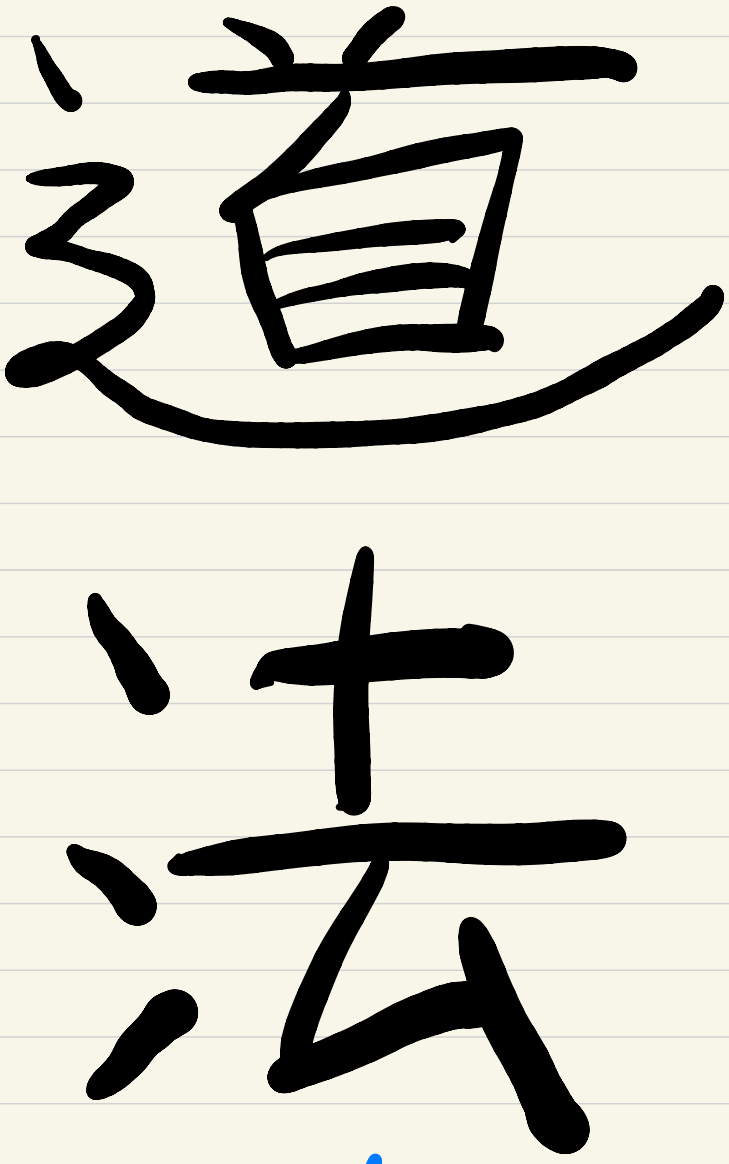
is part of

practice

Mental health over everything else

be water

The law of reversed effort.



The harder you try,
The worse it gets

phasing

phase 1

→ first half of algo sheet

(cool down period)

phase 2 ← by the end of year

→ practice key questions

Next year

phase 3

→ second half

(cool down period)

phase 4

→ practice

Problem characteristics

Outcome nature:

- contiguous
- sub-string / sub-array
- sub-sequence
- A pair
- connected
- edge with weight
- directional / bi-directional

String criteria:

- duplicate
- distinct
- palindrome v.s Parentheses
- anagram
- common subsequence / target str

SubSeq **vs** SubStr
A **subsequence** is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

A **substring** is a contiguous sequence of characters within the string.

你要快樂
就要學會偶爾消失

偶爾你要試試，徹底失聯式的放鬆

不上網、不說話、不見人

找一套書、一套劇或者一套遊戲，讓自己徹底沈溺進去

在那幾天的失聯裡，你會真的發現

日常生活裡絕大部分的焦慮、疑惑、不快樂

都是由人際間的影響造成的

遠離人群，在如今也包括遠離網路

你要快樂，你就要學會偶爾消失

number sequence criteria:

- target sum
- increasing
- capacity constraint
- fractional
- consecutive ($nums[i] = nums[i-1] + 1$)

Counting / combinatorial criteria

1. Fibonacci
2. non-ordering
   - combination (nCr)
     e.g 0/1 Knapsack
   - Unbounded Combination
     $(k_1 x_1 + k_2 x_2 + \cdots k_n \cdot x_n)$
     e.g Unbounded knapsack
3. Ordering
   - permutation (nPr)
   - Unbounded Permutation

Boolean / Return #occurence Goal:
- exists / Matches the pattern (isValid)
- search
- equality (e.g subarray sum = k)
  (i.e prefix sum problem)
- Missing Number in a certain range
  (i.e cyclic sort)

Optimization goal:
- shortest (Min of sum)
- longest (Max of sum)
- largest / smallest number in a seq.
- The right / left next smaller / greater element in an array.

Traversal goal:
- pre-order
- in-order
- post-order
- topological Order
  (reversed post order, dependency list)
- level Order
  (return a list of list by level)

In-place Operation goal:

- sorting
- Merge sorted
- Re-ordering v.s reversed
- Remove N-th element from the end

"Statistics" Goal:

- Top K
- Median, Average

2D Array (Matrix) criteria:

- Out-of-boundary
- diagonal
- vertical / horizontal

1D Array criteria:

- Rotated ([3, 4, 5, 1, 2])
  (i.e Circular Array)

Graph criteria:

- single source
- Multiple source
- destination
- All paths from one src to one dst
- At most K stops (vertices)
- has cycle (Acyclic)
  - positive/negative cycle
- opposite weight exists?

Interval criteria:

- Overlapping interval
- min number of intervals to be removed so that the rest is non-overlapping

(vs Min Meeting Room)

Tree - Criteria

- Balanced
- Binary
- Complete
- depth

Number Operation Goal:

- Reversed integer
- fractional to string

Singly Linked list goal:

- cycle detection
- entry of the cycle

Binary Tree Goal:

- serialize
- de-serialize
- subtree of another tree
- Inverted BinaryT (swap L/R)
- Is A binary search Tree?
  - ↳ K-th smallest

Algorithm

cheat sheet

Graph Search & classification

1. DFS

- traverse all vertices $|V|$

  Time: $O(V+E)$

  Space: $O(V)$

#sample question

1059 All paths from source lead to destination

- traverse all paths between any 2 vertices in a graph

  ⇒ Backtracking

「As revisit is needed in another possible path, mark (visit) & unmark are required」

「BFS won't do mark & unmark」

Sample Question

- 797. All paths from source to target
  - ↳ (what if Non-DAG)*

Time : $O[2^V \cdot (V+E)]$

space : $O(2^V \cdot V)$

2. Edge classification $u \longrightarrow v$

未發現 • White : A tree edge

(before discover v

(被發現) • Gray : A back edge

(After discovered ✓

but before finishes ✓

(結束) • Black :

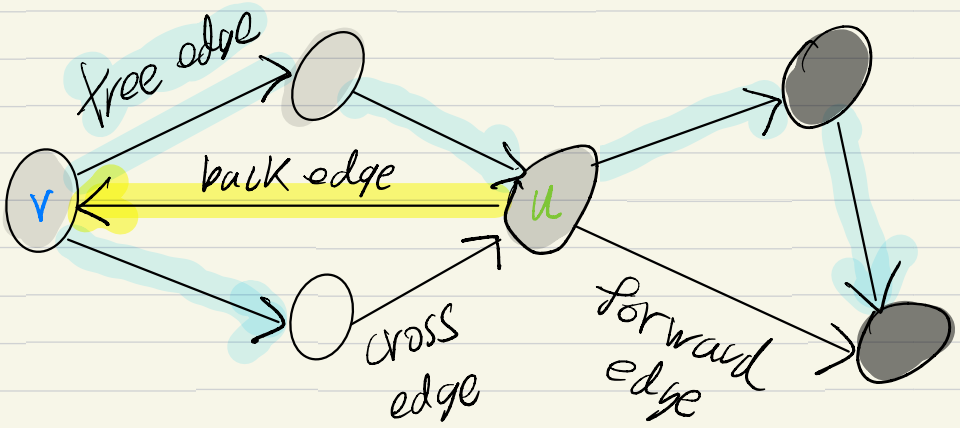
↳ forward edge

if v is finished

& start_t[u] < start_t[v]

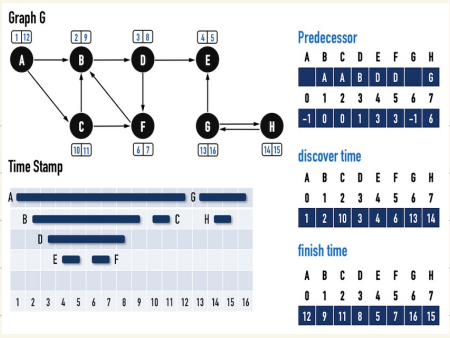
↳ cross edge
if v is finished
& start_t[u] > start_t[v]

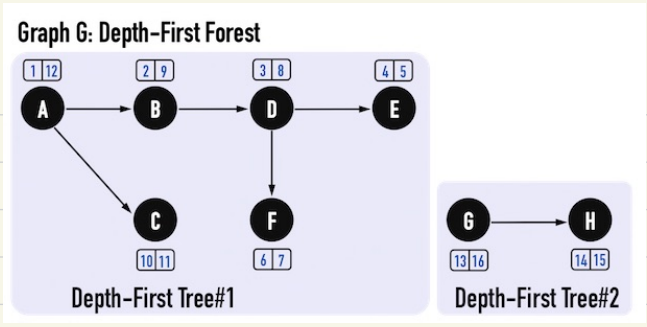
(1) ⇨ Colored vertex for cycle detection
G has a cycle if and only if DFS finds at least one back edge.



(2) ⇨ Depth-First Forest (# no. of tree ≥ 2)

如同BFS()，在Graph上進行DFS()同樣可以得到Predecessor Subgraph，又稱為Depth-First Tree。若Graph本身不是(strongly) connected component，則有可能得到Depth-First Forest，如圖五：





⇒ pseudo code

```
def dfs(G)
    for v : G
        color[v] = 'w'

    for v : G                          (2) ⇒ O(V)
        if Not noCycle:
            break

    if color[v] == 'w'
        noCycle = dfs_visit(v, G, color)

def dfs_visit(v, G, color)
    color[v] = 'G'
    noCycle = true
    for n : neighbours(v)              ⇒ O(E)
        if not noCycle:
            break
```

```
if color[n] = 2 'W':
    noCycle = dfs_visit(n, G, color)
else if color[n] = 2 'G'
    return false
color[v] = 'B'
return noCycle
```

Time : $O(V+E)$

space : $O(V)$

3. BFS

- find the shortest path without traversing all paths

  # DFS must traverse all paths between 2 vertices before finding it.

  # re-visit is not required to solve STP

  # equal and positive weight edges

Sample Question:

126. World ladder 2

↳ return all the shortest transformation seq. from beginWord to endWord.

- traverse all vertices in CT
  # BFS to traverse all path ?

797. All paths from source to target

↳ Undirected graph

Time: $O[2^V \cdot (V+E)]$

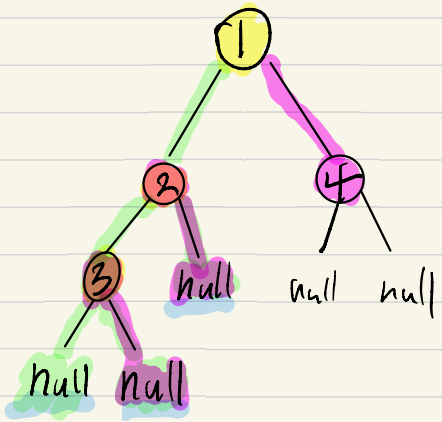
space: $O(2^V \cdot V)$

Traversal Ordering

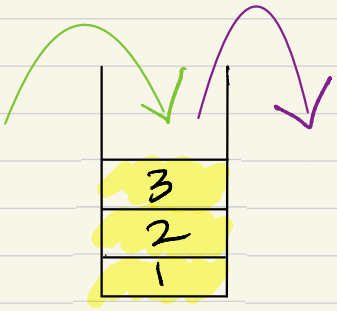
1. Tree DFS

↳ Preorder traversal (classic DFS)

```
void dfs(Node param){
    if(param == null)
        return;

    visit(param);
    dfs(param.left);
    dfs(param.right);
}
```

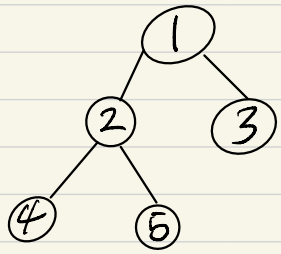



```
Node param = root;
while(param != null
    || !stack.isEmpty()){
    if(param != null){
        visit()
        push()
        param = param.left
    } else {
        pop()
        param = param.right
    }
}
```



stack

Ordering: Top → Bottom
Left → right



1, 2, 4, 5, 3

#Sample Question:

100. Same Tree

124. Binary Tree Maximum Path Sum

98. Validate Binary Search Tree

↳ Inorder traversal (Ascending order in BST)

```
void dfs(Node p){
  if(p==null) return;

  dfs(p.left)
  visit(p);
  dfs(p.right) #Sample Question
}
```

230. Kth Smallest Element in a BST

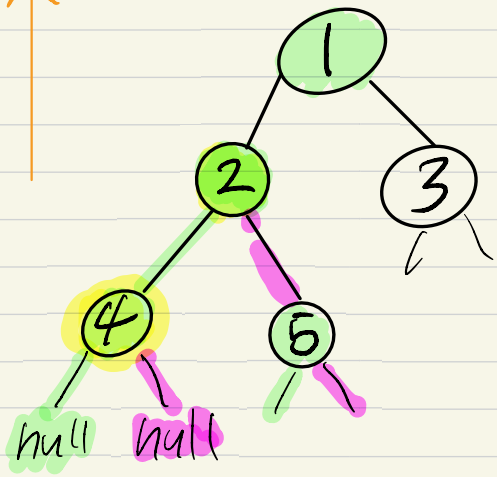
105. Construct Binary Tree from Preorder and Inorder Traversal

