Algorithms

DATE: 7/4/2022

L: Assume that edge cost are pairwise distinct.

Let $\emptyset \subset S \subset V$ be a set of Nodes of $G(V, E)$.

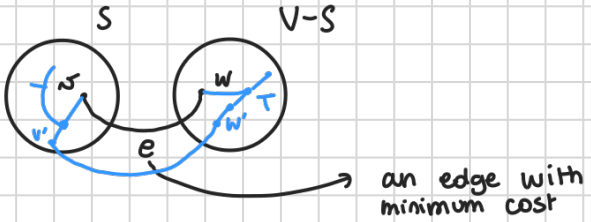
Let $e \in E$ be an edge having smallest cost among the edges having one endpoint in S, and one in $V-S$.

Then, EACH MST of $G(V, E)$ contains "e".

P: Let T be a spanning tree that does not contain the edge "e". We show that T is not a MST.

Let $e = \{v, w\}$, and suppose that $v \in S$ then $w \in V-S$. Since T is a spanning tree, there must exist a path π from v to w in T.

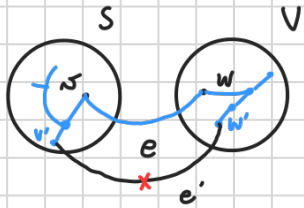
Let w' be the first node of π that is in $V-S$, and let v' be the one node preceding w' in π.



Then, $v' \in S$, o/w w' would not be the first node of π in

Let $e' = \{v', w'\}$.

Let us consider the set $T' = T - \{e'\} \cup \{e\}$.



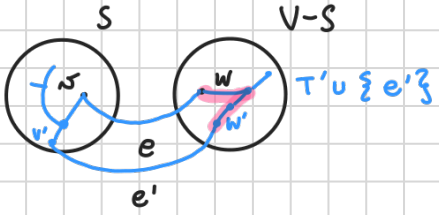
We want to prove that T' is a tree, and that its cost is smaller than the cost of T (this will entail that T is NOT a MST).

Observe that $G(V,T')$ is connected: $G(V,T)$ is connected (T is a spanning tree), and any path

in $G(V,T)$ that uses the edge e' can be rerouted on edges of T':

- we can first go through the portion of the path that goes from v' to v, then
- we can go through the new edge e,
- we can go from w to w'.

Thus, $G(V,T')$ is connected. It is also acyclic. Indeed, the only cycle in $G(V,T' \cup \{e'\})$ is the one going through "e" and "e'" but e' is not in $G(V,T')$. Thus, $G(V,T')$ does not have cycles.



Thus, $G(V,T')$ is a tree. We prove that the cost of T' is strictly smaller than the cost of T.
$T' = T - \{e'\} \cup \{e\}$. Thus, the $COST(T')$ of T' is equal to the $COST(T)$ of T minus $COST(e')$ plus $COST(e)$:

$$COST(T') = \sum_{f \in T'} cost(f) = \left(\sum_{f \in T} cost(f) \right) - cost(e') + cost(e)$$

but, $COST(e) < COST(e')$ since e is the edge with one endpoint in S, and one in V-S, having the SMALLEST COST in the set of edges having one endpoint in S and one in V-S and, also, e' has one endpoint in S and one in V-S.

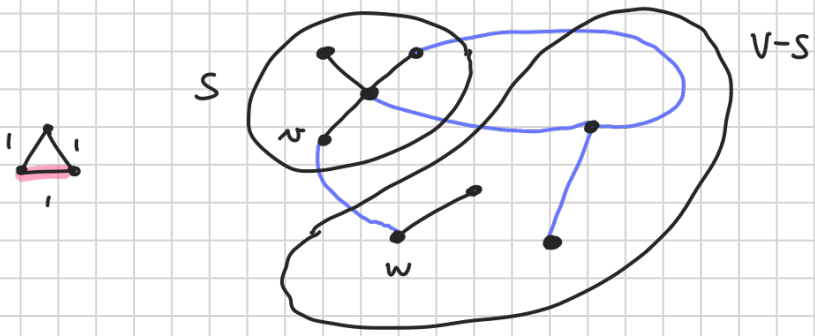
Thus, $COST(e') > COST(e)$ since there are no two edges having the same cost.

