Algorithms

DATE: 31/03/2022

v = (key, value)

PRIORITY QUEUES / HEAPS

- ADD (H, v) takes O(logn) time;
- FINDMIN (H) = O(1) // ;

  // returns one item of the heap with smallest key

- REMOVE (H, i) takes O(logn) time;

  // removes item in position i

- EXTRACTMIN (H) takes O(logn) time;

  // removes, and returns, an item with smallest key (that is, the item in position 1)

```
v = FINDMIN(H)
REMOVE(H,0)
RETURN v
```

- REMOVE V(H, value) takes O(logn) time;

  // removes the item in the heap having value equal to "value" (THIS WORKS ONLY IF ALL VALUES IN THE HEAP ARE DISTINCT)

- UPDATEKEY (H, value, newkey) takes O(logn) time;

  // changes the key of the item in the heap having value "value" to "newkey"

  (THIS WORKS ONLY IF ALL THE VALUES IN THE HEAP ARE DISTINCT)

For these we need a position vector.

~~(key, value) = H[POSITION[value]]~~ # There's no reason to know the old key.

REMOVE V(H, value)

ADD (H, (newkey, value))

When we remove a value, we also remove the key associated to that value.

Heapify up and down indirecty update the position vector.

With HEAPSORT we can sort an array of n elements in $O(n \log n)$ time.

Best time complexity to sort an array (lower bound).

By using heaps on Dijkstra, we can make it run in almost linear time.

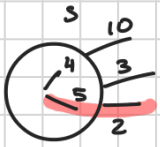
DIJKSTRA'S ALGORITHM (G(V,E), w, s)

```
// V = {0, 1, 2, ..., N-1}                                          (1)
O(N)     INITIALIZE A HEAP OF SIZE N
O(logN)  H.ADD((0, s))   // (KEY, VALUE) WITH KEY = 0 AND VALUE = s
O(N)     d = [NONE] * N  # d: distances.
O(1)     d[s] = 0  # The shortest path at the beginning.

// IN THE END, d[i] is going to contain the length
// of a shortest path from s to i

FOR EACH OUT-NEIGHBOR u of s:   # Once we're adding the source to the heap, we also have to add the neighbours of that source.
    ADD (H, (w(s,u), s))
    # Adding to the heap the weight of the edge s,u

FOR i = 1 TO N-1   (we start from 2 because we already started from 1 node (s))
```

(2) $O(N \log N)$

```
O(logN)  NEXT = EXTRACTMIN(H)
O(1)     v = NEXT.value
O(1)     d[v] = NEXT.key
```

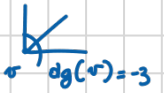
THESE ARE RUN FOR N-1 TIMES



(3) $O(deg(v) \log N)$

```
    FOR EACH OUT-NEIGHBOR u OF v:
        IF d[u] = NONE:                                   O(1)
            // we haven't yet selected/visited "u"
            dist = d[v] + w(v, u)                         O(1)   (dist: distance)
            IF H[POSITION[u]].KEY > dist:                 O(1)
                UPDATEKEY(H, u, dist)                     O(log N)
RETURN d
```

THIS IS RUN FOR degree of v (deg(v)) times



deg - nº nodes/neighbours

The degree of a node is the number of neighbours of a node.

Quicksort = N Log n
Bubblesort = n^2
Heapsort = ?

Almost linear, except for the "logN" factor.

The sum of all the out-neighbors of the nodes v.

TOTAL RUNTIME $= O(N) + O(N \log N) + \sum_{v \in V} O(\deg(v) \log N) =$

$= O(N \log N) + \left(\sum_{v \in V} \deg(v)\right) \cdot O(\log N) =$

$= O(N \log N) + 2M \cdot O(\log N)$

$= O((N+M) \log N)$

E is a subset of all the pairs of nodes.

M = number of edges.

$N+M$ = nodes of the graph + edges.

undirected

L: IF $G(V,E)$ is an undirected graph, then $\sum_{v \in V} \deg(V) = 2|E|$ # If we sum-up all the degrees of nodes in a graph, then the total is gonna be twice the number of edges.