Ordering: Left → node → right

⇒ 4, 2, 5, 1, 3

↳ post-order traversal

```
void dfs(Node p){
  if(p==null) return;
  dfs(p.left)
  dfs(p.right)
  visit(p)
}
```

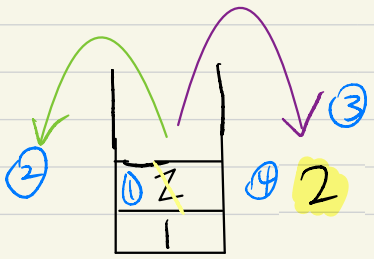


Ordering:

Bottom → Top

Left → right

⇒ 4, 5, 2, 3, 1



*(also add to the first)

what if we do it in reversed order?

```
/**
* The challenge here is
*     1. whenever we backtrack to the parent,
*     we don't know which side we still haven't finished
*
*     2. two Solutions
*         i) we have to mark(save) which side has been visited instead of
only store the node
*         ii) reverse of postorder
*             (Order: left, right, visit(addLast) => visit(addFirst), right, left)
*/
```

↳ then we do know pop() returns to do the left for next step.

```
while ( p != null
  || !stack.isEmpty() ) {

  if ( p != null) {
    visit(list::addFirst)
    push(p)
    p = p.right
  } else {
    p = pop()
    p = p.left
  }
```

2. Tree BFS

↳ level order traversal

```
q.offer(root)
while(!q.isEmpty){
    n = q.size();
    for(i..n){
        e = q.poll()
        // offer....
    }
    depth++;
}
```

Sample Question : 104. Maximum Depth of Binary Tree

3. Graph traversal

↳ Topological sort

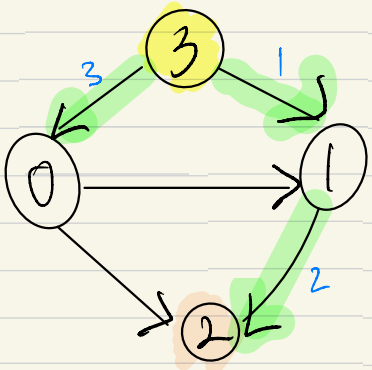
- A reversed post order traversal

1. post order traversal makes sure all the child nodes get processed before the current node
2. Regardless of the start/src node, it always generate the same order

Recursion call seq.

Top order: A, B
post-order: B, A

先 A 後 B



3, 0, 1, 2
2, 1, 0, 3

3. B is dependent on A
   or
   B's prerequisite = A

A directed graph

- what if start node is NOT root (first one in top order)
- what if # no. of tree ≥ 2
- what if G has cycle
- How pre-order traversal will always fail to work, No Matter which node we start from ?

post-order

0 2, 0

2

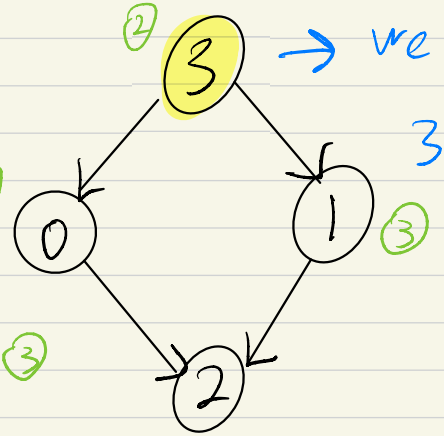
2, 0, 1, 3

3

2, 0, 1,

2

2, 0, 1, 3



→ we got wrong order

3, 0, 2, 1

pseudo code for DAG using DFS

```
def topSort(G)
    stack s
    for v : G                          // O(V)
        if v not visited
            post_dfs(v, G)
    s.popAll()

def post_dfs(v, G, s)
    for n : neighbours(v)              // O(E)
        if n not visited
            post_dfs(n, G, s)
    s.push(v)                          // post order traversal
    visited[v] = true                  // visited
```

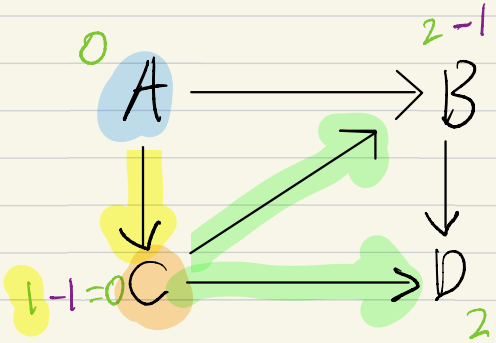
Time: O(V+E) # Sample Question

space: O(V)

210. Course schedule

↳ Non-DAG is possible

Alternative - BFS & Implicit Cycle Detection



Queue C

Result = A, ← The course A got finished, then which vertice's prerequisite contain the course A, their prerequisite will be deducted by 1

And when all prerequisite of a vertex have been done, start to do

For C

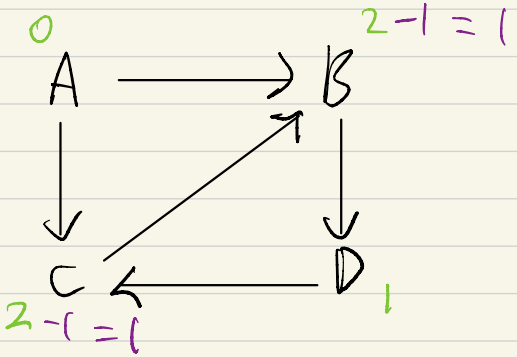
In-Degree = 1

out-Degree = 2

* Remaining in-Degree of the course - C
⇒ Already finished prerequisite of the course
How many are left to be finished before taking course - C ?

1. count inDegrees of all vertices
2. store all vertices with inDegree = 0 into the queue — zeroDegree
3. pop vertex of source / inDegree = 0
   (# no. of source ≥ 0)
   ↳ decrement its neighbours' indegree

4. What if cycles exist ?



⇒ check if all courses can be done.

```
int[] inDegree
Queue<Integer> zeroDegree

①
②

③ while (!zeroDegree.isEmpty()) {          // O(V)
    int v = zeroDegree.poll()
    result[i++] = v
    for (n : neighbours(v)) {              // O(E)
      inDegree[n]--
      if (inDegree[n] == 0){
        zeroDegree.offer(n)
      }
    }
  }
}
```

④ iterate inDegree[] or i == numCourses

Time: O(V+E)

space: O(V)

Brute Force

0. Recursion

- recurrence relation
- base case

⇒ the result of the case ~~with~~ without any further recursion calls

⇒ In-degree = 0

No prerequisite to compute the answer (Independent case)

- Time Complexity

$R \cdot O(s)$

// R = # no. of invocations

// O(s) ⇒ for each recursion call

↳ Excution Tree e.g $O(2^n)$

- Space Complexity
  - Recursion-related (i.e stack)
  - Non-Recursion-related (heap memory)

1. Backtracking

for CSP (Constraint Satisfaction Problem)

1. Base case ⇒ add to result
2. iterate through candidates list (窮舉)

   a) isValid / pruning

   If there is a mismatch we found early on, then decided to go back to the next step

   b) save state

   c) recursive call with new state
   ↳ one step further to incrementally add candidates to the solution

   d) backtrack - remove state

⇒ Combination : Candidates = numbers 0 to n

state = f, c

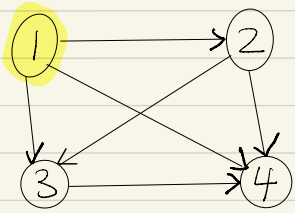
– bound : 77. Combination

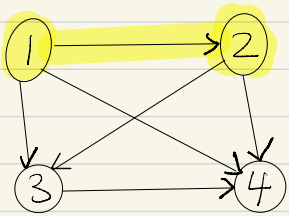
Candidate - iterative backtracking

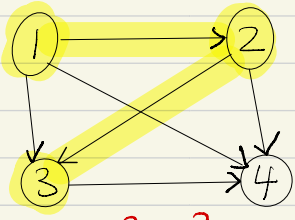
```
def backtrack(n, k, r, f, c, result)

    if(r == k)
      result.add(list(c))

    for i=f....n
      c.add(i+1)          // take
      backtrack(n, k, r-1, i+1, c, result)
      c.removeLast(c)     // skip
```

DFS (take & skip recursive) Backtracking

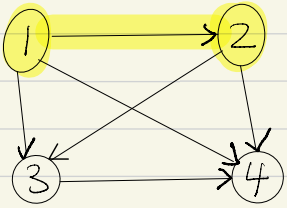
```
def dfs(i, r, n)

    if(r==0) return 1

    if(i==n) return 0
                          take          skip
    return dfs(i+1, r-1, n) + dfs(i+1, r, n)

def nCr(n, r)
    return dfs(0, r, n)
```

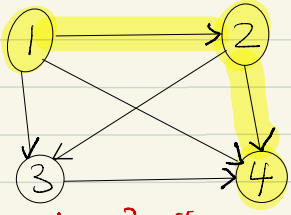




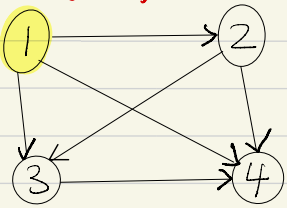


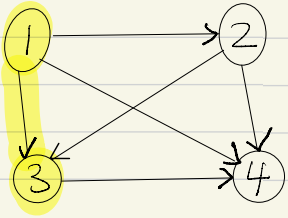
1, 2, 3

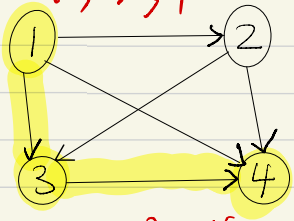




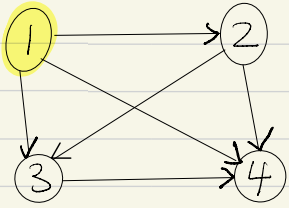
1, 2, 4

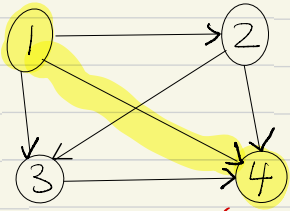




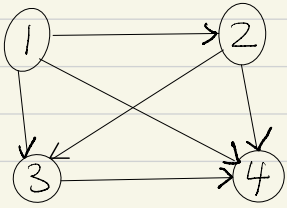


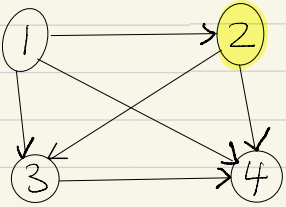
1, 3, 4

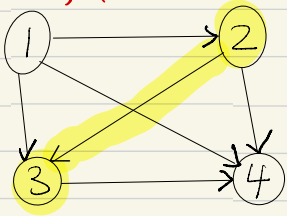


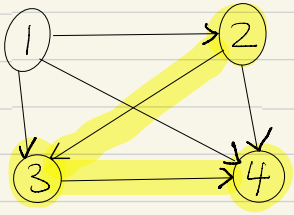


1, 4 → invalid

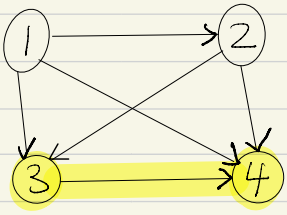








2, 3, 4



3, 4 ⇒ invalid

— Unbound : 39. Combination Sum

Recurrence Relation:

$f(i, s) = f(i, s-i)$ And $f(i+1, s)$

take but no pass skip

Base Case:

$f(i, 0)$ ⇒ If $s == 0$, then add to result

$f(i, s<0)$ ⇒ If $s < 0$, then return

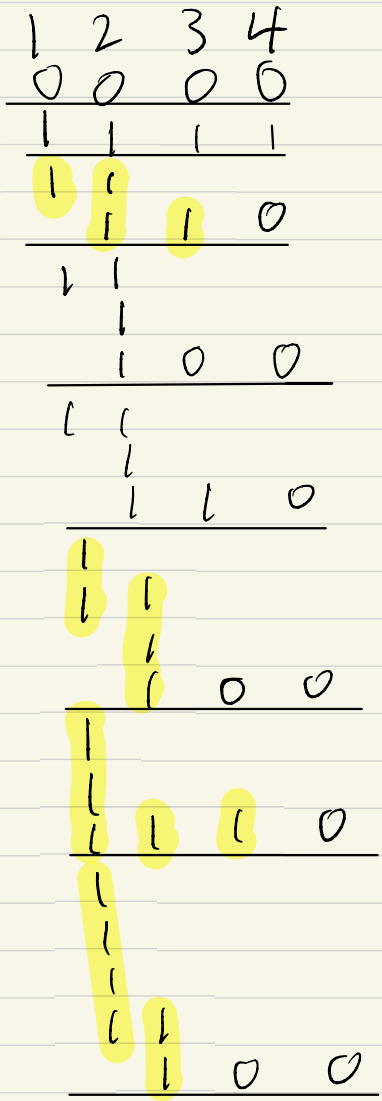
$f(n, s)$ ⇒ If $i == n$, then return

Binary (bounded)

