THEORY OF ALGORITHMS

1 BACKGROUND

DEFN: We consider functions of positive $\mathbb{R}$

$$O(f) = \{g \mid \exists k > 0 \to g(x) \leq k f(x),\ x \in \text{some half-line}\}$$

$$\Omega(f) = \{g \mid \exists k > 0 \to g(x) \geq k f(x),\ x \in \text{some half-line}\}$$

$$\Theta(f) = \{g \mid \exists c > 0 \to \tfrac{1}{c} f(x) \leq g(x) \leq c f(x),\ x \in \text{some half-line}\}$$

$$o(f) = \{g \mid \lim_{x \to \infty} \frac{g(x)}{f(x)} = 0\}$$

A *half-line* is a set of the form $[a, \infty)$

Notation We write $h = O(f)$ for $h \in O(f)$
and $h + O(k) = O(f)$ for $h +$ any member of $O(k) \in O(f)$

Note: $g \in o(f) \Rightarrow g \in O(f)$

Notation f is *linear* if $f = O(x)$
f is *polynomial* if $f = O(x^k)$, some $k \geq 0$

LOGARITHMS

$$\lg n \to \log_2 n$$
$$\ln n \to \log_e n$$
$$\log n \to \log_{10} n$$

LEMMA 1: $\log_b a = \log_b x \ \log_x a$

$$\log_b a = \frac{1}{\log_a b}$$

Examples

① $x^5 = O(5x^5)$ as $x^5 \leq 5x^5$ for $x \in [1, \infty)$, $k = 1$

② $x^5 = \Theta(5x^5 + 3x + 1)$

③ $x^5 = O(5x^5 + 3x + 1)$ since $\dfrac{x^5}{5x^5 + 3x + 1} \leq k$, some k.

④ $x^5 = O(x^6)$

⑤ $x^5 = o(x^6)$ as $\frac{1}{x} \to 0$ as $x \to \infty$

THM 1 (a) $\ln x = o(x)$ (i.e. logs have smaller order than linear fns)

(b) $x^k = o(e^x)$ (i.e. polynoms have smaller order than exp fns)

PROOF: (a) Must show $\lim_{x\to\infty} \frac{\ln(x)}{x} = 0$: use L'Hospital

(b) Must show $\lim_{x\to\infty} \frac{x^k}{e^x} = 0$ " "

COROLLARY: $\lg x = o(x)$

PROOF: $\lg x = \log_2 e \log_e x = \lg e \ln x$

and $\lg e \lim_{x\to\infty} \frac{\ln x}{x} = \lg e \cdot 0 = 0$

Similarly for log to any base.

DEFN 2: $\lceil x \rceil$ = least integer $\geq x$, called the ceiling of x

$\lfloor x \rfloor$ = greatest integer $\leq x$, called the floor of x

Note If n is an integer, then $\lfloor \lg n \rfloor + 1$ = no. of digits in binary expansion of n.

2 Introduction.

In the theory of algorithms we study classes of problems, usually indexed by the positive integers. We assume that given a class of problems indexed by $n \in \mathbb{N}$, we can assign a number $T(n) \in \mathbb{N}$ to each problem n, called its cost.

Usually $T(n)$ = time to execute a number of steps. We also refer to $T(n)$ as the complexity. There are two main ways to assign time complexity:
- worst case is maximum time required
- average

