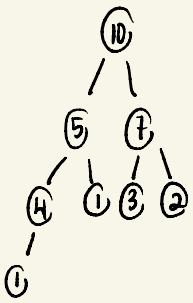
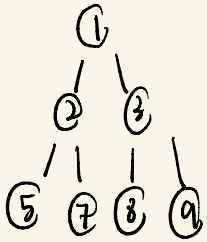
FIT2004 Complete Summary

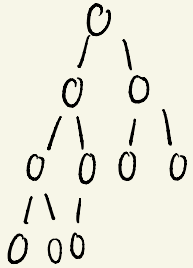
max heap



min heap



complete heap

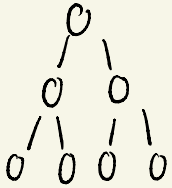


fill top → bottom,

✓ left → right

✓ not null / empty

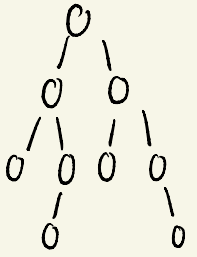
full heap



fill top → bottom,

✓ left → right

incomplete heap



fill top → bottom,

X left → right

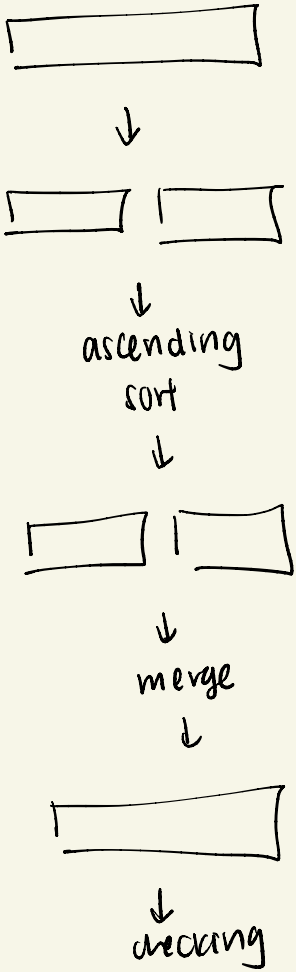
✓ null / empty

merge sort

↳ divide to 2 parts
  ↳ or more smaller parts

↳ solve and/or sort the smaller parts
  ↳ eg. ascending / descending

↳ once done, merge 2 array tgt



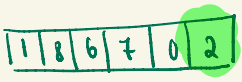
quicksort

↳ divide & conquer

① choose any elem as pivot

② divide main array to smaller arrays

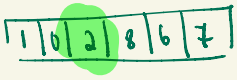
③ compare and sort all elem with pivot
- ↳ left : smaller
- ↳ right : bigger

eg. 

← original array

choose pivot
- ↳ eg. rightmost elem : 2

① compare all elem with pivot & place pivot within array so that
- right elem > pivot
- left elem < pivot



② choose pivot for left & right and continue to arrange

③ choose pivot for left & right and continue to arrange

★ SORTED

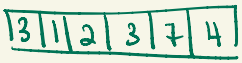
counting sort

① find max value in array

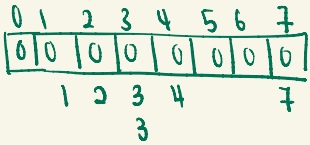
② initialize and create new array of size max+1 with all 0

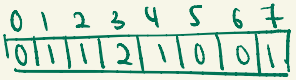
③ iterate through original array while increment the count of each element in count array
- ↳ original array no change yet

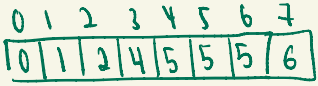
④ find index of each elem of original array in count array
- ↳ place array elem in correct position
- ↳ decrement count in count array

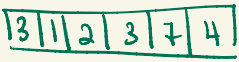
eg. 