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

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

Binary (Unbounded) target = 8



⇒ Permutation Candidates: numbers 0 to n
state : visit

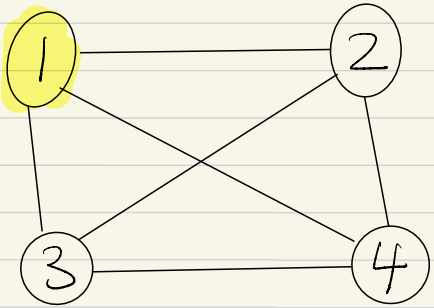
Candidate - iterative backtracking

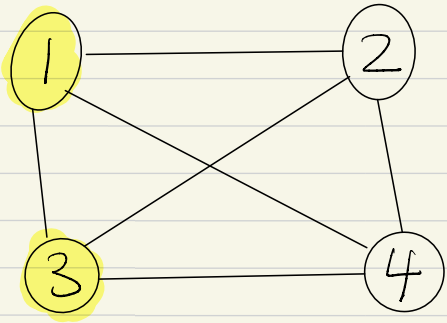
```
def backtrack(r, nums, visit, result)
    if(r == size(nums))
        result.add(list(visit))

    for i = 0...size(nums)
        if(visit.contains(nums[i]))
            continue

        visit.add(num[i])
        backtrack(r+1, nums, visit, result)
        visit.remove(num[i])
```

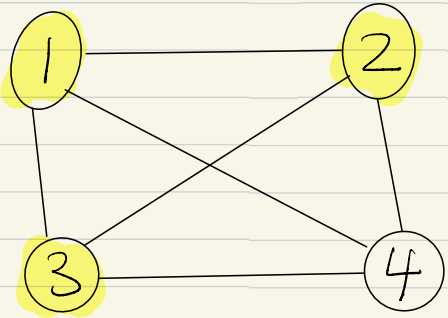
DFS (take & skip recursive) Backtracking

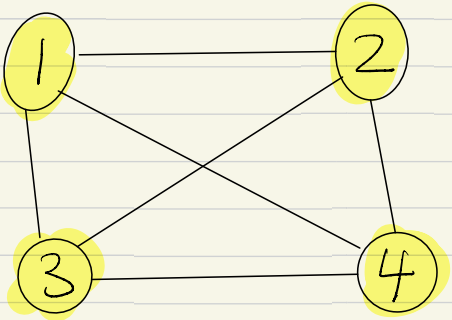
```
def dfs(r, n)
    if(r == 0)  return 1
    if(n == 0)  return 0

    return dfs(r-1, n-1) * n

def nPr(n, r)
    return dfs(r, n)
```

start from 1

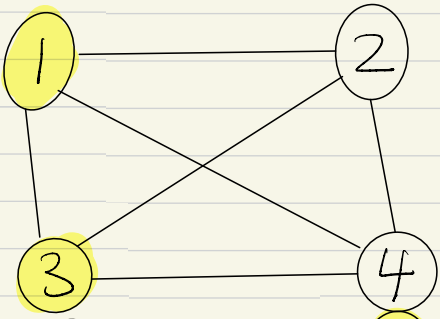


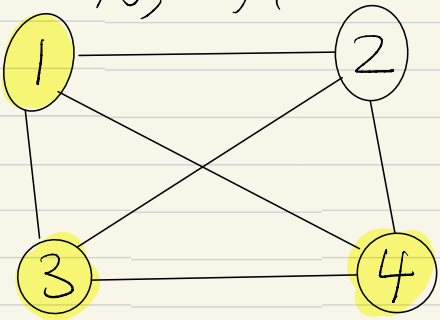


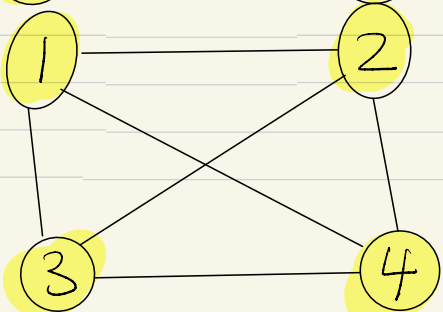




⇒ 1, 3, 2, 4







⇒ 1, 3, 4, 2

⇒ directions

candidates: directions

states: visited[][]

pruning: word.charAt(index) != board[x][y]

79. Word Search

```
def backtrack( board[][],
               word,
               index,
               i , j
               n , m
               visited[][])
```

2. Dynamic Programming

Time = # subproblems × time per subproblem

5 steps

1. define subproblems
2. relate subproblem by guessing
   - # no. of guess = # Indegree of vertex(v)
   - # no. of subproblem = # vertice in DP state Machine
   - Time per subproblem = O(# Indegree of vertex(v) + 1)

     // +1 for addition of w(u, v)

⇒ Optimal Structure

i.e To find best guess, try all (|V| choices) and use best.

- with at most k edges:

f(v,k) = min{ f(v, k−1), f(u, k−1) + w(u,v) | for all u to v }

relaxing (refer to SSP)

- DAG / Topological order

3. Memoization + recursion

- Top-down
- Bottom-up

// Topological order

for k = 0...|V|-1

No specific order

{ for v in V

for edges in adjList(v)

Total time = O(V(V+E))

4. combining subproblems - O(V)

(extra time to)

300. Longest Increasing Subsequnce

↳ return max(dp[])

Examples:	Fibonacci	Shortest Paths
subprobs:	F_k for $1 \le k \le n$	$\delta_k(s,v)$ for $v \in V$, $0 \le k < \lvert V \rvert$ = min $s \to v$ path using $\le k$ edges
# subprobs:	n	V^2
guess:	nothing	edge into v (if any)
# choices:	1	indegree$(v)+1$
recurrence:	$F_k = F_{k-1} + F_{k-2}$	$\delta_k(s,v) = \min\{\delta_{k-1}(s,u) + w(u,v) \mid (u,v) \in E\}$
time/subpr:	$\Theta(1)$	$\Theta(1 +$ indegree$(v))$
topo. order:	for $k = 1, \ldots, n$	for $k = 0, 1, \ldots \lvert V \rvert - 1$ for $v \in V$
total time:	$\Theta(n)$	$\Theta(VE)$ + $\Theta(V^2)$ unless efficient about indeg. 0
orig. prob.:	F_n	$\delta_{\lvert V \rvert - 1}(s,v)$ for $v \in V$
extra time:	$\Theta(1)$	$\Theta(V)$

5. Parent Pointer

- To know which guess is used at each subproblem

  And then walk back

- #KnackSack 01 (isSelected = new int[n])

prefix[:i] ⇒

```
def optimalSol(capacity, w, v){
    dp = dp(capacity, w, v)

    for(i = size(w) ... 1, j = capacity){
        if(dp[i][j] != dp[i-1][j]){
            isSelected[i-1] = true;
            j -= w[i-1]
        }
    }
    // bottom-up order
    return reversed(isselected)
```

item 4 is at index 3 of input arrays

7 != 6, so item 4 (v_4=5,w_5=4) was selected

		0	1	2	3	4	5	6	7
Empty	0	0	0	0	0	0	0	0	0
v_1=2,w_1=3	1	0	0	0	2	2	2	2	2
v_2=2,w_2=1	2	0	2	2	2	4	4	4	4
v_3=4,w_3=3	3	0	2	2	4	6	6	6	8
v_4=5,w_4=4	4	0	2	2	4	6	7	7	9
v_5=3,w_5=2	5	0	2	3	5	6	7	9	10

0 == 0, so item 1 (v_1=2,w_1=3) was not selected

		0	1	2	3	4	5	6	7
Empty	0	0	0	0	0	0	0	0	0
v_1=2,w_1=3	1	0	0	0	2	2	2	2	2
v_2=2,w_2=1	2	0	2	2	2	4	4	4	4
v_3=4,w_3=3	3	0	2	2	4	6	6	6	8
v_4=5,w_4=4	4	0	2	2	4	6	7	7	9
v_5=3,w_5=2	5	0	2	3	5	6	7	9	10

// for f(i) sample

- Or DP[i][0] → value
  DP[i][1] → Parent

i = 0, dp[n] = (0, None)

while (i != None)
  i = DP[i][1]

⇒ Subproblem as suffix [i:]

- Guess as where to end line at j [i:j]

  # no. of choices = n - i = O(n)

- Base case: DP[n]
- Top Order: From n to 0
- Total Time:

  # no. of subproblems = O(n) for [i:]

  X

  # no. of Guess = O(n) for # n - i per [i:j]

  = O(n²)

- Goal: DP[0] for [0:]

⇒ Prefix [:i] gets played

And

suffix [i:] left

where i = #no. of cards "Already played"

- #Knack Sack 01

$$f(i, c) = \max\{ f(i-1, c), f(i-1, c - w_i) + v_i \}$$

// Base case: $f(i, 0) = 0$

$f(0, c) = 0$

suffix way: i+1

prefix way: i-1

To elegantly do the bottom-up do, we need N/S units for the parameters in state machine to handle the base cases(i.e "empty items left" state)

But NOT n-1 or s-1. e.g for prefix, we don't want to handle f(0-1)=f(-1)=dp[-1]

Goal: f(n, c) // we have n items and remaining capacity = c

* f(0, j) ⇒ we have zero items left and remaining capacity = j

* let say we have 4 items naming 1, 2, 3, 4 for ith item, w_i / v_i is at the index i-1 of the input array = w[i-1], v[i-1]

* if using suffix[i:], w_i/v_i is at w[i]/v[i]

Knapack: suffix[i:] of candidates(items) in arbitrary order with a constraint factor (i.e remaining capacity)

//Even though the suffix[i:] is in use , it is no longer a concept of sequence.
//Alternatively: prefix[:i] can be used to represent iCr

Try all possible Combinations instead of permutations
Order of items doesn't matter here

- Subproblem:
  - The max sum of value
    - for a combination of items suffix[i:] - n-i+1 C r
      - the total weight of r *items* must be <= X

1. subproblem = value for suffix i:
given knapsack of size X
$\implies$ # subproblems $= O(nS)$!

- Recurrence Relations:

- $DP[i, X] = \max(DP[i+1, X], v_i + DP[i+1, X - s_i]$ if $s_i \leq X)$
- $DP[n, X] = 0$

$\implies$ time per subproblem $= O(1)$

- bottom up from suffix[n:] to suffix[0:]

4. topological order: for i in $n, \ldots, 0$: for X in $0, \ldots S$
total time $= O(nS)$

- Goal:

5. original problem $= DP[0, S]$
(& use parent pointers to recover subset)

⇒ Subproblem for string & subsequence x

- suffix $x[i:]$
- prefix $x[:i]$

} $O(|x|) \Rightarrow$ [ure if possible]

- substring $x[i:j] \rightarrow O(x^2)$

Parathesization $[i:j]$

DP is NOT JUST shortest Path In DAG + where to break in the middle(k)

- # no. of subproblem
  $= \#$ substring $= O(n^2)$

- Guess : where is the middle point between the parenthesis $[i:k]$ and the parenthesis $[k:j]$

↳ where to break at index k

↳ #choice $= \#k = j - i + 1 = O(n)$

- $f(i,j) = \min\{ left(i,k) + right(k,j) + cost\ at\ k \}$ for all k in range$(i+1, j)$

- Base Case: $f(i, i+1) = 0$

- Top Order: From size = 1 to size = n (Increasing size)

- Goal: $DP[0][n]$

⇒ Edit Distance: Multiple string input $\{x[i:]$ and $y[j:]\}$

How to deal with 2 inputs of string x and string y ?

- Subproblem(i,j) = Min edit distance $\{x[i:], y[j:]\}$

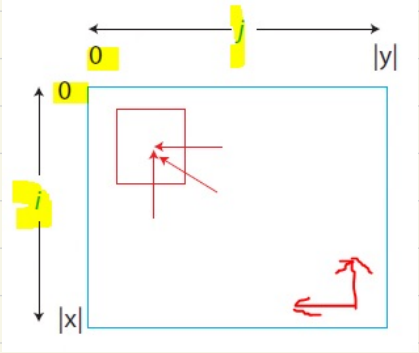
#Subproblem = #no. of substring X · #no. of substring Y

$= O(|x| \cdot |y|)$

- Guess: 3 ways to convert x to y
  - insert y[j] in front of x[i:]
  - delete x[i]
  - replace x[i] with y[j]

choices $= O(3) = O(1)$

- Recurrence Relation: f(i,j) = min{
  move down: cost(insert) + f(i,j+1) ,
  move right: cost(delete) + f(i+1,j),
  diagonal move: cost(replace) + f(i+1,j+1)
  }
- base case: f(|x|, |y|) = 0
- Topological Order:
  for i=|x|.....0
  for j=|y|.....0



- Goal: f(0,0) = min edit distance{x[0:] , y[0:]}

Space:
- only need to keep last 2 rows/columns
  $\Longrightarrow$ linear space

Time: O(1) * O(|x||y|) = O(|x||y|)

⇒ 2 kinds of guessing

A) guess which other subproblems to use
(used by every DP except Fibonacci)

B) create more subproblems to guess / remember more structure of solution used by Knapsack DP

- wrong attempt to define subproblems without enough parameters to describe the exact subproblems

↳ missing parameter for the function d()
↳ missing information for the constraint factor at each state

⇒ add more parameter to the state to fully describe the constraint at each state

Hard Questions: 188. Best Time to Buy and Sell Stock IV

Top 5 DP

1. Fibonacci Numbers

```
"/**
    * Recurrence Relation:
    *    f(k) = f(k-1) + f(k-2)
    *    f(2) = f(1) + f(0)
    *         = 1 + 1
    *         = 2
    *
    * Base Case:
    *   f(0) = 1
    *   f(1) = 1
    */"
```

198. House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

```
"/**
    *  Recurrence Relation:
    *      f(i) = max{arr[i] + f(i+2) , f(i+1)}
    *  Base:
    *      f(>=n) = 0
    *  Goal:
    *      f(0)
    */"
```

2. Zero/One knapsack

416. Partition Equal Subset Sum

Given an integer array nums, return true if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or false otherwise.