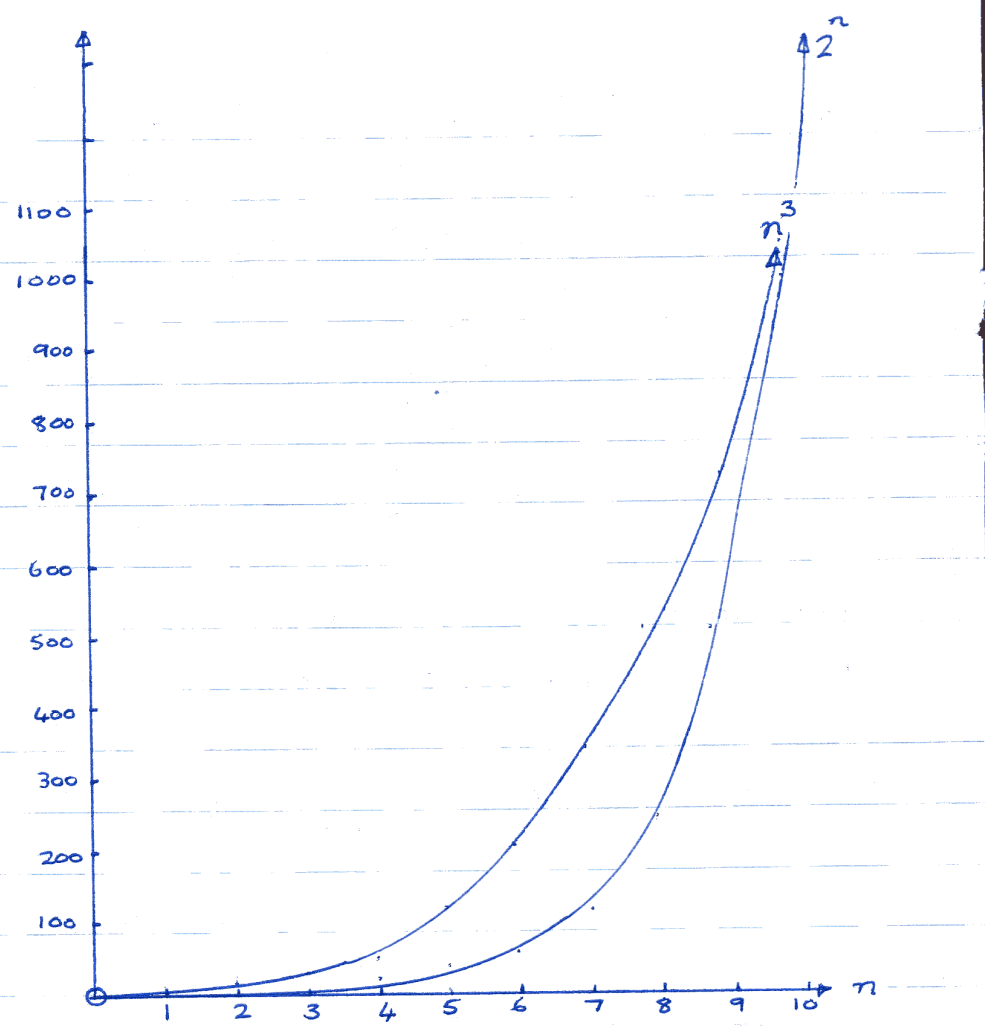
For any particular problem, we would like to know, over all algorithms, the upper and lower bounds of the complexity, as well as the asymptotic complexity.

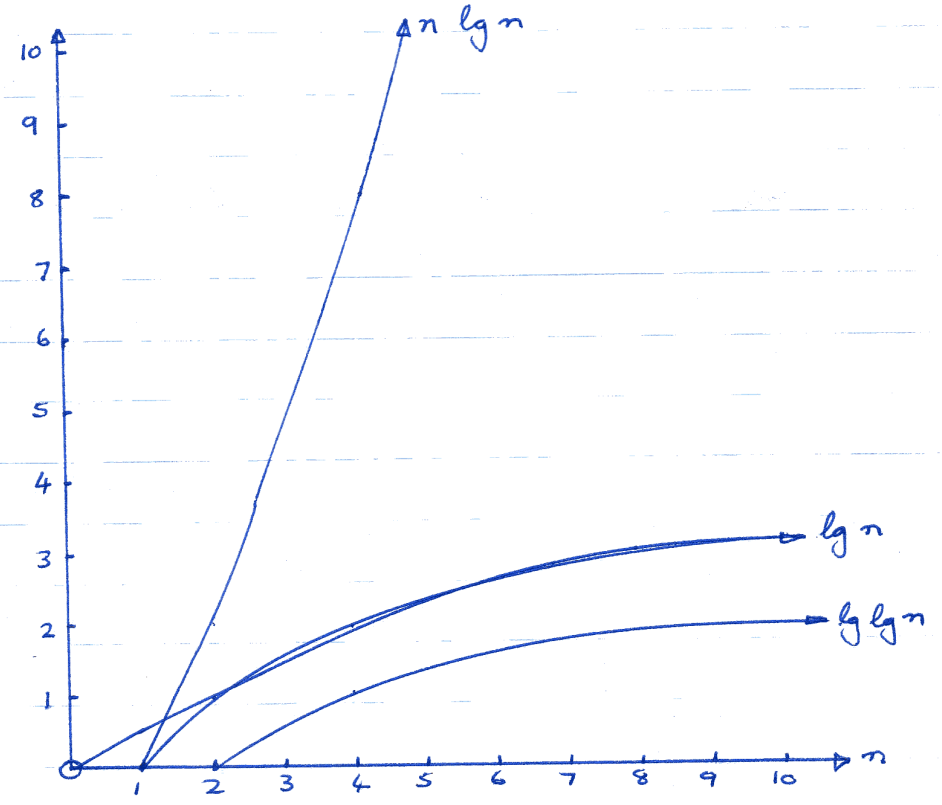
For any algorithm, we wish to investigate its validity and complexity, and provide an efficient "machine" implementation.