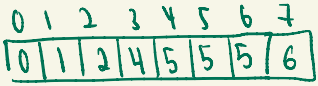
max : 7

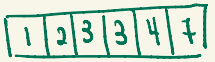






original 

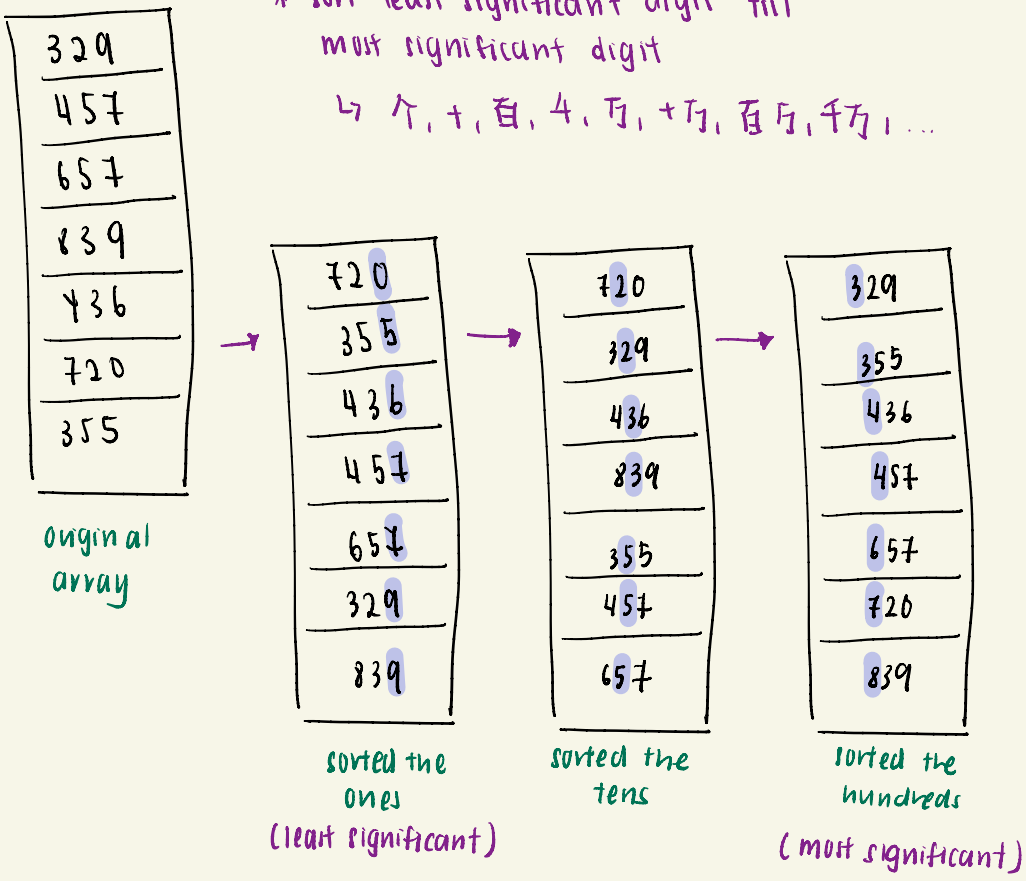
count (cumulative) 

output 

radix sort

- numbers in array : 329, 457, 657, 839, 436, 720, 355

* sort least significant digit till most significant digit
  ↳ 个, 十, 百, 千, 万, 十万, 百万, 千万, ...



quickselect

↳ selection algorithm

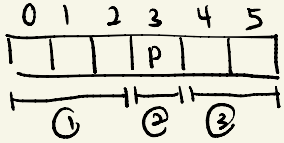
↳ find k-th smallest element from unordered list

↳ @ see quick sort (sorting algorithm)

① select a pivot
   ↳ random / first / last / median

② sort elements to left right
   ↳ compare each elem with pivot

③ k value if > pivot index, search right
   k value if < pivot index, search left
   k value if = pivot index, found, return



① if k = 0/1/2
② if k = 3
③ if k = 4/5

eg. array = | 7 | 10 | 1 | 4 | 20 | 15 |

① k = 3
   ↳ output = 7

② k = 4
   ↳ output = 10

② k = 6
   ↳ output = 20

median of medians

↳ approximate median selection algorithm

↳ divide and conquer

↳ find kth smallest num in unsorted array

① divide array to smaller array

② find median in the smaller array

③ put all median to a list, find median of the list

↳ median of the list = pivot

** median of list = pivot

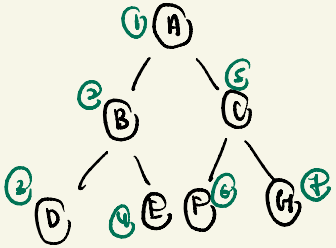
④ compare array elem with pivot } quickselect