Example 1:

```
Input: nums = [1,5,11,5]
Output: true
Explanation: The array can be partitioned as [1, 5, 5] and [11].
```

2. Guessing

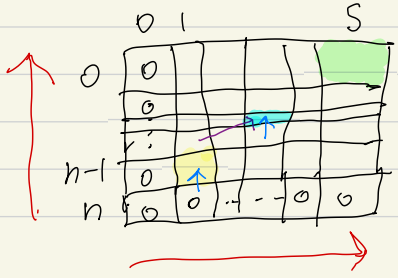
$$f(i, R) = \max\{ f(i+1, R - nums[i]) + nums[i] \text{ if } nums[i] \le R,\ f(i+1, R) \text{ //skip} \}$$

3. Goal: $f(0, S)$

base: $f(n, ?) = 0$

$f(?, 0) = 0$

4. Top Order



(? (d space) ←

3. Unbounded Knapsack

322. Coin Change

```
"/**
         * s = amount
         * Recurrence Relation
         *      f(s) = min(f(s-coin) + 1 | for any coin
in coins)
         *
         * Base case:
         *      f(0) = 0; //valid combination
         *      f(s<0) = -1; //invalid combination
         *
         * max possible #no = amount
         *
         *             to find f(n)
         */"
```

518. Coin Change II

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the number of combinations that make up that amount. If that amount of money cannot be made up by any combination of the coins, return 0.

You may assume that you have an infinite number of each kind of coin.

The answer is guaranteed to fit into a signed 32-bit integer.

Example 1:

```
Input: amount = 5, coins = [1,2,5]
Output: 4
Explanation: there are four ways to make up the amount:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

```
"/**
        * Idea:
        *  - dfs/backtrack for a combination nCr
        *    - not quit so ....becos 1 + 1 + 1 + 2 = 5 , 1 is duplicate number
        *  - duplicate number/coin is allowed in a combination
        *     - use it repeatedly until going to another coin
        *       - traverse by suffix[i:] (or prefix[:i]) to make sure previous
coins won't be reused
        *            - e.g coins[1, 2, 5]
        *              - 1+.....+1+ 2+....+2+ 5+....+5 in a particular sequence
        *              - X Ones , Y twos , Z fives   for 0<=(X, Y, Z)
        * subproblem:
        *    suffix[i:] with sum = amount
        *
        * Recurrence Relations:
        *  f(i, k) = f(i, k - coins[i]) + f(i+1, k) //suffix way: i+1 , prefix way: i-1
        *
        * Base Case:
        *  f(i,0) = 1
        *
        * Goal:
        *  f(0,S) //suffix[0:]
        */"
```

4. Longest Common Subsequence

1143. Longest Common Subsequence

```
"/**
        * x = text1
        * y = text2
        *
        * Recurrence Relation:
        *  - f(i,j) = max{ f(i+1, j) for move forward i, f(i,j+1) for move forward j,
f(i+1,j+1) + equal(i,j) }
        *
        * Base case:
        *  - f(|x|, j) = 0
        *  - f(i, |y|) = 0
        *
        * Goal:
        *  - f(0,0) for x[0:] and y[0:]
        *
        */"
```

300. Longest Increasing Subsequence

Given an integer array nums, return the length of the longest strictly increasing subsequence

Example 1:

```
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore
the length is 4.
```

please refer to notes

" //N choices for a subproblem(i): N numbers points to a number at i
//N subproblem(i) for a sequence which ends at i
//O(N^2)

//version 1
//from vertex i
//to different vertex j(i+1.....n)

//version 2
//from different vertex j (0.....i-1)
//to the vertex i"

```python
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        LIS = [1] * len(nums)

        for i in range(len(nums) - 1, -1, -1):
            for j in range(i + 1, len(nums)):
                if nums[i] < nums[j]:
                    LIS[i] = max(LIS[i], 1 + LIS[j])
        return max(LIS)
```

5. Palindromes

inward to center

```
"public String longestPalindrome(String s) {
    int max = 0;
    int start = 0;
    boolean[][] dp = new boolean[s.length()][s.length()];
    for(int size=1;size<=s.length();size++)
    {
       for(int i=0, j = i+size-1 ;j<s.length();j=++i+size-1)
       {
```

516. Longest Palindromic Subsequence

Given a string s, find the longest palindromic subsequence's length in s.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Example 1:

Input: s = "bbbab"
Output: 4
Explanation: One possible longest palindromic subsequence is "bbbb".

516. Longest Palindromic Subsequence

Given a string s, find the longest palindromic subsequence's length in s.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the

"/*

Sub-problem:

substring[i:j]

Recurrence Relation:

f(i,j) = max{ f(i+1,j),

f(i,j-1) ,

f(i+1,j-1) + (s[i]==s[j] ? 2 : 0)

| if i<=j }

Base case:

f(i,j=i) = 1

Goal:

f(0, n-1)

Take-away:

Could we apply the above recurrence relation to question - longest palindrome substring ?

- let's say if we're gonna return the value of the longest length of palindrome

- instead of substring as return type

- Why can't we use Math.max to resolve it ?

- No. becos the integer value of sub-problem cannot tell if is palindrome

- The main difference why above function works is

- subsequence(0/1 in order) vs substring(contiguous)

*/"

生活是
計劃、計劃
再計劃

計劃是
平衡、
平衡、
再平衡。

Divide & conquer And Sorting

1. Divide & conquer

=> **A balanced or unbalanced tree** *with each node as a subproblem/recursive call*

Steps:

- **Divide** :
  - Breaking the problem into smaller sub-problems Recursively until ***no*** sub-problem is ***further divisible***
- **Conquer** :
  - The sub-problems are considered ***'solved' on their own***
- **Merge** :
  - *Recursively* ***combines*** them until they formulate a solution of the original problem

Advantages:

1. Solving difficult problems
   a. Tower of Hanoii problem
2. Algorithm efficiency
   a. Merge-sort and quick-sort (O(nlogn))
3. *Parallelism*
   a. distinct sub-problems can be executed on different processors.
   b. Map-reduce

Master Theorem

```
procedure T( n : size of problem ) defined as:
  if n < some constant k then exit

  Create 'a' subproblems of size n/b in d(n) time

  repeat for a total of 'a' times
     T(n/b)
  end repeat

  Combine results from subproblems in c(n) time

end procedure
```

=>Formula:
T(n) = aT(n/b) + f(n) , f(n) = d(n) + c(n) , a>=1, b>=2

$$T(n) = a\,T(n/b) + \Theta(n^c)$$

=>3 General Form:

- If $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$
- If $c = \log_b a$, then $T(n) = \Theta(n^c \log n)$
- If $c > \log_b a$, then $T(n) = \Theta(n^c)$

Divide-and-conquer recurrences. For each of the following recurrences we assume $T(1) = 0$ and that $n / 2$ means either $\lfloor n / 2 \rfloor$ or $\lceil n / 2 \rceil$.

RECURRENCE	$T(n)$	EXAMPLE
$T(n) = T(n/2) + 1$	$\sim \lg n$	*binary search*
$T(n) = 2T(n/2) + n$	$\sim n \lg n$	*mergesort*
$T(n) = T(n-1) + n$	$\sim \frac{1}{2} n^2$	*insertion sort*
$T(n) = 2T(n/2) + 1$	$\sim n$	*tree traversal*
$T(n) = 2T(n-1) + 1$	$\sim 2^n$	*towers of Hanoi*
$T(n) = 3T(n/2) + \Theta(n)$	$\Theta(n^{\log_2 3}) = \Theta(n^{1.58\ldots})$	*Karatsuba multiplication*
$T(n) = 7T(n/2) + \Theta(n^2)$	$\Theta(n^{\log_2 7}) = \Theta(n^{2.81\ldots})$	*Strassen multiplication*
$T(n) = 2T(n/2) + \Theta(n \log n)$	$\Theta(n \log^2 n)$	*closest pair*

2. Sorting

QuickSort - unstable sort , Array(much Random-Access), In-memory Access

- Space Complexity:
  - Quicksort with **in-place** and unstable partitioning uses only constant additional space before making any recursive call.
  - a **space** complexity of O(log n), even in the worst case,

Merge Sort - Stable sort, Linked-List(Traversal instead of Random Access), Huge data with Hard-drive Access

- Space Complexity:
  - **Not sort in place**
  - The memory size of the input must be allocated for the sorted output to be stored in
  - O(n)

=> PriorityQueue(Binary Heap Array)

- Binary Heap Array
  - The root element will be at Arr[0].
  - For i-th node - Arr[i] , below are its corresponding parent/childs

Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node

  - Queue operations
    - Time Complexity: O(**log**(n)) time for the enqueing and dequeing methods (offer, poll, remove() and add)

Java Implementation:

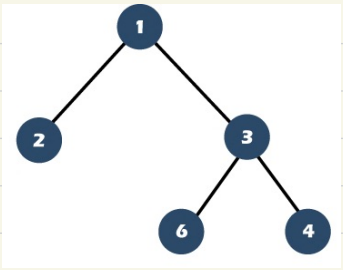
- MergeSort/Timsort for object
  - Becos object has more than 1 attribute, so for cards, 1st sorted by number, 2nd sorted by color, then stable sort fits this use case.
- Quick sort for primitive
  - only 1 attribute , no need for stable sort but sort in place.
- Heap sort using priorityQueue
  - Sort some data by adding it to a PriorityQueue and then removing it again.
    - in descending order
      - PriorityQueue q = new PriorityQueue(1000, Collections.reverseOrder())

ALGORITHM	IN PLACE	STABLE	BEST	AVERAGE	WORST	REMARKS
selection sort	✓		½ n^2	½ n^2	½ n^2	n exchanges; quadratic in best case
mergesort		✓	½ n lg n	n lg n	n lg n	n log n guarantee; stable Space: O(n)
quicksort	✓		n lg n	2 n ln n	½ n^2	n log n **probabilistic** guarantee; **fastest in practice** Space: O(logn)
heapsort	✓		n (equal keys) lg n (distinct keys)	2 n lg n	2 n lg n	n log n guarantee; in place Space: **O(1)**

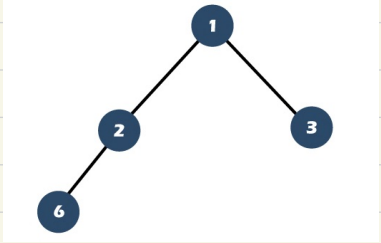
Functional Data-structure

Tree

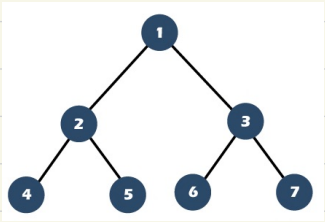
n = 2m+1-1 for a perfect binary tree



full

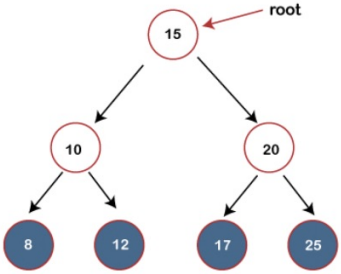


complete



perfect

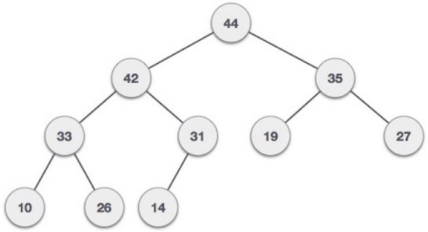
- **Binary Search Tree** (sorted binary trees)
  - The value of the left node <= the parent node
  - the value of the right node >= the parent node.
  - Binary Search:
    - The middle element = the root node
    - Almost a balanced tree: **O(logn)**
    - Either left or right-skewed: **O(n)**
  - Types:
    - AVL trees
    - Red-black trees



AVL Tree (a self-balancing binary search tree)
A height balanced tree (BalanceFactor <= 1)
BalanceFactor = height(left-sutree) – height(right-sutree)

Red-black Tree (a self-balancing binary search tree)
The color of the node
. Red or Black

- Heap (A **Complete** binary tree): MIT Reference: heap
  - **The Complete Binary Tree is always balanced.**
  - Max-Heap
    - If **α** has child node **β**
      - **key(α) ≥ key(β)**
    - Where the value of the root node is greater than or equal to either of its children



  - Build Heap: **O(n)**
    - produce a max-heap from an **unordered array**
    - from wiki: Binary heap

      Building a heap from an array of n input elements can be done by starting with an empty heap, then successively inserting each element. This approach, called Williams' method after the inventor of binary heaps, is easily seen to run in O(n log n) time: it performs n insertions at O(log n) cost each. However, Williams' method is suboptimal. **A faster method** starts by arbitrarily putting the elements on a binary tree, respecting the shape property (the tree could be **represented by an array**)

#Priority queues

DATA STRUCTURE	INSERT	DEL-MIN	MIN	DEC-KEY	DELETE	MERGE
array	1	*n*	*n*	1	1	*n*
binary heap	log *n*	log *n*	1	log *n*	log *n*	*n*
binomial heap	1	log *n*	1	log *n*	log *n*	log *n*
Fibonacci heap	1	log *n* †	1	1 †	log *n* †	1

#Searching

	worst case			average case		
DATA STRUCTURE	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
sequential search (in an unordered list)	*n*	*n*	*n*	*n*	*n*	*n*
binary search (**in a sorted Array**)	log *n*	*n*	*n*	log *n*	*n*	*n*
binary search tree (**unbalanced**)	*n*	*n*	*n*	log *n*	log *n*	sqrt(*n*)
red-black BST (left-leaning)	log *n*	log *n*	log *n*	log *n*	log *n*	log *n*
AVL	log *n*	log *n*	log *n*	log *n*	log *n*	log *n*

stack

6 Types Of Toxic People

The Energy Drainer

They make you feel tense. They put you down for no reasons. Can't be happy for other's good fortune.

The Fake Complimentor

Gives you fake compliments. Lacks empathy. Put you in uncomfortable positions.

The Pessimist

Talks down to you to make themselves feel better. Only cares about themselves. Tries to talk you out of your dreams.

The Criticiser

Doesn't support in your decisions. Criticizes every move you make. Makes you feel like you can't do anything right.

The Manipulator

Tries to control everything. Pretends to like you and other people. Wants to make every decision for themselves & others

The Victim

Blames others for their misfortune. Constantly seek attention from others Talks mostly about their excuse for failing

Greedy

1. Properties of Greedy Algorithms

- **Optimal Substructure**: the **optimal** solution **to a problem incorporates** the **optimal** solution to **sub**problem(s)

- **Greedy choice property**: **locally** optimal **choices lead to** a **globally** optimal solution
  - We can make whatever **choice seems best at the moment** and **then solve** the subproblems that arise later. The choice made by a greedy algorithm **may depend on choices made so far**, **but not on future choices or all** the solutions **to the subproblem**. It **iteratively** makes **one** greedy choice **after another**, **reducing** each given problem **into a smaller** one.
  - In other words, a greedy algorithm **never reconsiders its choices**. This is the **main difference from dynamic programming**, **which is *exhaustive*** and is ***guaranteed to find the solution***.
    - After every stage, dynamic programming **makes decisions based on all** the decisions made in the **previous** stage and may reconsider the **previous** stage's **algorithmic path to the solution**.

- 5 functions
  - A **candidate set**
    - from which a solution is created
  - A selection function
    - which **chooses the best candidate** to be added to the solution
  - A feasibility function
    - that is used to **determine if** a candidate can be used to **contribute to a solution**
  - An objective function
    - which assigns a value to a solution, or a **partial solution**, and
  - A solution function
    - which will indicate when we have discovered a **complete solution**

2. "Cut Property" tool to identify Above properties (prove) greedy

- a "cut" is a partition of vertices in a "graph" into two disjoint subsets.
  - (B, A, E) forms one subset, and (C, D) forms the other subset.
- a crossing edge is an edge that connects a vertex in one set with a vertex in the other set.
  - (B, C), (A, C), (A, D), (E, D) are all "crossing edges".

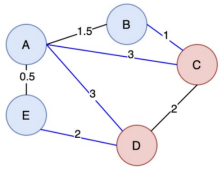
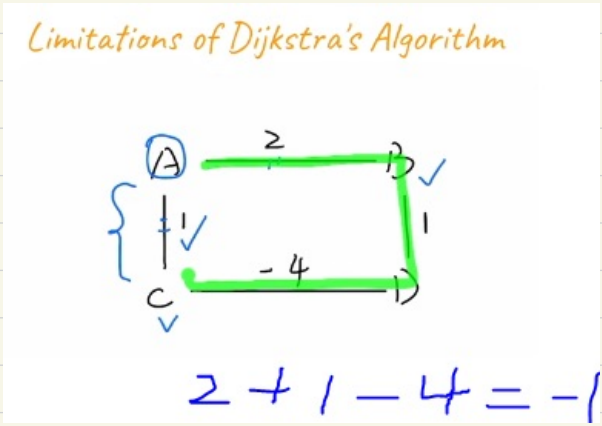
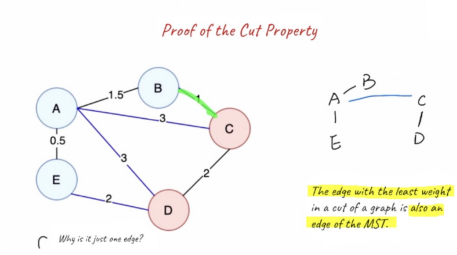


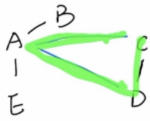
Figure 10. Graph with a cut



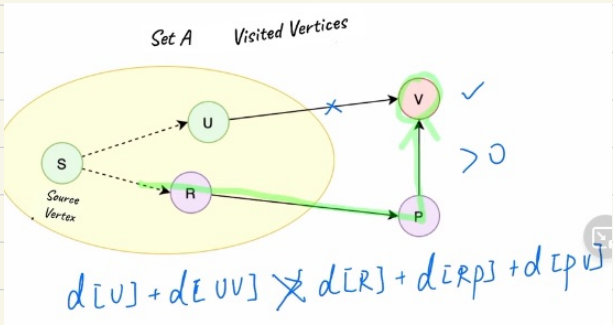
For any cut C of the graph, if the weight of an edge e in the cut-set of C is strictly smaller than the weights of all other edges of the cut-set of C, then this edge belongs to all MSTs of the graph.



Why is it just on edge



This is a sub-MST problem (A, C , D)



Sample Question

FractionalKnapsack

find-minimum-number-of-coins

* Given a value V, if we want to make a change for V Rs,
* and we have an infinite supply of each of the denominations in Indian currency,
* i.e., we have an infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes,
* what is the minimum number of coins and/or notes needed
* to make the change?
*
* Input: V = 70
* Output: 2
* We need a 50 Rs note and a 20 Rs note.
*
* Input: V = 121
* Output: 3
* We need a 100 Rs note, a 20 Rs note and a 1 Rs coin.