Generally, we assume that a good algorithm has $T_n = O(n^k)$ (ie polynomial complexity) while a poor algorithm has $T_n = O(2^n)$, (ie exponential complexity).

Further reading: Aho, Hopcroft & Ullman - Design & Analysis of Computer Algorithms.

FUNCTION	1	2	3	4	5	6	7	8	9	10	12	15	16	32	64	100
n^3	1	8	27	64	125	216	343	512	729	1000	1728	3375	4096	32768	262144	10^6
$\lg n$	0	1		2				3					4	5	6	
$n \lg n$	0	2		8				24					64	160	384	
$\lg \lg n$	–	0		1									2			
2^n	2	4	8	16	32	64	128	256	512	1024	4096	32768	65536	$\sim 4.3 \times 10^9$	$\sim 1.8 \times 10^{19}$	$\sim 1.27 \times 10^{30}$





3. A TECHNIQUE - "DIVIDE & CONQUER"

An an example, let us consider the following problem:
Given an input set S of n integers (let $n = 2^k$, some k, for simplicity), we wish to output $(\max_{a \in S} a, \min_{a \in S} a)$.

Algorithm 1:

```
PROCEDURE MM(S)
BEGIN
    MAX := RANDOM ELT OF S
    FOR x ∈ S DO
        IF x > MAX THEN MAX := x;
END;
```

Similarly for min.

Validity: (MAX, MIN) is clearly the max-min pair of elts of S at termination.

Complexity: $|S| = n$

Let $T(n)$ = no of comparisons used
To find max, we use $(n-1)$ comparisons
To find min, we use $(n-2)$ comparisons
$\therefore$, $T(n) = 2n - 3$

Algorithm 2:

```
PROCEDURE MAXMIN (S)
BEGIN
  IF #(S) = 2 THEN
     return (max {a, b}, min {a, b})
     else begin
        split S into two sets S1, S2, of size n/2
     (max1, min1) := MAXMIN (S1)
     (max2, min2) := MAXMIN (S2)
     return (max {max1, max2}, min {min1, min2})
     end (else)
END.
```

Validity: can give an inductive proof

Complexity: Let $T(n)$ = no. of comparisons

$T(2) = 1$

$T(n) = 2\,T\left(\frac{n}{2}\right) + 2$

This is a difference equation with initial condition $T(2) = 1$

It has a unique soln for every $n = 2^k$ (some k)

Guess solution $T(n) \simeq \frac{3}{2}n - 2$ if of form $a \cdot n + b$

Check: $T(2) = 1$

$T(2n) = \frac{3}{2}(2n) - 2 = 3n - 2$

$2\,T(n) + 2 = 3n - 2$

} equal.

Supplement: Solving difference equations (recurrence relations).

$$\left.\begin{aligned} T(n) &= a\,T\left(\frac{n}{b}\right) + d(n), \quad n = b^k \text{ some } k \\ T(1) &= 1 \end{aligned}\right\} \quad (1)$$

$$\begin{aligned} \text{i.e. } T(n) &= a\,T\left(\frac{n}{b}\right) + dn = a\left[a\,T\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right)\right] + d(n) \\ &= a^2\,T\left(\frac{n}{b^2}\right) + a\,d\left(\frac{n}{b}\right) + d(n) \\ &= a^2\left[a\,T\left(\frac{n}{b^3}\right) + d\left(\frac{n}{b^2}\right)\right] + a\,d\left(\frac{n}{b}\right) + d(n) \\ &= a^3\,T\left(\frac{n}{b^3}\right) + \left(a^2\,d\left(\frac{n}{b^2}\right) + a\,d\left(\frac{n}{b}\right) + d(n)\right) \\ &= \cdots\cdots \\ &= a^j\,T\left(\frac{n}{b^j}\right) + \sum_{i=0}^{j-1} a^i\,d\left(\frac{n}{b^i}\right) \end{aligned}$$