↳ < pivot = left

↳ > pivot = right

Depth - First Search (DFS)

- start : root node
- movement : adjacent unmarked node
- aim : visit and mark unmarked nodes untill no unmarked nodes exist
- output : print nodes in path

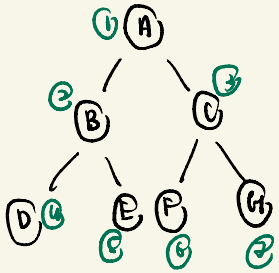


DFS path : ABDECFG

" left to right "

category → visited / discovered
category → not visited / discovered

breadth - first search (BFS)



"layer by layer, up to down"

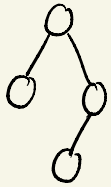
BFS : ABCDEFG

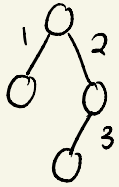
category → visited / discovered
→ not visited / discovered

- start : root node
- movement : unmarked node layered
- aim : visit and mark unmarked nodes untill no unmarked nodes exist
- output : print nodes in path

graph

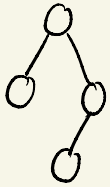
unweighted vs weighted

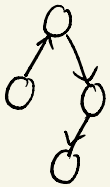


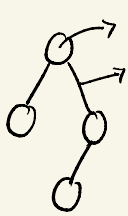


* cycle in graphs:
  begins and ends in the same vertex

undirected vs directed







* with distance & node, can construct the sequence of shortest path from start → end

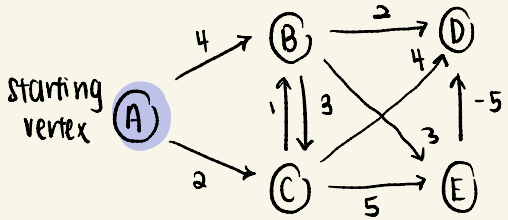
shortest path

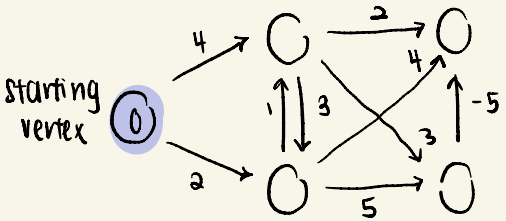
2 array
- → distance
- → vertex / node that proceeds to another vertex / node of a specific distance saved in the other distance array

Bellman-Ford algorithm

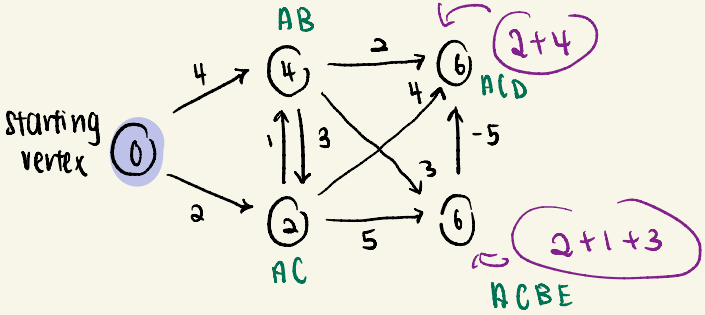
- solve single-source shortest path in weighted graph
- calculate shortest path in bottom-up manner
- ✓ negative weights, check negative weight cycles
- goes through each edge in every iteration

① weighted graph, choose a starting vertex

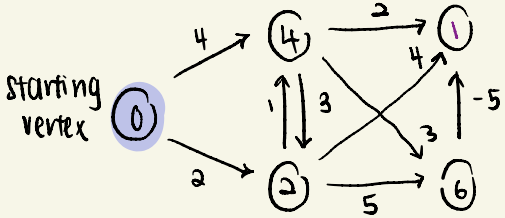




② visit each edge



③ adjustments



ACBED

$= 2+1+3+(-5)$

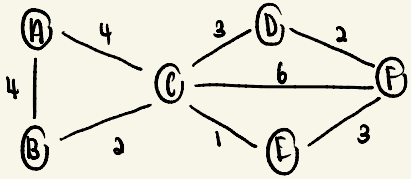
$= 6-5$

$= 1$ #

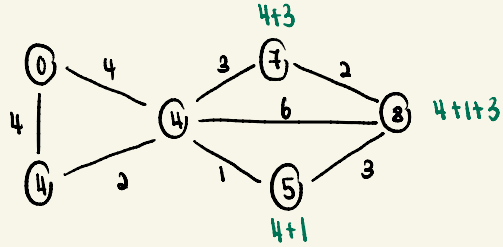
④ check negative cycle

Dijkstra's algorithm

- X negative weight graph
- shortest path in weighted graph from source to target
- √ use weight of edges
- √ greedy algorithm



starting vertex = A



* repeatedly visit vertex until all vertex are visited

* complexity
  1. time → O(E log(V))
  2. space → O(V)