P: Recall that $E \subseteq \{\{(u,v)\} \mid u, v \in V, \text{ and } u \neq v\}$

Moreover, the set of neighbours $N(v)$ of "v" is equal to $N(v) = \{u \mid \exists \{u,v\} \in E\}$.

And $\deg(v) = |N(v)|$

Let us also define the set of edges.

Incident on v, $I(v) = \{\{u,v\} \mid v \in \{u,v\} \text{ AND } \{w,v\} \in E\}$

set of edges

12, 15
21, 24
35
42, 45
51, 53, 54

$N(2) = \{1,4\}$
$I(2) = \{\{1,2\}, \{2,4\}\}$

Then $\deg(v) = |I(v)|$

We proved that we're summing twice the number of edges. 2-4 4-2 POV

$$\sum_{v \in V} \deg(v) = \sum_{v \in V} |I(v)| = \sum_{e \in E} 2 = 2|E| \quad \blacksquare$$

For each node v of V we're summing up all the nodes incident on v.

QUESTION: # Just create 3 variables and count the number of 0, 1, 2.

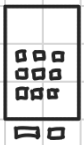
Suppose that V is an array of N elements V[i] is the score that student i got in the algorithm class (scores are "PASS": 1, "INSUFFICIENT": 0, "HONORS": 2)

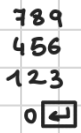
Sort V increasingly as fast as you can. (In less than $N \log n$).

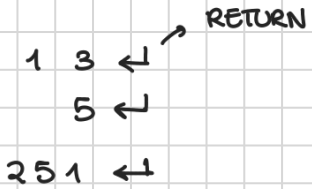
Try greedy approach when they ask about interval scheduling.

The greedy approach for the interval scheduling is different from the interval partitioning.

SALOON







Let's say there is a gunfire and you lose your numbers (keys). how can you keep inserting your entries?

If you have 3 keys, then you can write integers in binary (0,1 FOR AN INTEGERY, ↵ TO SPLIT INTEG.)

If you have only 2 keys, I could write in unary.

3 AND 5

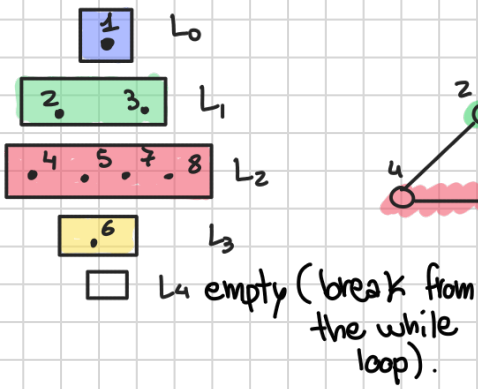
11101111110

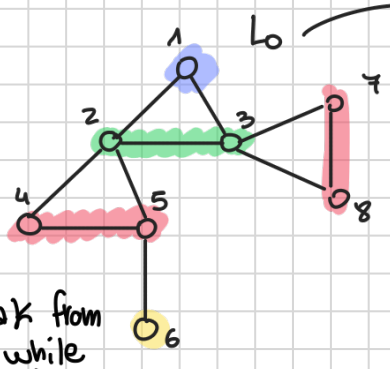
What if you only have the key "zero"?

BFS (BREADTH FIRST SEARCH)

DFS (DEPTH = =)

Two greedy algorithms for visiting graphs.





Layer 0
Layer 1 (Green) reachable from L_0
Layer 2 (Red) = from L_1
Layer 3 (Yellow) = = L_2

If the graph is unweighted, so all the edges have a length of 1, then BFS finds the shortest path.

DFS is used for other purposes, such as finding a strongly connected component (a set of nodes of a directed graph such that you can go from each node to each other node).

BFS (G(V,E), s)

```
S ← {s}      // S is the set of visited nodes
L0 ← {s}     // Li will contain all the nodes at DISTANCE
i ← 0        // i from s
```

L_i = layer

WHILE TRUE:

$L_{i+1} \leftarrow \emptyset$

While there exists some v in L_i and some w in the set of nodes - S such that v,w is an edge. *

WHILE $\exists v \in L_i$ AND $w \in V - S$ S.T. $\{v, w\} \in E$:

$L_{i+1} \leftarrow L_{i+1} \cup \{w\}$ # w is some node that is at distance i+1 from the source.