Since $n = b^k$ $\left(\Rightarrow T\left(\frac{n}{b^k}\right) = T(1) = 1\right)$, let $j = k$

$$\Rightarrow T(n) = a^k + \sum_{i=0}^{k-1} a^i\,\underbrace{d\left(b^{k-i}\right)}_{= d\left(\frac{b^k}{b^i}\right) = d\left(\frac{n}{b^i}\right)}$$

But $k = \log_b n = (\log_a n)(\log_b a)$

$$\Rightarrow a^k = \left(a^{\log_a n}\right)^{\log_b a} = n^{\log_b a}$$

4. SORTING

INPUT: Set S of integers
$\#(S) = 2^k$, some $k > 1$
OUTPUT: An n-tuple containing elts of S in ascending order

4.1 BUCKET SORT

```
PROCEDURE BUCKETSORT (S, m)      (* 0 ≤ Si ≤ m *)
BEGIN
    initialize Q(i)              (0 ≤ i ≤ m)
    append Si to Q(Si)           (0 ≤ i ≤ m)
    concatenate Q(0), Q(1), ..., Q(m)
END
```

Validity: clear
Complexity: $O(m+n)$

4.2 MERGE SORT

```
PROCEDURE MERGE (S, T)           (* S, T sorted lists *)
do until both S, T empty
    append max of two MAX elts of S, T to output list L
if max ∈ S then S = S - {max}
          else T = T - {max}
end
```

This does $\#(S) + \#(T)$ comparisons.

PROCEDURE SORT (i, j, S) $(* \; i \le j \le n, S = (S_1, \ldots, S_n) \; *)$
$(* \text{ we sort } (S_i, \ldots, S_j) \; *)$

if $i = j$ then return (S_i)
else begin
$m = \lfloor \frac{1}{2}(i + j - 1) \rfloor$
return MERGE (SORT (i, m, S), SORT (m, j, S))
end
end

To sort S, we call SORT $(1, n, S)$

Validity Can be proven by induction
Complexity $T(n)$ = no. of comparisons.

$T(1) = 0$

$T(2) = 1$

$T(n) = 2T(\frac{n}{2}) +$ (comparisons in merging two lists of size $\frac{n}{2}$) (if $n = 2^k$, some k)

$\Rightarrow T(n) = 2T(\frac{n}{2}) + n - 1$, a difference equation.
or recurrence relation

For mergesort, show that $T(n) \le an \log n + bn + c$

with $b, c = 0, a = 1$

i.e. $T(n) \le n \log n \quad (n \ge 1, n = 2^k)$

We have: $\begin{cases} T(n) = a\,T\left(\frac{n}{b}\right) + d(n) \\ T(1) = 1 \end{cases}$

Then $T(n) \begin{cases} = O\left(n^{\log_b a}\right) & b \neq a \\ = O(n \log n) & b = a. \end{cases}$

If we split a problem of size n into similar problems of size $\frac{n}{b}$ by doing linear work, then

$$T(n) = a\,T\left(\frac{n}{b}\right) + \alpha n + \beta$$

In view of the above, guess:

$$T(n) \leq K_1\, n^{\log_b a} + \alpha' n + \beta' \qquad (b \neq a)$$
$$\leq K_2\, n \lg n + \alpha' n + \beta' \qquad (b = a)$$

Try a proof by induction, and hence guess the constants.

Example: Mergesort: $T(n) = 2\,T\left(\frac{n}{2}\right) + (n-1)$
$T(1) = n$

Guess $T(n) \leq n \log n$ (ie $T(n) = O(n \log n)$)
Prove by induction

Note: For $T(n) = a\,T\left(\frac{n}{b}\right) + n$:

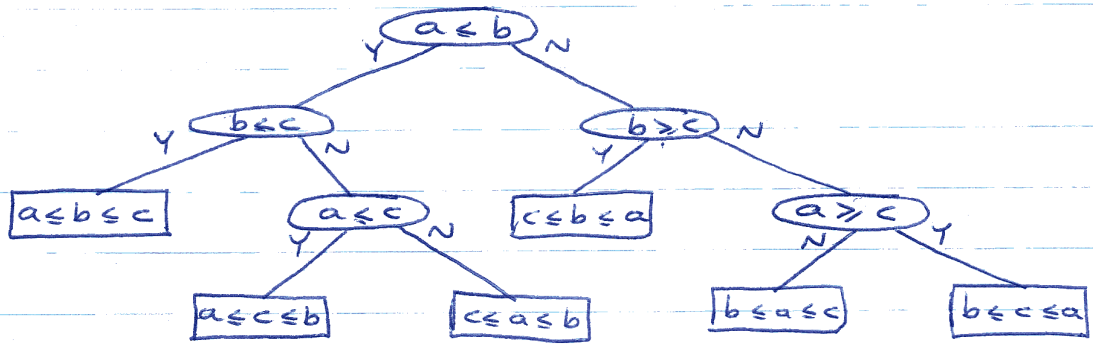
Problems	$T(n)$	Problems
2 probs size $\frac{n}{2}$	$O(n \log n)$	4 probs size $\frac{n}{4}$
3 probs size $\frac{n}{2}$	$O(n^{\log_2 3})$	9 " " "
4 " " "	$O(n^2)$	16 " " "
8 " " "	$O(n^3)$	64 " " "