Floyd-Warshall algorithm

X negative weighted, negative cycle graph

V both directed, undirected

- shortest path in weighted graph
- dynamic programming approach

complexity

① time : $O(n^3)$

② space : $O(n^2)$

matrix method: empty matrix with V x V size

* V = vertex

fill in current version of graph to matrix

iterate through v_1 to v_n with other v in the mid to find paths

↳ all possible paths with nth number of v in the mid for v_1 to regin v_n

kahn's algorithm & topological sorting

- ordering vertices of directed graph
- directed edge (u,v), u vertex comes before v vertex
- helps determine the order
- possible for directed acyclic graphs (DAG)
  - ↳ x directed cycles
    - ↳ if √ cycle, x topological ordering
      - ↳ circular dependency

↳ iteration

① empty ordering array

② calculate incoming edges to specific vertex

③ queue all vertex with 0 incoming edges (start vertex)

④ while queue != 0,

```
deque vertex v
emptyarray.append(v)
for i in v.neighbour:
    i.incomingEdge -= 1
    if i.incomingEdge == 0:
        queue.enqueue(i)
```

* ordering array = valid topological order

** if algorithm not finish, means there is a cycle

*** time complexity = O(V+E)

DFS

① empty ordering array

② x incoming edges = start
  - ↳ mark 0, visited

③ adjacent vertex visited + add current vertex to array

④ continue until all vertex is visited

* ordering list / array = valid topological ordering

** if still have unvisited vertex = cycle present

graph connectivity

pg 149

① incremental connectivity problem

↳ connect(u,v) & link(u,v)

↳ check connections

↳ add edge

prim's algorithm

↳ greedy algorithm

↳ find minimum spanning tree (MST) of weighted, undirected graph
  ↳ connect all vertices with min weight

① choose a starting point

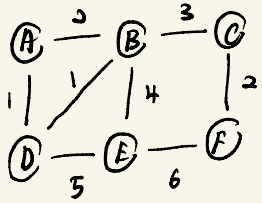
② branch out to find unvisited & unconnected vertex
  ↳ connect to MST

③ until all vertices is connected, it is not complete

④ once completed, the MST is min weight edge chosen to connect unvisited node to visited node in MST

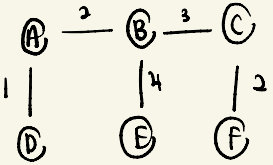
** 100% min spanning tree
  ↳ lowest weight always chosen if can
  ↳ connect all vertex to MST

eg.



starting vertex = A

MST:



weight: 2 + 3 + 1 + 4 + 2
= 12 #

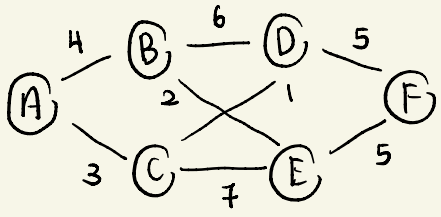
kruskal's algorithm

↳ greedy algorithm

↳ find minimum spanning tree (MST) of weighted, undirected graph
  ↳ connect all vertices with min weight

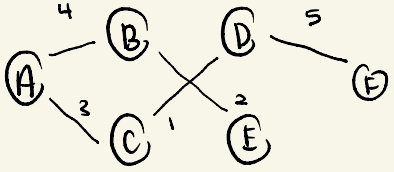
1. sort all edges in non-decreasing order of weight
2. empty array (store MST)
3. iterate through sorted edges smallest → largest
4. if add to MST = ✓ cycle, don't add
   else add
5. repeat until all vertex is visited and have (V-1) edges