Thus, $Cost(T') < COST(T)$, and T is not a MST ■

THM: Assume that edge costs are pairwise different. KRUSKAL Algorithm produces a MST.

P: Suppose that, in a given iteration, Kruskal adds $\{v, w\}$ to T.

Let S be the set of nodes reachable from v, before adding $\{v, w\}$ to T.



Then $v \in S$, and $w \notin S$. Otherwise, the edge $\{v, w\}$ would create a cycle (and, by contradiction, Kruskal never creates cycles).

But then, $v \in S$ and $w \in V-S$. Moreover $\{v, w\}$ is the cheapest edge in the $S, V-S$ cut.

By the previous lemma, then, the edge $\{v, w\}$ is part of each MST. Thus, $T \cup \{\{v, w\}\}$ is still a partial optimal solution. The greedy property is then satisfied.)

Thus, the output of Kruskal algorithm is always a subset of a MST.

We prove that it is also a spanning tree - so that it must be a MST.

Well, Kruskal tries to add each edge "e" and avoids adding "e" only if it induces a cycle. Thus, given that the $G(V, E)$ is connected, the output of KRUSKAL algorithm will also be connected.
Since it is acyclic, it is a spanning tree ■

THM: Assume that edge costs are pairwise different. KRUSKAL Algorithm produces a MST.

P: Apply the lemma to the sets $S_1 = \{s\}, S_2, S_3, .., S_{n-1}$ that PRIM's algorithm considers.

At each step, PRIM selects the edge that the lemma guarantees to be in a MST ■

L: Assume that edge costs are pairwise distinct. Let C be the a cycle in G, and let $e \in C$ be the most expensive edge of C.

Then, no MST contains e.

Lemma for REVERSED-KRUSKAL

RIDDLE

Suppose you live in a building with "n" floors. You have a bunch of NOKIA 3310 phones. You would like to test the theory that they're indestructible... or, more seriously, you would like to pinpoint the highest floor i such that throwing a 3310 from the window of floor i guarantees that the 3310 reaches the ground unscathed (so that you can reuse it).

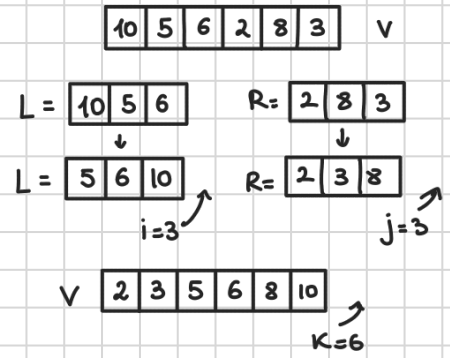
Given that you have k different NOKIA 3310 phones, what is the minimum number of tests you have to perform to find out the magic floor i?

DIVIDE-ET-IMPERA / DIVIDE-AND-CONQUER

– A TECHNIQUE FOR SPEEDING UP ALGORITHMS –

Let's start with the most famous divide-et-impera algo: (recursive)

```
DEF MergeSort (V):
    If len(V) ≤ 1:                      } O(1)
        RETURN V                        }
    L = V[ : len(V)/2 ]                 O(len(V))
    R = V[ len(V)/2 : ]                 O(len(V))
    L = MergeSort(L)                    T(len(V)/2)      RECURSION
    R = MergeSort(R)                    T(len(V)/2)
    i = 0
    j = 0                               } O(1)
    k = 0
    WHILE k < len(V):                   }
        IF i == len(L)                  }
            V[k] = R[j]                 } O(len(V))
            j += 1                      }
        ELIF j == len(R)                }
            V[k] = L[i]                 }
            i += 1                      }
```