$O(n+m)$ $S \leftarrow S \cup \{w\}$ # we store the node that we visit.

If L_{i+1} is empty, we're done.

IF $L_{i+1} = \emptyset$: * # While I can find one node in the last layer L_i and a node w that's connected to this last layer, and that I haven't visited yet, then I know that the node w is at distance i+1 from the source. Here we don't look for nodes in a particular way, because they're at the same distance from s.

BREAK

ELSE:

$i \leftarrow i+1$

RETURN S, $(L_0, L_1, \ldots, L_i, \ldots)$ GREEN BECAUSE ITS OPTIONAL IF YOU WANT TO RETURN THIS OR NOT

NOTE: Works better than DIJSKSTRA but works only with unweighted graphs. Faster (slightly)

The graph can be directed, but unweighted.

A node v.

DFS (G(V,E), v) : # The quintessential recursive algorithm

" MARK " v as EXPLORED # The 1st thing to do.

FOR EACH NEIGHBOUR w OF v: $O(n+m)$

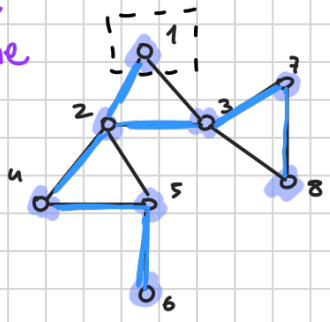
IF w HAS NOT BEEN MARKED AS EXPLORED (YET):

RUN DFS (G(V,E), w)

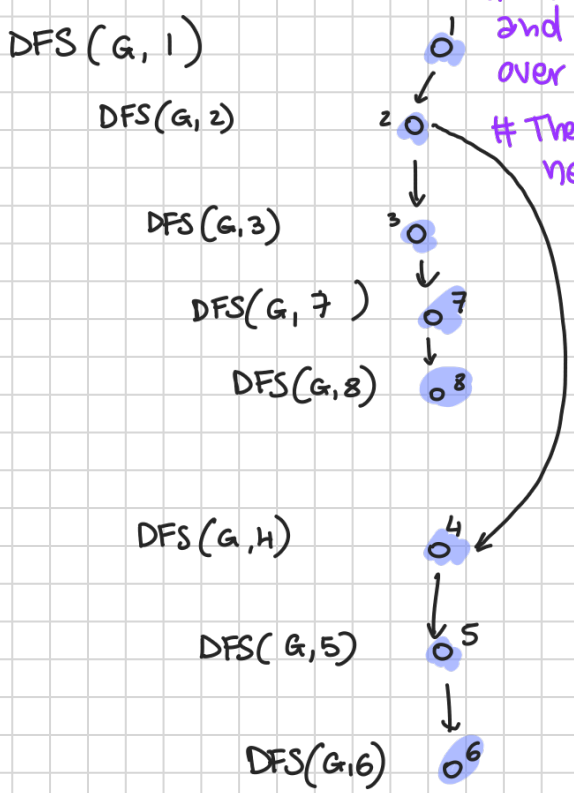
This algorithm has the property that it will visit each node that is reachable from the source.

This algorithm DOES NOT return a shortest path.

G(V,E)



Let's try to run the algo starting from 1.



The 1st thing that it'll do is to visit the node 1 and mark it as visited; then a loop will run over all the neighbours of 1.

Then it'll we move onto 2 and look over all the neighbours of 2.

← we randomey chose 3, we could have chosen 4 or 5 too.

$n+m$

DFS takes linear time because each edge will be considered exactly twice, from one of its endpoints and from its other endpoint, but we'll go through that edge at most once (when I go from v to w or when I go from w to v, or maybe never because I went to w or v through other edges).

Just remember these 2 searches as ways to visit nodes of a graph without writing a long code.

DFS does not guarantee for shortest paths, instead BFS guarantees for shortest paths if the graph is unweighted.