```
public int coinChange(int amount) {
        Arrays.sort(deno);

        int count = 0;
        for(int tmp=amount, i=deno.length-1;tmp>0;tmp-=deno[i]){
            while(deno[i]>tmp){
                i--;
            }

            count++;
        }

        return count;
    }
```

Comparison

- Divide-and-Conquer：Divide-and-Conquer 和 Dynamic Programming都是將問題切割再採用遞迴方式處理子問題，但是Divide-and-Conquer子問題的解**通常不會重覆**，重複時，通常會對相同子問題進行重覆計算，**而不會像**Dynamic Programming的子問題有大量的重複(overlap)，可以以table儲存不用再次計算，**用空間換取時間**。
  - DC!=DP!=DC + memo , it is a fatal misconception to regard them similar
    - Please refer to the detail dynamic-programming-vs-divide-and-conquer-approach
      - e.g optimal structure in DP, master theory in DC etc…

- Greedy Approach：Greedy Approach具有*Selection Procedure*，自某起始點開始在每一個階段逐一檢查每一個輸入是否適合加入答案中。**如果**所要處理的最佳化問題**無法找到一個選擇程序(e.g ordered)**來逐一檢查，**而需要以一次考慮所有的可能情況**的方式來處理，那就是屬於Dynamic Programming。
  - 因此若遇**最佳化問題**，**先思考可否用Greedy Approach解**，若不行**再考慮Dynamic Programming**。

Coin Change (recursion inside for loop without eliminating candidates/coins at each iteration = No Selection process)

Types of problems they solve

We mostly solve optimization and counting problems using dynamic programming where the solution space is very large. But there is no such pattern in divide and conquer problems.

Enjoy learning, Enjoy algorithms!

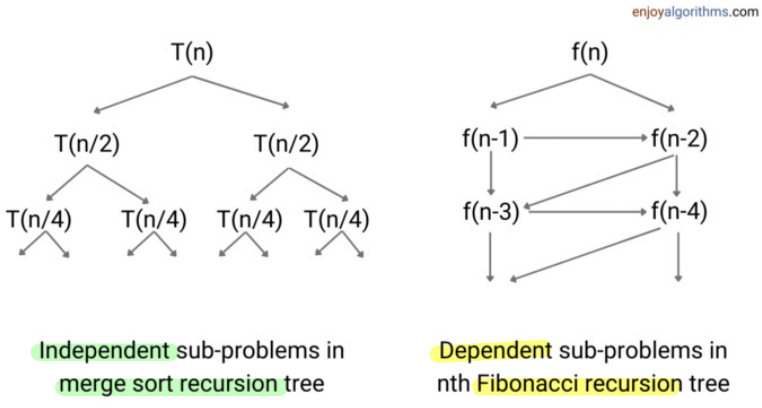
Divide and conquer and dynamic programming are popular problem-solving approaches in data structure and algorithms. Both approaches look similar in one way: They use a similar idea to break problems into subproblems and combine their solutions to obtain the solution to the original problem. But there are a lot of differences between both approaches in terms of:

- The nature of sub-problems involved in the solution
- Performance when sub-problems are overlapping
- Thinking and building recursive solution
- Time and space complexity analysis
- State and input size of subproblems
- Types of problems they solve

Let's understand the difference between Divide and conquer and dynamic programming with respect to each one of the above points.

Nature of sub-problems involved in the solution

We apply divide and conquer when sub-problems are independent of each other and apply dynamic programming when subproblems are dependent on each other. In other words, in the recursive solution of the divide and conquer problems, we decompose the problem into independent subproblems but in the recursive solution of dynamic programming problems, it is almost always decomposed into dependent and overlapping subproblems.



Independent sub-problems in merge sort recursion tree

Dependent sub-problems in nth Fibonacci recursion tree

Data-Specific Algorithm

Pointers

1. sliding window

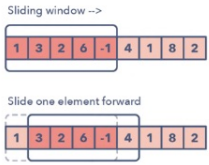
#Dufq

- iterated over in sequence
- Contiguous means that the elements are sequentially placed next to each other

strings
arrays
linked lists

Sliding Windows start from the 1st element and

- Keep shifting right by one element



- Adjust the length of the window
  - In some cases, the window size remains constant
  - in other cases the sizes **grows** or **shrinks**.

- 決定何時該縮減 window ，也等同於決定何時應該停止增長 window ，只有當縮減 window 的條件不再成立，才能向右增長 window 。

```
int[] chars = new int[128]; //letters, digits, symbols and spaces
int max=0;
for(int i=0,j=0; i<s.length();i++){
    while(chars[s.charAt(i)]>0){
        chars[s.charAt(j)]--;
        j++;
    }

    chars[s.charAt(i)]++;
    max = Math.max(max, i-j+1);
}
```

i) Given an array as input, extract the **pair** of contiguous integers that have the highest sum of all pairs. Return the pair as an array.

[5, 2, 4, 6, 3, 1] [5, 2, 4, 6, 3, 1] ..... [5, 2, 4, 6, 3, 1]

ii)Given a string, find the length of the **longest substring** without repeating characters.

" p w w k e w " " p w w k e w " " p w w k e w " " p w w k e w "

" p w w k e w " " p w w k e w " " p w w k e w "

Considering that the **substring** can be any length ≤ **26**. In an example with less duplicate characters, the solution may even run in strict linear time.

By using HashSet **as** a sliding window

- only need to check if s[j] is already in the substring sij - O(1)
  - " p w w k e w "
    - As 'ww' is duplicate and part of the 'pwwk', whatever substring containing 'kk' between p and k is invalid , we can only start from 'wk' (to excludes 'ww').
- 有效減少不必要的 iteration ， 進而降低時間複雜度 - O(2n)
  - " p w w k e w "
    - we does verify 'max' by 'w', 'wk', 'wke' but excludes 'ke'
    - becos 'ke' is part of 'wke' and must be shorter than 'wke'. which eliminates j for-loop in brute force.

- Time complexity : O(2n)=O(n). In the worst case each character will be visited twice by i and j.
- Space complexity (HashMap) : O(min(m,n))

- Space complexity (Table): O(m). m is the size of the charset.
  - *If we know* that the charset is rather small, we can replace the Map with an integer *array as direct access table*.
  - Commonly used tables are: int[] chars = new int[?];
    - int[26]
      - for Letters 'a' - 'z'
        - chars[s.charAt(i) - 'a']++;
      - **or** 'A' - 'Z'
        - chars[s.charAt(i) - 'A']++;
    - int[128] for ASCII
    - int[256] for Extended ASCII

Any problem for any of the following return values:
Minimum value
Maximum value
Longest value
Shortest value
K-sized value
Maximum sum subarray of size 'K' (easy)
Longest substring with 'K' distinct characters (medium)
String anagrams (hard)
/ˈæn.ə.græm/ a word or phrase made by using the letters of another word or phrase in a different order

```
/**
    * Given a string s and an integer k, return the length of the longest substring of s that
contains at most k distinct characters.
    *
    *
    *
    * Example 1:
    *
    * Input: s = "eceba", k = 2
    * Output: 3
    * Explanation: The substring is "ece" with length 3.
    * Example 2:
    *
    * Input: s = "aa", k = 1
    * Output: 2
    * Explanation: The substring is "aa" with length 2.
```

```
Map<Character, Integer> chars = new LinkedHashMap<>( (k+1) * 4/3 + 1);
        int max = 0;

        for(int i=0, j=0;i<s.length();i++){
            Character c = s.charAt(i);
            if(chars.containsKey(c)){
                chars.remove(c);
            }
            chars.put(c, i);

            if(chars.size() > k){
//                int leftMost = Collections.min(chars.values());
                int leftMost = chars.values().iterator().next();
                j = leftMost + 1;
                chars.remove(s.charAt(leftMost));
            }

            max = Math.max(max , i-j+1);
        }
```

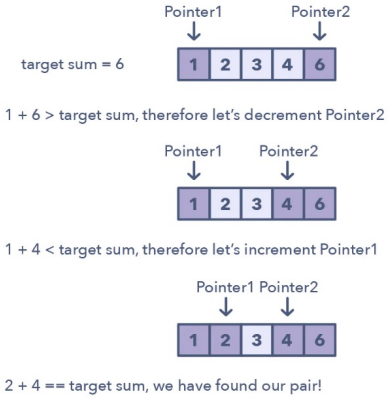
```
/**
    * Given two strings s and t of lengths m and n respectively, return the minimum window
    * substring
    * of s such that every character in t (including duplicates) is included in the window. If there
is no such substring, return the empty string "".
    *
    * The testcases will be generated such that the answer is unique.
    *
    *
    *
    * Example 1:
    *
    * Input: s = "ADOBECODEBANC", t = "ABC"
    * Output: "BANC"
    * Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from
string t.
    *
    * Example 2:
    *
    * Input: s = "a", t = "a"
    * Output: "a"
    * Explanation: The entire string s is the minimum window.