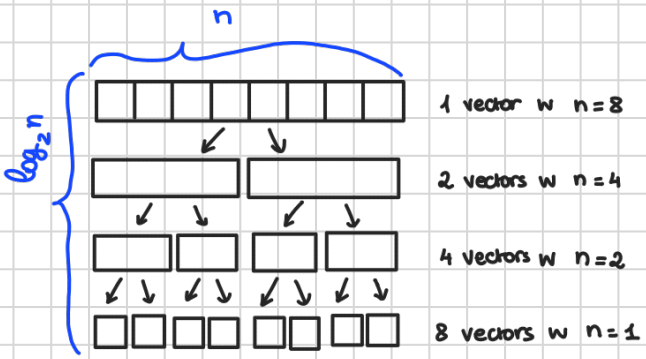
$$O(len(V)) + 2T(len(V)/2)$$

↑ function T

```
ELIF L[i] ≤ R[j]:
    V[k] = L[i]
    i += 1
ELSE:
    V[k] = R[j]
    j += 1
k += 1
```

O(1)

RETURN V



O(nlogn)

Let T(n) be the worst-case runtime of the algorithm on input of SIZE "n".

Let us assume for simplicity that $n = 2^t$ for some int. $t \geq 1$

If n is a power of 2 is easier to split each level of the array in equal parts.

Observe that

$\forall n \geq 2: \quad T(n) \leq 2 \cdot T(n/2) + c \cdot n$ (FOR SOME CONSTANT $c > 0$)

$T(0), T(1) \leq c$

We don't know the value of c; It's impossible to compute it.

Here we're saying that we don't know what the runtime of the algorithm is, what we know is that, If with T(n) we represent the runtime of the algorithm on an input of size n, then, that runtime is no more than twice the runtime of the algorithm on an input of size $\frac{n}{2}$ + some linear term.

This is a recurrence.

A recurrent inequality is when we have on both sides an unknown that depends on the other one.

Think of It as a recursive bound or a recursive equation/inequality that calls itself to bound Itself.

Another way to write It down is:

$$T(n) \leq \begin{cases} 2 \cdot T(n/2) + c \cdot n & \text{IF } n \geq 2 \\ c & \text{IF } n = 2 \end{cases} \qquad (R_2)$$

If $n=2$, c is a constant.

R_2 by itself does not give us a runtime bound like the "usual" ones.

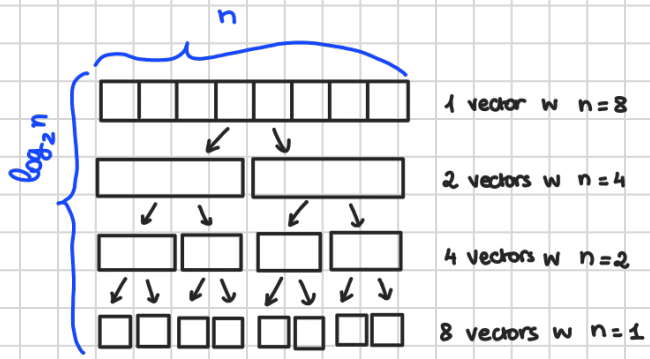
To get the "usual" bounds ($O(n \log n)$ or $O(n)$ or $O(n^2)$) we need to SOLVE the recurrence.

HOW TO SOLVE RECURRENCES? # In some cases is really hard.

VARIOUS APPROACHES

APPROACH 1

Unroll the recurrence for some number of levels, searching for a pattern that we can employ to solve the recurrence.



$O(n \log n)$

Approach 1 is similar to this, but a little bit more formal.

APPROACH 2 # Sometimes this can be faster.

Guess the solution, and verify that it works.

If for some reason we can guess the solution to the recurrence, such as $T(n) = 10 n \log n$, then we can use this into the inequality and check if it holds.

This is not trivial...

In this case we can almost do it, because we already know that mergesort takes $O(n \log n)$ time.