Lower Bounds for Comparison Sorts.

Represent algorithm by "decision tree" (binary tree) with nodes: 

Any comparison algorithm can be written in this way.

Ex Sort $\{a, b, c\}$.



Note: Each possible ordering is a leaf of the decision tree. The worst case performance of an algorithm is the longest route from the root to a leaf (the "height" of the tree)

LEMMA: A binary tree of height h has $\leq 2^h$ leaves.

COROLLORY (a) If n = no of vertices of binary tree of height h, then $n \leq 2^{h+1} - 1$

(b) If all levels but one of the trees are filled (ie not terminal but proper binary nodes) then $h = O(\log n)$

THEOREM: The decision tree for a comparison sort of n integers has height h such that $h \geq n!$
(there are $n!$ possible orderings, ie, leaves)

COROLLARY: Comparison sort is $\Omega(n \log n)$

PROOF: Since each ordering must appear as a leaf,

no. of leaves $\begin{cases} \geq n! \\ \leq 2^h \end{cases}$

So $h \geq \log n!$

and $h \geq \log n(n-1)\ldots\lceil \frac{n}{2} \rceil$

$\geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2}$

hence $h \geq \Omega(n \log n)$

GREEDY ALGORITHMS

Given: coin denominations 1, 5, 10, 20.
Find: the least number of coins totalling 57

Greedy method: "use largest denomination possible and subtract"

Consider: denominations 1, 10, 25; total 31
In this case, the greedy method fails to find optimal soln $3 \times 10 + 1$

Finding Minimal Spanning Trees.

A graph $G = (V, E)$ is a set of vertices joined by a set of edges.

A cycle is a set of edges which can be followed cyclically

A tree is a connected graph without cycles.

A spanning tree of graph G is a tree passing through all vertices of G.

Note: (i) In a tree, any two vertices are joined by a unique path (by connectivity and lack of cycles)

(ii) Given a tree, if we join any two vertices with an (additional) edge, we create a cycle.

Suppose each edge of connected graph G has a cost $C : E \to \mathbb{R}^+$

Problem: Find the minimal (cost) spanning tree of a connected graph G with edge cost $C : E \to \mathbb{R}^+$

The number of spanning trees is exponentially proportional to the number of edges of G, so exhaustion is a poor method.

Algorithm: To find a spanning tree of G.

repeat break any cycle until no cycles remain

To find the spanning tree, however, it is preferable to build up a tree rather than breaking one down.
For minimal spanning trees:

Method 1: Build tree at each stage by choosing least cost outgoing edge, starting with edge of least cost.

Method 2: (pto) Spanning tree only.

$T := \phi$; $U := \{1\}$

do while $U \neq V$

find edge $e = (u, v)$ with $u \in U$, $v \in V \setminus U$

$T := T \cup \{e\}$

$U := U \cup \{v\}$

end while

Method 3 (Prim's alg. for min. sp. trees)

$T := \phi$; $U := \{1\}$

do while $U \neq V$

find edge $e=(u,v)$ with $u \in U$, $v \in V \setminus U$.

which has least cost amongst all such edges.

$T := T \cup \{e\}$; $U := U \cup \{v\}$

end while.

Complexity : The loop is executed $(n-1)$ times, where $n = \|V\|$.

Each time may have to run through approx n steps in each loop, i.e. $O(n^2)$.

Validity Let $\{(V_1, T_1) \ldots (V_k, T_k)\}$ be a spanning forest for $G_1 = (V, F)$.