   /*
            1. use a map to store the key to remaining number of chars that is not yet
   satisfied (i.e it can be over-statisfied - negative)
            2. shrink(shift j forward) to approch the miminum of window size till the condition
   is no longer met
            3. use of a counter to keep track of how many chars in the string 't' we meet
         */

        /*
            time: O(m)
            space: O(m)
         */
        Map<Character, Long> m = t.chars()
                        .mapToObj(c->(char)c)
                        .collect(Collectors.groupingBy(Function.identity(),
                              Collectors.counting()));
        int start=0,end=0;
        int min = Integer.MAX_VALUE, counter = t.length();
```

2. Two Pointers

Two Pointers is often useful when **searching pairs** in a ***sorted*** array or linked list.

Two pointers are needed because with **Single** pointer, you would have to **continually loop back** through the array to find the answer.

- This back and forth with a single iterator is **inefficient for time and space complexity** — a concept referred to as asymptotic analysis.
- While the **brute force** or naive solution with 1 pointer would work, it will produce something along the lines of **$O(n^2)$.**
- In many cases, **two** pointers can help you find a solution with **better space or runtime complexity**.



The two-pointer technique sometimes will relate to **Greedy** Algorithm which helps us design our pointers' **movement** strategy.

左右指標技巧用於想從頭跟尾同時出發的問題，尤其題目要求做到 in-place （Space Complexity 為 O(1)）時，就可以使用左右指標技巧，必須注意的是處理的資料**必須要排序過**，否則使用雙指標技巧就沒意義。

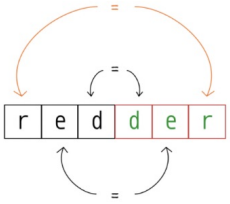
when to use the Two Pointer method:
sorted arrays (or Linked Lists)
need to find a set of elements that fulfill certain constraints
The set of elements in the array
a pair
a triplet /ˈtrɪp.lət/ one of three children born to the same mother at the same time
a subarray

problems:
Squaring a sorted array (easy)
Triplets that sum to zero (medium)
Comparing strings that contain backspaces (medium)

i) Given a string s, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

Example 1:

```
Input: s = "A man, a plan, a canal: Panama"
Output: true
Explanation: "amanaplanacanalpanama" is a palindrome.
```



Since the input string contains characters **that** we need to ignore in our palindromic check, it becomes tedious to figure out the real middle point of our palindromic input.

Instead of going outwards from the middle, we could just **go inwards towards** the **middle**!

```
        for(int i=0, j=s.length()-1; i<j; i++, j--){
            while(i<j && !Character.isLetterOrDigit(s.charAt(i))){
                i++;
            }

            while(i<j && !Character.isLetterOrDigit(s.charAt(j))){
                j--;
            }

            if(Character.toLowerCase(s.charAt(i)) !=
Character.toLowerCase(s.charAt(j))){
//                result = false;
                return false; // break the loop immediately
            }
        }

        return true;
```

ii) Given a 1-indexed array of integers numbers that is already sorted in non-decreasing order, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= first < second <= numbers.length. Return the indices of the two numbers, index1 and index2, as an integer array [index1, index2] of length 2.

The tests are generated such that there is **exactly one solution**. You **may not use the same** element **twice**.

Example 1:

```
Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore index1 = 1, index2 = 2.
```

We can apply **Two Sum**'s solutions directly to get

- O(n^2) time, O(1) space using brute force
- O(n) time, O(n) space using hash table.

However, both existing solutions do not make use of the property that

- the input array is **sorted**.
- We can do better.

If the sum is equal to target, we found the exactly only solution.

- If it is less than target, we increase the smaller index by one.
- If it is greater than target, we decrease the larger index by one.
- Move the indices and repeat the comparison until the solution is found.

Let [... , a, **b**, c, ... , d, **e**, f, ...] be the input array that is sorted in ascending order and let the elements b and e be the exactly only solution.

- Because we are moving the smaller index from left to right, and the larger index from right to left, at some point, one of the indices **must reach either b or e.**
- Without loss of generality, **suppose** the smaller index **reaches b first.** At this time, the sum of these two elements **must be greater than target**.
- Based on our algorithm, we will keep moving **the larger** index to **the left until we reach** the solution.

Two-Sum

```
int i=0,j= numbers.length - 1;
while(i<j){
    if(numbers[i] + numbers[j] > target){
        j--;
    } else if(numbers[i] + numbers[j] < target) {
        i++;
    } else {
        return  new int[]{i+1, j+1};
    }
}

return null;
```

3. Fast - slow Pointers

when to use:
Deal with a loop in a linked list or array
To know the position of a certain element or the length of the linked list.

Use it over the Two Pointer method
a singly linked list where you can't move in a backwards direction.
To determine if a linked list is a palindrome

Two pointers which move through the array (or sequence/linked list)

- **at different speeds**.
- This approach is quite useful when **dealing with cyclic** linked lists or arrays.

By moving at different speeds (say, in a cyclic linked list), the algorithm proves

- that the two pointers are **bound to meet**.
- The **fast** pointer should **catch the slow** pointer **once both** the pointers are **in a cyclic loop**

透過一慢一快的兩個指標朝著同個方向前進，其中一個指標用於順序走訪 ，而另外一個則根據需求來訂定移動的策略，這樣的技巧通常用於解決 linked list 的各種問題。

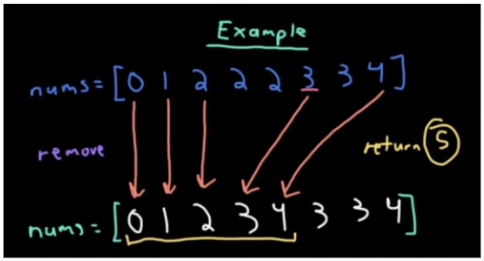
利用**快慢指標**可以解決以下問題

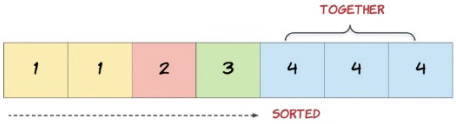
1. 將 sorted array 中**重複元素剔除**
2. 判斷 linked list 是否**有環**
3. 尋找 linked list 的環**起點**
4. 尋找 linked list 的**中間**節點
5. 尋找 linked list 上**倒數第 K 個點**

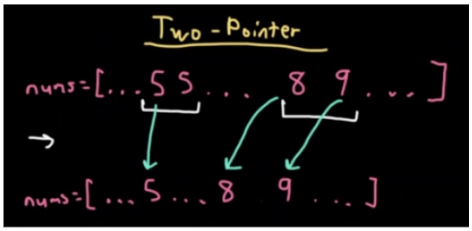
↳ 19. Remove Nth Node From End of List

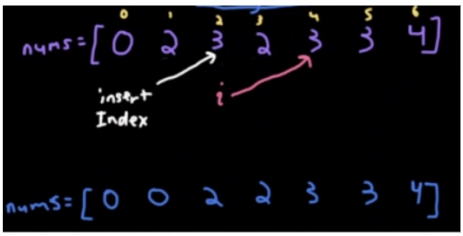
Problems:
Linked List Cycle (easy)
Palindrome Linked List (medium)
    it is more like reverse linked list problem
    fast-slow pointers help to get the end of the first half
Cycle in a Circular Array (hard)

i) Given an integer array nums sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**.









Complexity analysis

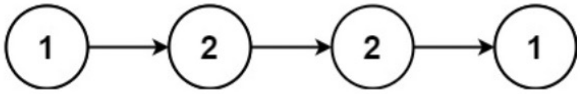
- Time complextiy : O(n)). Assume that n is the length of array. Each of i and j traverses at most n steps.
    - Compared with SortedSet approach for removing duplicates
        - O(logn) for insertion
            - O(n x logn) // do it n times

- Space complexity : O(1)
    - Compared with SortedSet approach for removing duplicates
        - O(n) for n distinct elements(worst)

ii) Palindrome (plz also refer to Cyclic sort & In-place reversal)

Given the head of a singly linked list, return true if it is a palindrome.

Example 1:

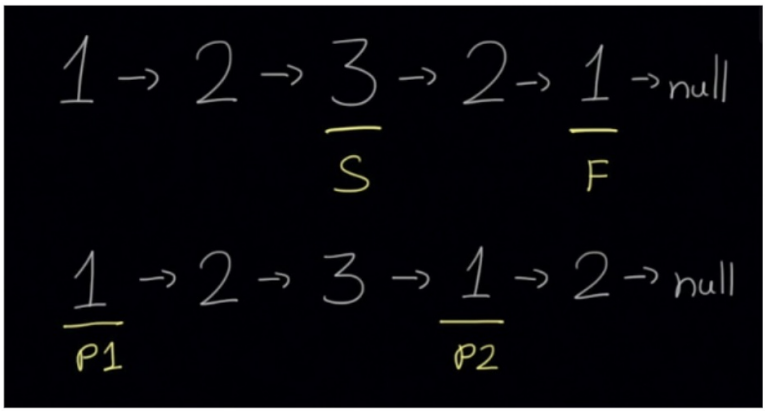


Input: head = [1,2,2,1]

Output: true

Approach 3: Reverse second half In-place

The ***only*** way we can avoid using O(n) extra space is *by modifying the input* **in-place**.



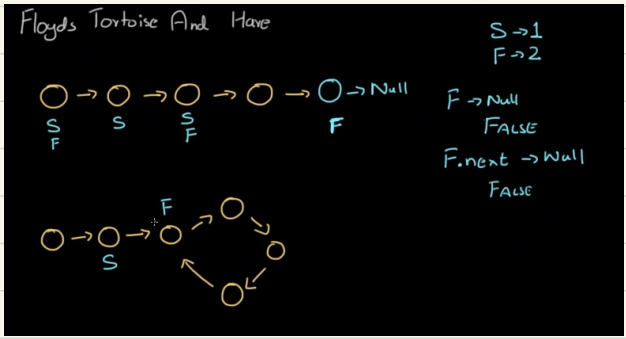
Idea:

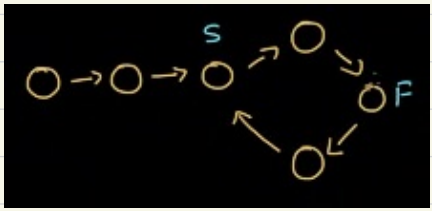
- The strategy we can use is to reverse the second half of the Linked List in-place (modifying the Linked List structure)
- comparing it with the first half.
- Afterwards, we should re-reverse the second half and put the list back together.

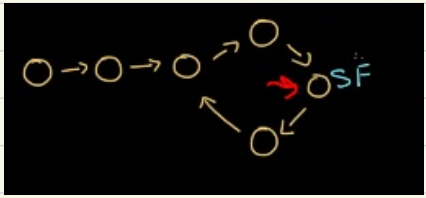
iii) Linked list Cycle (**Floyd判圈算法**)

Beware:How to handle the edge case (head **- index 0 as the start/entrance** of the cycle) ?

- 0 -> 1 -> 0 ([0, 1] with index 0 as entrance) **OR**
- 0 -> 0 ([0] with index 0 as entrance)







s = y + x

f = y + x * 2 + 2

f - s = x*2 + 2 - x

= x + 2

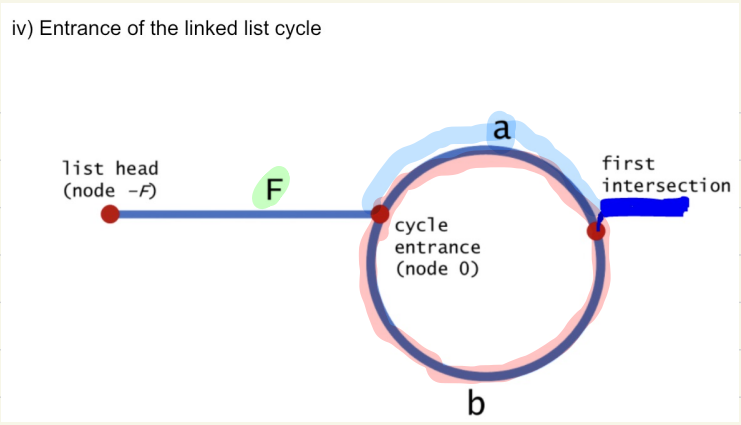
Complexity analysis

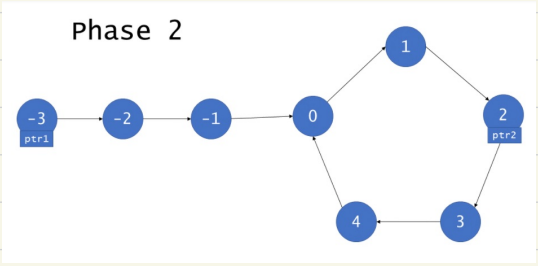
- Time complexity : $O(n)$. Let us denote n as the total number of nodes in the linked list. To analyze its time complexity, we consider the following two cases separately.
  - ***List has no cycle:***
    The fast pointer reaches the end first and the run time depends on the list's length, which is $O(n)$.
  - ***List has a cycle:***
    We break down the movement of the slow pointer into two steps, the non-cyclic part and the cyclic part:
    1. The slow pointer takes "non-cyclic length" steps to enter the cycle. At this point, the fast pointer has already reached the cycle. $\text{Number of iterations} = \text{non-cyclic length} = N$
    2. Both pointers are now in the cycle. Consider two runners running in a cycle - the fast runner moves 2 steps while the slow runner moves 1 steps at a time. Since the speed difference is 1, it takes $\frac{\text{distance between the 2 runners}}{\text{difference of speed}}$ loops for the fast runner to catch up with the slow runner. As the distance is at most "cyclic length K" and the speed difference is 1, we conclude that $\text{Number of iterations} = \text{almost "cyclic length } K\text{"}$.

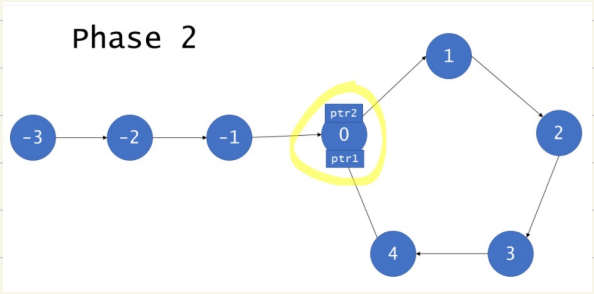
Therefore, the worst case time complexity is $O(N + K)$, which is $O(n)$.