KRUSKAL VS PRIM

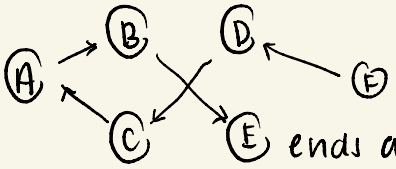


sort by order :

CD – 1
BE – 2
AC – 3
AB – 4
DF – 5
EF – 5
BD – 6



check for cycle

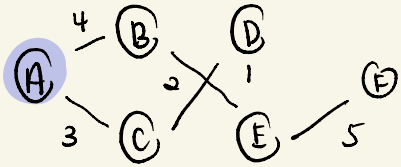


ends at E or F, no cycle exist

total weight : 1 + 2 + 3 + 4 + 5

= 15 #

choose starting point (A)



✓ all vertices visited

✓ shortest edge chosen

weight

= 1 + 2 + 3 + 4 + 5

= 15 #

same !

DYNAMIC PROGRAMMING

√ overlapping subproblems
- ↳ solution to subproblems needed multiple times
  - ↳ store in memorization table
    - ↳ access & use immediately
    - ↳ avoid repeated / redundant calculation
    - ↳ efficiently solve larger problems

* find base case of subproblem
- ↳ solve & store solution in matrix/table
- ↳ eg. fibonacci
  - ↳ solution = sum of 2 previous numbers
  - ↳ computed, stored in table
    - ↳ retrieve for next calculation with index
    - ↳ X count / calculate again
- ↳ eg. knapsack
  - ↳ select item with certain values & weight to maximise total value within limit
    - ↳ matrix to store info (item x weight)
      - ↳ filled iteratively on possible weight
      - ↳ get max value within limit from possible values
        - ↳ previous item & weight

SORTING ALGORITHM

selection sort	$O(1)$	$O(n^2)$	$O(n^2)$
insertion sort	$O(1)$	$O(n)$	$O(n^2)$
merge sort	$O(n)$	$O(n \log n)$	$O(n \log n)$
heap sort	$O(1)$	$O(n \log n)$	$O(n \log n)$
quick sort	$O(\log n)$	$O(n \log n)$	$O(n^2)$
counting sort	$O(k)$	$O(n+k)$	$O(n+k)$
radix sort	$O(n+k)$	$O(nk)$	$O(nk)$

time complexity : Best, WORST space complexity : AUX

SELECTION ALGORITHM

quick select

median of medians

} time and space complexity depends on pivot choice and implementation.

SORTING ALGORITHM

selection sort ----- not stable ----- in-place
insertion sort ---------- stable ----- in-place
merge sort ---------- stable ---- not in-place
heap sort ----- not stable ----- in-place
quick sort ------ not stable --- in-place
counting sort ------ stable --- not in-place
radix sort ------ stable --- not in-place

IMPORTANT POINTS

↳ heap sort's sorting is in-place
↳ quick sort's sorting is in-place
BUT pivot choice will affect stability

TOP-DOWN & BOTTOM-UP DYNAMIC PROGRAMMING

TOP-down dynamic programming

- "memorization"
  - break big problems to small problems to be solved recursively
  - check if the problem is solved b4 recomputation
    - solved = obtain solution from memory
    - not solved = computation to solve and store in memory
  - avoid redundant computation

BOTTOM-up dynamic programming

- "tabulation"
  - solve small problems then big ones, solution store in memory
    - check if the problem is solved b4 recomputation
      - solved = obtain solution from memory
      - not solved = computation to solve and store in memory
    - iteration

HASHING & HASHTABLES

Hashtables (Hash map)
- efficient storage
- efficient retrieval of key-value pairs

Hashing
- mapping key to specific array using hash function
- input : key
  output : hash code
  ↳ hashtable index
- control hashtable size
  = control memory usage

Open addressing (probing)
- collision resolution strategy
- search alternate empty space present in hashtable to store key
- methods :
  - linear probing
  - quadratic probing
  - double hashing

Chaining
- collision resolution strategy
- maintain link list at each position in the hashtable
  ↳ same hash code
  = store in link list in same hash table position

hash function

- good ? minimise collision probability
- uniformly distribute keys across a range of hash values
  - ↳ bad if keys concentrated within small range of hash values
- efficiently handles collision if collisions has occurred

hashing integers

- convert interger keys to indices for storage in hashtable using hash function
- method : ① divisional
  ② multiplicative