Let $T = \hat{\bigcup_{i=1}} T_i$

Then there is a spanning tree of G which has all the edges in T and also the edge e of least cost in $E - T$ which is of least cost among all spanning trees containing T

Proof: Suppose $S' = (V, T')$ is a spanning tree of lesser cost than any containing T, $T \subseteq T'$, $e \notin T'$. $e = (v, w)$, $v \in V_i$ so $w \notin V_i$ (or else $e \in T_i$)

KRUSKAL'S ALGORITHM FOR A MINIMAL SPANNING TREE.

Notation: T - spanning tree
VS - a collection of disjoint sets of vertices
Q - a list of edges in order of increasing cost.

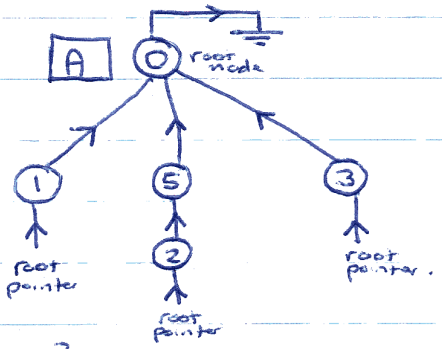
Algorithm:
```
begin
  T := Φ , VS := Φ
  Construct Q of edges E of G in increasing order of cost
  For each v ∈ V do VS := VS + {v}
  while ||VS|| > 1 do begin
      let (v, w) be edges of Q of least cost.
      Q := Q \ {(v, w)}
      if v, w are in different sets W1, W2 of VS
          then begin
              replace W1, W2 by W1 ∪ W2 in VS
              add (v, w) to T
          end
  end
end
```

Validity: By Lemma
Complexity: Constructing Q takes $O(\|E\| \log \|E\|)$ by mergesort.
The while loop looks at $\|E\|$ edges, and for the endpoints checks which sets they're in, and (possibly) performs a union operation.
The time taken to do this depends on how we implement the data structures and the union/find operations.

Implementation of Set, Union, Find.

Data structure for set: Represent a subset of $\{0, 1, 2, \ldots, n\}$ by a tree whose vertices are elts of sets. The root node has extra data (the name of the set).

Eg $A = \{0, 1, 5, 3, 2\}$



How do we find the union? We are dealing with disjoint sets so we need not check for redundancy. We simply take the null pointer of one set and change it to point to a root pointer of the other.

Find starts at a terminal node and works up from child to parent. Thus the upper bound on find is the maximum height of any tree in the forest.

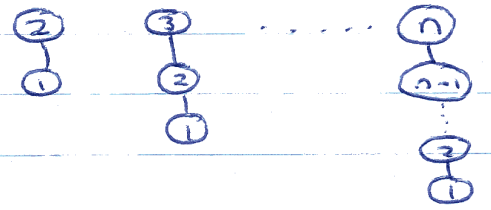
What is the worst case complexity of n unions and n finds?

Notation: UNION $(A, B, C) :=$ set $C = A \cup B$

So we have: UNION $(1, 2, 2)$ UNION $(2, 3, 3), \ldots,$ UNION $(n-1, n, n)$

FIND (1), FIND (2), $\ldots$, FIND (n)

The method of taking unions will be a worst case as we keep adding onto the end of the tree, and build up a very high tree.

if we have trees 

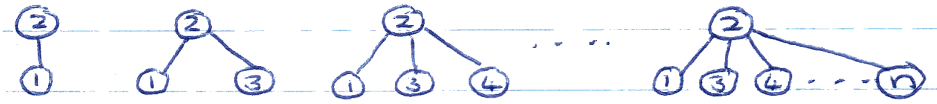
so FIND(1) takes $n-1$

FIND(2) takes $n-2$

FIND($n-1$) takes 1.

Total is: $1 + 2 + \dots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$

However, by a more intelligent building of unions, we can cut this down. Build up unions as:



so new union alg: always adjoin tree of fewer vertices to the tree with more vertices – this only adds a simple addition operation to keep track of heights.

What is this complexity?

Lemma: Let there be n-elts on which we apply the improved UNION-FIND. Then no tree has height $> \log_2 n$.

[PROOF: When we move an elt to a new set via the union op (ie elt comes from a smaller set), then the distance to the root increases by 1, and #(new set) ≥ 2 #(old set).

Hence $2^{\text{total no of moves}} \leq n$, so tot no of moves $\leq \log_2 n$

$\Rightarrow$ height $\leq \log_2 n$ ⌋

So each find is $O(\lg n)$ and total finds is $O(n \lg n)$

And n unions is $O(n)$. The computation is thus $O(n \log_2 n)$.