```
public boolean hasCycle(ListNode head) {
        /**
         * Can you solve it using O(1) (i.e. constant) memory?
         */
        if(head==null){
            return false;
        }

        ListNode s = head, f = head;
        while(f.next!=null
                && f.next.next!=null){
            //to handle [1,2] with pos = 0
            //or [1] with pos = 0
            s = s.next;
            f = f.next.next;

            if(s==f){
                return true;
            }
        }

        return false;
    }
```

iv) Entrance of the linked list cycle







We can harness the fact that `hare` moves twice as quickly as `tortoise` to assert that when `hare` and `tortoise` meet at node h, `hare` has traversed twice as many nodes. Using this fact, we deduce the following:

To compute the intersection point, let's note that the hare has traversed twice as many nodes as the tortoise, *i.e.* $2d(\text{tortoise}) = d(\text{hare})$, that means

$2(F + a) = F + nC + a$, where n is some integer.

> Hence the coordinate of the intersection point is $F + a = nC$.

Thus, If given that two pointers below

- p1 at the node head (-3) //0 steps
- p2 at the node a (2) //it implied that it has run a steps

And Both is gonna run F steps to reach at the entrance of the cycle (node 0)

```
/**
        * edge case:
        *  1. null
        *  2. 0 -> null
        *  3. 0 -> 1 -> 0  //pos = 0
        *
        *  non-trivial cases: ( different combinations )
        *  1. -3 -> -2 -> -1 -> 0 -> 1 -> 2 -> 3 -> 4 -> 0 //pos = 0, cycle length = 5, F = 3 , h = 3%5 = 3, a = 5 - h = 2
        *  2. -4 -> -3 -> -2 -> -1 -> 0 -> 1 -> 2 -> 0     //pos = 0, cycle length = 3, F = 4 , h = 4%3 = 1, a = 3 - h = 2
        *  3. -2 -> -1 -> 0 -> 1 -> 0 //pos = 0, cycle length = 2, F = 2, h = 2%2 = 0 , a = 2 - h = move 2 step from 0 (finally at
node 0)
        *  4. -2 -> -1 -> 0 -> 0 //pos = 0, cycle length = 1, F=2 , h=2%1 = 0, a = 1 - 0 = move 1 step from node 0 (finally at
node 0)
        */

            if(head==null){
                return null;
            }

            //detect a cycle
            ListNode s = head, f = head;
            ListNode p1 = head, p2 = null;
            while(f.next!=null
                    && f.next.next!=null){
               s = s.next;
               f = f.next.next;
               if(s==f){
                   p2 = f;
                   break;
               }
            }

            if(p2!=null){ //if there is a cycle and both s & f intersects at a
                while (p1!=p2){
                    p1 = p1.next;
                    p2 = p2.next;
                }
            }

            return p2;
```

In-place Operation

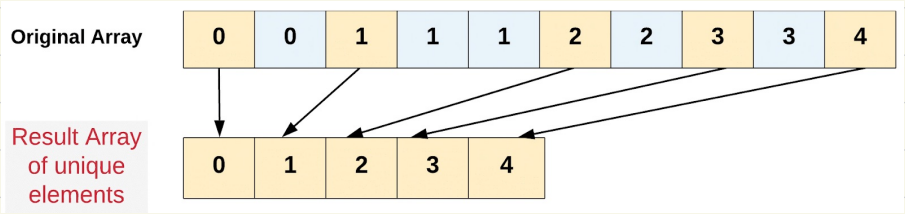
Space = O(1)

1. Array In-place

Given sorted array, convert array to distinct sorted element

① New Copy

One potential problem is that we actually don't know how long the result Array needs to be.
Remember how that must be decided when the Array is created?
do an initial pass, counting the number of unique elements.
Then, we can create the result Array and do a second pass to add the elements into it



② 2 Pointers

1. Read all the elements like we did before, to identify the duplicates. We call this our `readPointer`.
2. Keep track of the next position in the front to write the next unique element we've found. We call this our `writePointer`.

You're quite possibly surprised that this even works. How are we **not overwriting any elements** that we haven't yet looked at?!

- it is impossible for writePointer to ever get ahead of the readPointer.
    - never overwrite a value that we haven't yet read

2. Cyclic Sort space = O(1)

This pattern describes an interesting approach to deal with problems involving
arrays containing numbers in a given range.

input: [1,2,0]，且已知 array 中的 element 就是從 0 ~ array.size()-1
output: [0,1,2]

這個問題，我們可以用任意一個較佳的 sort 演算法在 O(nlogn) 的時間複雜度內完成。不過，既然我們已經知道這個 array 中 element 的範圍，就應該好好利用這個資訊。

以上就是 Cyclic Sort 的基本概念。這時你可能會有個疑問，這樣真的有比 O(nlogn) 快嗎？

答案是有的，因為**每次 swap 都會確保有一個 element 被放對位置，所以最多只需要 `n-1 次 swap`**（經過 n-1 次 swap，第 n 個 element 也會在對的位置，如果你覺得有點不通透，可以從上面的例子觀察得到）；加上有時候不會 swap，最多需要 increment i n 次，所以時間複雜度會是 O(n-1) + O(n)，所以還是 O(n) 的時間複雜度。

How do I identify this pattern?
They will be problems involving (Input) a sorted array with numbers in a given range
If the problem asks you to find the missing/duplicate/smallest number in an sorted/rotated array

Problems featuring cyclic sort pattern:
Find the Missing Number (easy)
Find the Smallest Missing Positive Number (medium)

Note: please compare

- Find All Duplicates in an Array - LeetCode (Cyclic Sort)
- And Find the Duplicate Number - LeetCode (Fast-slow pointers)
  - refer to #287 Find the Duplicate Number
  - refer to Sliding Window & Pointers

Given an integer array `nums` of length `n` where all the integers of `nums` are in the range `[1, n]` and each integer appears once or twice, return *an array of all the integers that appears twice*.

You must write an algorithm that runs in O(n) time and uses ***only constant extra space***.
=>in-place

Example 1:

```
Input: nums = [4,3,2,7,8,2,3,1]
Output: [2,3]
```

Duplicate Numbers

please refer the illustration in good notes

442. Find All Duplicates in an Array ⇒ Cyclic Sort

287. Find The Duplicate Number ⇒ Fast-Slow

Brute Force Code ! # only works for at most twice

space → O(1)

```
class Solution {
    public List<Integer> findDuplicates(int[] nums) {
        List<Integer> ans = new ArrayList<>();

        for (int i = 0; i < nums.length; i++) {
            for (int j = i + 1; j < nums.length; j++) {
                if (nums[j] == nums[i]) {
                    ans.add(nums[i]);
                    break;
                }
                }
        }

        return ans;
    }
}
```

Complexity Analysis

- Time complexity : $\mathcal{O}(n^2)$.

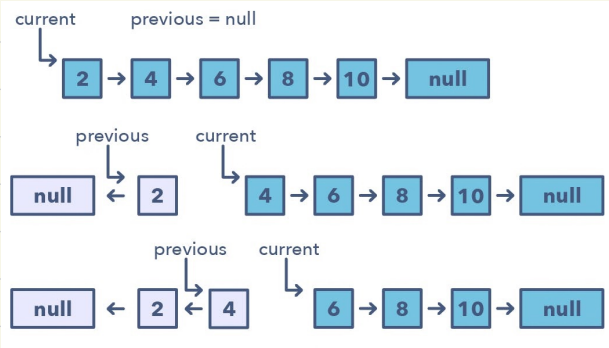
  For each element in the array, we search for another occurrence in the rest of the array. Hence, for the i^{th} element in the array, we might end up looking through all $n - i$ remaining elements in the worst case. So, we can end up going through about n^2 elements in the worst case.

  $n-1+n-2+n-3+....+1+0 = \sum_1^n (n-i) \simeq n^2$

- Space complexity : No extra space required, other than the space for the output list.

3. In-place reversal of Linked List

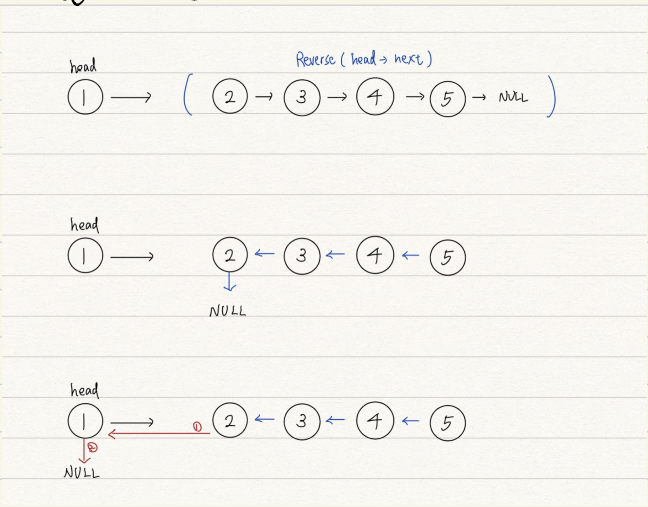
In a lot of problems, you may be asked to reverse the links between a set of nodes of a linked list.
Often, the constraint is that you need to do this in-place,
i.e., using the existing node objects and without using extra memory.
This is where the above mentioned pattern is useful.



Iterative

```
public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while(curr!=null){
            ListNode tempNext = curr.next;
            curr.next = prev;
            prev = curr;
            curr = tempNext;
        }

        return prev;
    }
```

Recursive




```
/**
        * recurrence relation:
        *      f(k):= write(nk.next.next = nk)
        *              And write(nk.next = null)
        *              And f(k+1) //top-down
reverse it
        *
        * Base case:
        *      f(tail->null) = return(tail)  //tail
is the head of the reversed linkedlist
        */
```

```
//if it is empty list or tail
        if(head==null || head.next==null){
            return head;
        }

        //goes to the tail of the list
        ListNode tail = reverseList(head.next);

        //then backtrack to reverse list
        head.next.next = head;  //set next node's pointer to itself
        head.next = null;       //unset its next reference (unbind to the
forward node)