① divisional
- modulo of key with table size

② multiplicative
- constant 'A' to compute fraitional part of 'k*A' while scaling by 'm'

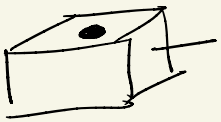
hashing strings

- convert string keys to indices for storage in hashtable using hash function
- method : ① polynomial

① polynomial
- utilize Horner's method to evaluate efficiency of polynomial hashing
- constant time calculation of adjacent substring hashing

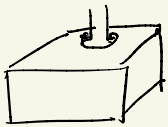
Universal hashing

- approximate totally random hashing while still being computationally efficient
- select random hash functions from universal family of hash functions
- introduce randomness into hash function while being efficient and practical
  - ↳ ensure lower probability that 2 distinct keys hashing will be hashed to the same slot in the hashtable
- helps distribute keys evenly across hashtable



a lot of hash functions

randomly choose 1 hash function



random hash function to be used for hashing 'A'

BALANCED BINARY SEARCH TREES

AVL Trees

- height of left & right subtrees of any node differ by at most one
- remain balanced after insertion & deletion
  - ↳ maintain efficient search, insertion, deletion operation

Definitions

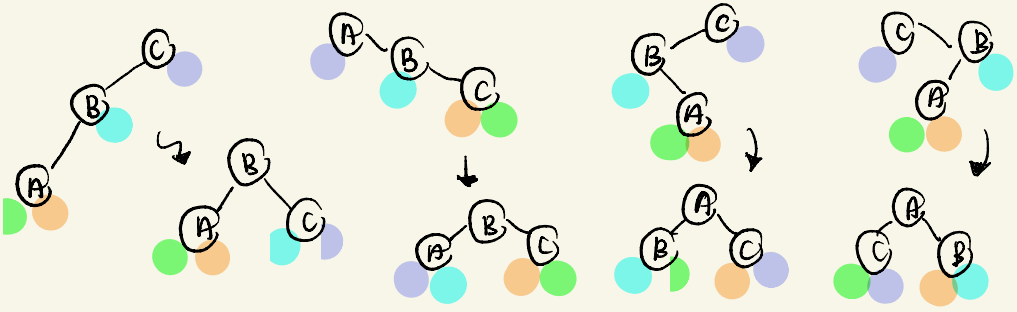
- height : length of longest path
- empty tree height = -1
- left subtree − right subtree = balance factor

absolute balance factor of AVL <= 1

rebalancing

- rotation when insertion / deletion causes difference in node / leaf

① L-L ② R-R ③ R-L ④ L-R



NETWORK FLOW

Finding optimal flow of certain quantity from source node to sink node while taking into consideration on edge's capacity constraint

Ford-fulkerson algorithm

- find maximum flow in a flow network
- repeatedly find paths from source to target while increasing flow

Residual network

- remaining capacity of edges in flow network after flow was pushed through
- check if need increase or decrease flow

augmenting path

↳ path from source to target in residual network where all edges' remaining capacity > 0

↳ gradually increase overall flow

dfs

↳ find augmenting path

↳ explore residual network until reach sink

min-cut max-flow theorum

↳ min flow = min cut capacity

PREFIX TRIES

- retrieval tree
  - ↳ store strings
  - ↳ organised like graph
- 1 root, n child node
  - ↳ weight stored in node
  - ↳ end with $ terminal for each string
- methods:
  1. array
  2. AVL
  3. hash tables

SUFFIX TREES

reduce space complexity

- efficiently store all suffix of given string
- pattern matching problem
  - ↳ find pattern occurrences
  - ↳ eg. text string, T
    pattern, P
    Q: find all occurrence of P as a substring of T

solution

- insert all suffix of text string into prefix tree.
- find / check pattern is a prefix in the tree

SUFFIX ARRAY

- efficiently store and process suffixes of a given string

↓

store starting positions of all the suffix of a given string in order → lexicographical order

The binary search on the suffix array is used to locate the range where the pattern might exist. At each iteration, the middle index "mid" is calculated and the suffix starting at SA[mid] is compared to the pattern P. The position of the first occurrence is stored in the variable "begin". The last occurrence is found at positions SA[begin] through SA[end].

binary search is used to find first and last occurrance of pattern in text

time complexity : $O(m \log(n))$