        //return tail as the head of reversed list
        return tail;
```

"Simple" Path Problem

SSP

Generic S.P. Algorithm

Initialize: for $v \in V$: $d[v] \leftarrow \infty$, $\Pi[v] \leftarrow$ NIL

$d[S] \leftarrow 0$

Main: repeat

select edge (u, v) [somehow]

"Relax" edge (u, v):

if $d[v] > d[u] + w(u, v)$:

$d[v] \leftarrow d[u] + w(u, v)$

$\pi[v] \leftarrow u$

until you can't relax any more edges or you're tired or ...

Complexity:

Termination: Algorithm will continually relax edges when there are negative cycles present.

Definition of "Simple"

A path is called simple if it does **not have** any **repeated** vertices; the length of a path may either be measured **by its number of edges, or (in weighted graphs)** by the sum of the weights of its edges.

Bellman-Ford

Bellman-Ford(G,W,s)

Initialize ()
for $i = 1$ to $|V| - 1$
  for each edge $(u, v) \in E$:
    Relax(u, v)
for each edge $(u, v) \in E$
  do if $d[v] > d[u] + w(u, v)$
    then report a negative-weight cycle exists

At the end, $d[v] = \delta(s, v)$, if no negative-weight cycles.

Theorem:
If $G = (V, E)$ contains no negative weight cycles, then after Bellman-Ford executes $d[v] = \delta(s, v)$ for all $v \in V$.

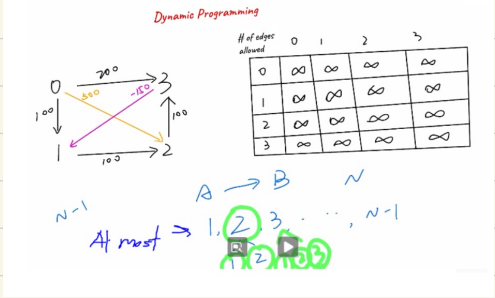
* 3 Types of Relaxation

↳ iterate $k = |V| - 1$ times
regardless of the order of edges.

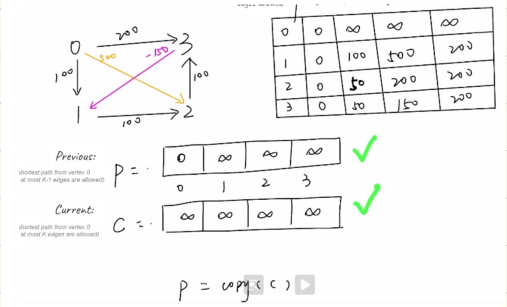
↳ iterate till no more relaxing

↳ using Queue to keep track of which to relax next. (SPFA)

vertex Zero As starting point



*But **using 1D array** here becos **only** need the **previous value(k-1)** but no more to calculate k's result.*

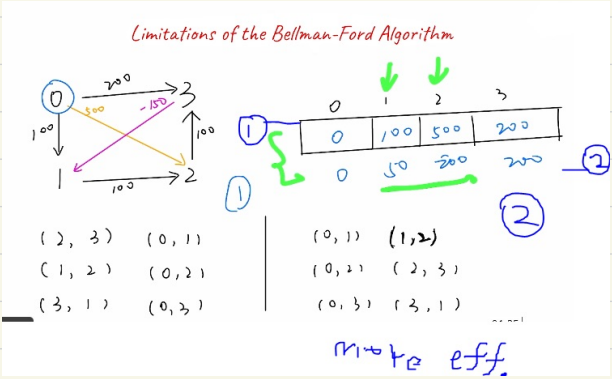


Limitation of Bellman Ford

The improvement is that for a graph without cycles with negative weights, after relaxing each edge N-1 times, we can get the minimum distance from the starting vertex to all other vertices. However, there could be unnecessary computation when relaxing all edges N-1 times, resulting in suboptimal time complexity in some cases.

=>Comparison with different orders for iterations

(second ordering has less iteration to find the shortest path…….)



The algorithm consists of $n-1$ rounds, and on each round the algorithm goes through all edges of the graph and tries to reduce the distances. The algorithm constructs an array `distance` that will contain the distances from x to all nodes of the graph. The constant `INF` denotes an infinite distance.

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

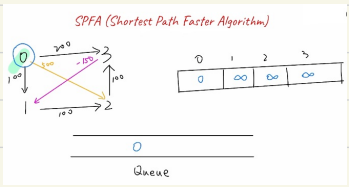
The time complexity of the algorithm is $O(nm)$, because the algorithm consists of $n-1$ rounds and iterates through all m edges during a round. If there are no negative cycles in the graph, all distances are final after $n-1$ rounds, because each shortest path can contain at most $n-1$ edges.

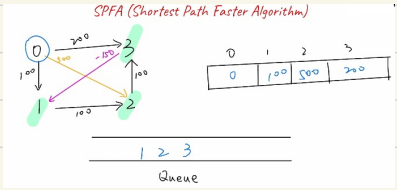
=>("the Shortest Path Faster Algorithm")SPFA algorithm

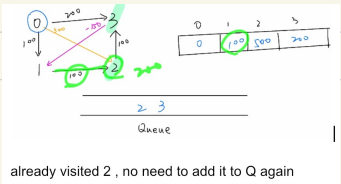
To address the limitations, we introduce an improved variation of the Bellman-Ford algorithm by using a queue.

Instead of choosing among any untraversed edges, as one does by using the "Bellman-Ford" algorithm, the "SPFA" Algorithm uses a "queue" to maintain the next starting vertex of the edge to be traversed.

- Only when the shortest distance **of a vertex is relaxed (new min value is set)** and that **the vertex** is **not** in the "queue"(re-visited is allowed but popped before newly relaxed), we **add** the vertex to the queue.
- We iterate the process until the queue is empty. **At this point**, we have **calculated** the **minimum** distance **from** the given **vertex to any** vertices.

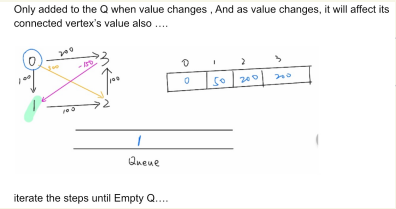






already visited 2 , no need to add it to Q again

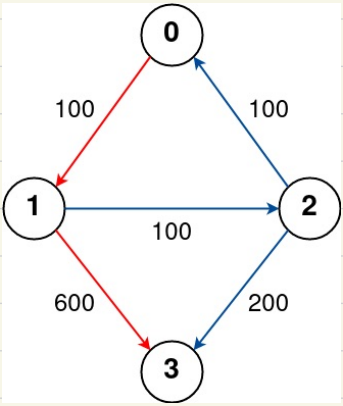
Only added to the Q when value changes , And as value changes, it will affect its connected vertex's value also ....



iterate the steps until Empty Q....

There are n cities connected by some number of flights.

* You are given an array flights where flights[i] = [fromi, toi, pricei]
* indicates that there is a flight from city fromi to city toi with cost pricei.
*
* You are also given three integers src, dst, and k,
* return the cheapest price from src to dst
* – with at most k stops (k stops exclude src, dst nodes)
* If there is no such route, return –




```
Input: n = 4, flights = [[0,1,100],[1,2,100],
[2,0,100],[1,3,600],[2,3,200]], src = 0, dst
= 3, k = 1
Output: 700
Explanation:
The graph is shown above.
The optimal path with at most 1 stop from
city 0 to 3 is marked in red and has cost 100
+ 600 = 700.
Note that the path through cities [0,1,2,3]
is cheaper but is invalid because it uses 2
stops
```

/*

0. At most K stops = At most k stops between src and dst
   = At most [k + 2 (src&dst) – 1] Edges
   = At most k + 1 edges
   = At most X edges
   = O(k)

1. Recursion Relations:
   f(v,x) = min{f(v, x–1), f(u, x–1) + w(u,v) | for all u to v}
2. Base case:
   f(src, 0) = 0
3. Our Goal:
   f(dst, X=k+1)

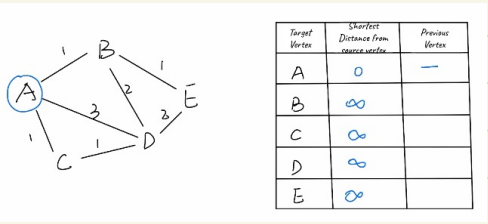
*/

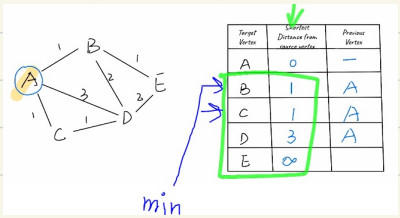
```
        int[][] dp = new int[n][k+2]; //including zero edges
        for(int[] arr:dp) {
            Arrays.fill(arr, Integer.MAX_VALUE);
        }
        dp[src][0] = 0;
        for(int i=1;i<k+2;i++){      //O(k) or in worse case (V)
            for(int j=0;j<n;j++){    //O(V)
                dp[j][i] = dp[j][i-1];
            }

            for(int[] flight:flights){  //O(E)
                final int prev = flight[0];
                final int next = flight[1];
                final int price = flight[2];

                if(dp[prev][i-1]!=Integer.MAX_VALUE){
                    dp[next][i] = Math.min(dp[next][i], dp[prev][i-1] +
price);
                }
            }
        }


        //O(K)*[O(V) + O(E)]
        //for max input k = O(V), O(V^2) + O(V*E)
        return dp[dst][k+1]!=Integer.MAX_VALUE ? dp[dst][k+1] : -1;
```

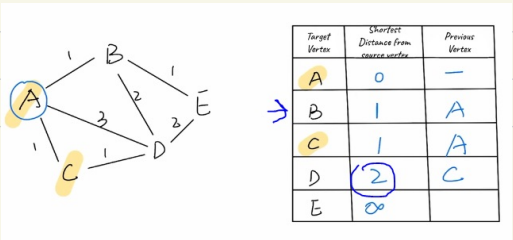
Dijkstra – Sample Question

```
/**
* You are given a network of n nodes,
*  - labeled from 1 to n.
* You are also given times,
*  - a list of travel times as directed edges times[i] = (ui, vi, wi),
*  - where ui is the source node,
*         vi is the target node,
*      and wi is the time it takes
* for a signal to travel from source to target.
*
* We will send a signal from a given node k.
*  - Return the time it takes for all the n nodes
*      - to receive the signal.
* If it is impossible for all the n nodes to receive the signal,
*  - return -1.
*
* Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
* Output: 2
* Example 2

/**
 * src = k
 * vertexIndex: i (1...n)
 * path distance(not yet confirmed that it is shortest) from s to i: dist(i)
 * Graph:
 *  - from j to all connected i with cost wji
 *  - Map{j,List<[i, wji]> or vertex[j] = List<[i, wji]>
 * Min Heap:
 *  - connected vertices{i, dist(i)}
 *  - for greedy choice
 * Recursion Relations:
 *  - dist[i] = min{heap} + w(j,i) | given vertex(j) of min{heap} is the locally optimal choice
 *  - And given that w(j,i) >= 0
 * Visited And/Or Shortest Distance(sd):
 *  - boolean[] visited
 *      - visited[i] = true
 *  - sd[i] = min{heap} | given vertex(i) of min{heap} is the global optimal value
 *      - if(sd[i]!=Integer.MAX_VALUE)
 * Our goal:
 *   Max{sd}
 */
```

```
int[] sd = new int[n+1];
  Arrays.fill(sd, Integer.MAX_VALUE);     //O(V)

  Queue<int[]> pathDistQ = new PriorityQueue<>(Comparator.comparingInt(d -> d[1]));
  pathDistQ.add(new int[]{k, 0});
  while(!pathDistQ.isEmpty()){
     int[] pathDist = pathDistQ.poll();
     int vertex = pathDist[0];
     int dist = pathDist[1];

     if(sd[vertex]!=Integer.MAX_VALUE){ //if visited     //O(V)
        continue;
     }
     sd[vertex] = dist;

     if(vertices.containsKey(vertex)) {
        List<int[]> edges = vertices.get(vertex);
        for(int[] edge: edges){     //O(E)
           int to = edge[0];
           int cost = edge[1];
           if(sd[to]!=Integer.MAX_VALUE) { //if visited
              continue;
           }

           pathDistQ.add(new int[]{to, sd[vertex] + cost});    //O(logE)
        }
     }
  }
```







MON TUE WED THU FRI SAT SUN | DATE / / | NO | SUBJECT

2. Shortest Path — single Source

↳ Dijkstra

- weighted directed graph with non-negative weights.
- Each step selects the "minimum weight" from the currently reached vertices to find the shortest path to other vertices,
  - * current shortest distance from src.

Time: $O(E \log V)$

space: $O(V)$

e.g Network Delay Time
#743

↳ Bellman-ford $O(V \cdot E)$, including
- weighted directed graph ~~with~~ negative weight (but still non-negative cycle)

- A graph with no negative cycles and 'N' vertices, shortest path between any 2 vertices has at most N-1 edges.

e.g cheapest flights within K stops
#787

Double A

For a general weighted graph
single source shortest distances in O(VE) time using Bellman–Ford Algorithm.

For a graph with no negative weights
single source shortest distances in O(E + VLogV) time using Dijkstra's algorithm.

Can we do even better for Directed Acyclic Graph (DAG)?

For DAGs **(only DAG has top order but BF can be applied on non DAG)**
single source shortest distances in O(V+E) time
The idea is to use Topological Sorting

Restrictions		SSSP Algorithm	
Graph	Weights	Name	Running Time $O(\cdot)$
General	Unweighted	BFS	$\|V\| + \|E\|$
DAG	Any	DAG Relaxation	$\|V\| + \|E\|$
General	Any	Bellman-Ford	$\|V\| \cdot \|E\|$
General	Non-negative	Dijkstra	$\|V\| \log \|V\| + \|E\|$

Non-negative Cycle:

↳ BF: • From Graph perspective, we don't need to care about (topological) ordering of relaxing

• From Graph duplication (conversion to DAG) perspective, we do it (relexation) in a topological order

↳ DP (DAG Relaxation):

Doesn't support Cycle

Non-Negative weight

↳ bijkstra:

- Don't need to care about top order
- Non-bruteforce

↳ DP: Doesn't support cycle

DAG with any (+/-) weights

↳ DP: do it in top order

- longest / shortest simple Path

Longest Path in DAGs

If *G* is a DAG, because there is no cycles, the problem of finding longest simple path can be *solved using topological sort in linear time*. The solution is similar to the solution for finding the shortest distance in DAGs except that we take the maximum value as we relax the distances.

Negative Cycle

↳ BF:

- Simple path cannot be defined in d[v]
- simple path does exist but just NP-hard

To understand it better, suppose there is a negative cycle in *G*. In this case none of our famous algorithms can find a shortest path ***because it doesn't exit***. *However, there is still a shortest* ***simple*** *path in the graph where no vertices repeat.*

- Finding that shortest simple path is NP-hard!

- Such cycle can be detected by BF

Longest "Simple" Path Problem

The Longest Distance Problem

The sister problem to finding the shortest path is to find the longest path. But first notice that there is a huge confusion when we talk about the longest path:

> The longest path problem **commonly** means finding the longest *simple* path. The shortest path problem (as discussed above), however, focuses on finding the shortest (*simple* or *non-simple*) path.

So, in the literature even *when people talk about finding the longest path*, they usually mean finding the longest **simple** path.

↳ By reduction to short simple path Problem (- G)

- Cycle Exists
  - All Positive weight → Negative Cycle exists
    - ↳ NP hard
  - All Negative weight → Non-Negative Cycle only
    - ↳ BF-Defined
- Negative & Positive weight (Mixed)

- Positive cycle exists → Negative Cycle exists
  - ↳ NP-hard
- Negative Cycle only → Non-Negative Cycle only
  - ↳ BF-defined

⇒ Negation: P $\xrightarrow{-G}$ NP-hard

- DAG $\xrightarrow{-G}$ DAG

What you need to know about the longest simple path problem

- Finding the longest simple path in general is NP-Hard. This can easily be shown by reducing from the Hamiltonian Cycle problem.
- It follows that finding the longest simple path *in the presence of* ***positive*** *cycles in G* is NP-hard.
- If there is no positive cycles in *G*, the longest simple path problem can be solved in polynomial time by running one of the above shortest path algorithms on *-G*.

「相信你自己是可以實現夢想的人，
你就會吸引到自己需要的一切。」

當你充滿正能量時，一切都會轉變，
宇宙會幫你得到想要的東西。

當你不再依賴任何人或事物的時候，
周圍的人和事反而會自然而然地幫助
你，因為你不再是一個匱乏的人了。

不用加油，很開心。

不用加油，很愉快。
不用加油，就是刻畫自己的時間。
不用加油，很幸福。
不用加油，對身體很好。
不用加油，對心也很好。
不用加油，很健康。
不用加油，就不用爭鬥。
不用加油，很環保。
不用加油，不會傷害任何人。
不用加油，是真正的「和平」。
不用加油，繼續去愛這個地球。
宇宙，不用加油。
我，不用加油。