# TOC

- Computing Min- Max
- Largest and Second Largest
- Lower Bounds (adversary)

# Algorithm1 to find Min-Max in 2n-3 comparisons

- STEPS:

1.All keys are assumed to be distinct.

2.Maximum of n keys can be found in n-1 comparisons.

3.Eliminating max key we are left with n-1 keys.

4. Minimum of n-1 keys can be found in n-2 comparisons.

5.Total no of comparisons are (n-1)+(n-2)=2n-3.

# Algorithm2 to find Min-Max in 3n/2 - 2 comparisons

- STEPS:

1. All keys are assumed to be distinct.
2. We perform comparison on n/2 pairs.
3. After n/2 comparisons we get n/2 winners and n/2 loosers.
4. So MAX of n keys is the maximum of these n/2 winners.
5. Similarly MIN of n keys is the minimum of these n/2 loosers.

# Lower Bounds

- Any algorithm to find min and max of n keys by comparison of keys must do at least $3n/2 - 2$ comparisons in the worst case.
- Proof:
  - Assume that the keys are distinct
  - Let x denotes max and y denotes min
  - We must know that every key other than x has lost to some comparison ----- n-1 unit of information required.
  - Also, we must know that every key other than y has won in some comparison ----- n-1 unit of information required.
  - Total : $2n - 2$ unit of information required to declare that x is max and y is min.

# Adversary

- Devises a strategy that gives away minimum information.

| Key status | Meaning |
|---|---|
| W | Has won at least one comparison and never lost |
| L | Has lost at least one comparison and never won |
| WL | Has won and lost at least one comparison |
| N | Has not yet participated in a comparison |

| Status of keys x and y compared by an algo | Adversary chooses | New status | Units of new information |
|---|---|---|---|
| N,N | x >y | W,L | 2 |
| W,N/ WL, N | x > y | …,L | 1 |
| L,N | x < y | …, W | 1 |
| W,W | x > y | …,WL | 1 |
| L,L | x > y | WL,… | 1 |
| W,L/WL,L/W,WL | x > y | No change | 0 |
| WL,WL | Consistent with assigned values | No Change | 0 |

# Finding the Largest n the Second Largest

- If an algorithm first finds largest in (n-1) comparisons and then finds $2^{nd}$ largest from the remaining (n-1) elements in (n-2) comparisons, then

  Total no. of comaprisons to find $2^{nd}$ largest

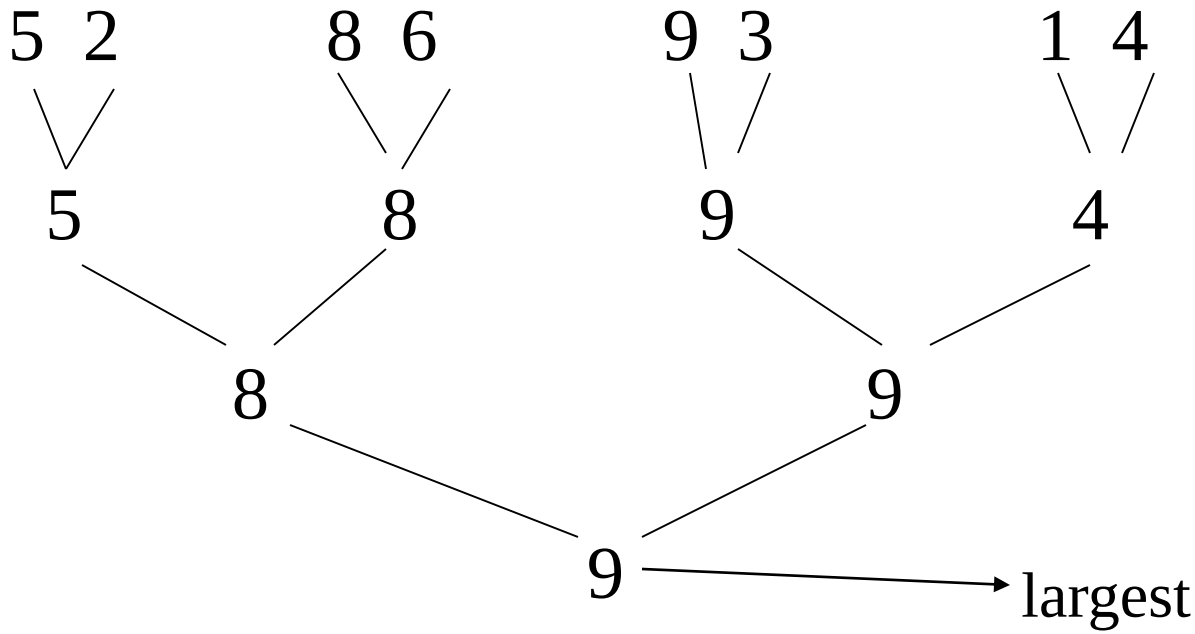
  $= ( n-1 ) + ( n-2 )$

  $= 2n - 3$

Next, we'll see an algorithm which does better than this.

# Tournament Method

- Consider the method to find largest.
  For e.g



- In this tree, we have 8-1 = 7 comparisons.

with the largest element

As 8 , 3 & 4 loose only once to reach largest, they are the only candidates for $2^{nd}$ largest.

no. of candidates for $2^{nd}$ largest

= height of the tree

= log n

Note:- If no. of keys is odd then extra key is considered in the end.

- As no.of keys = lg n (for $2^{nd}$ largest )

  therefore, no. of comparisons to find $2^{nd}$ largest

  = lg n -1 .

  Thus , total comparisons

  = ( n-1) + ( lg n -1)

  = n + lgn – 2

# Lower bound for Largest n Second Largest

- Any algorithm that works by comparing keys to find the second largest in a list of n keys must do at least n + log n – 2 comparisons.

- If we can show that `max' participated in at least log n comparisons we are done because
  - Any algorithm that computes second largest must compute the largest. Why?
    - To know that x is second largest, it must know that it is not the largest. That is it must loose to at least one comparison ( and in fact exactly one, since if it looses to two or more it is not the second largest.). And, when x looses to exactly one key , it knows that that key must be the largest.
  - If 'max' looses to only one key, we know that that key must be the second largest. But what if 'max' looses' to more than one key.
  - So, any algorithm must do at least n-1 comparisons (where 'max' is compared with the rest of the keys ) and compare the "loosers to max" to get the second largest.

# Adversary

- Assign weight w(x) to each key x.
- Initially w(x) = 1 for all x.

# Adversary modifies the weights as per the following strategy.

| Case | Adversary response | New weights |
|---|---|---|
| $w(x) > w(y)$ | $x > y$ | $w(x) \leftarrow w(x) + w(y)$, $w(y) \leftarrow 0$ |
| $w(x) = w(y) > 0$ | $x > y$ | $w(x) \leftarrow w(x) + w(y)$, $w(y) \leftarrow 0$ |
| $w(y) > w(x)$ | $y > x$ | $w(y) \leftarrow w(x) + w(y)$, $w(x) \leftarrow 0$ |
| $w(x) = w(y) = 0$ | consistent with previous results | No change |

# Observations

- A key has lost a comparison iff its weight is zero.i.e. once a key has lost a comparison, its weight will never become non zero in future.

Proof: Notice that once a key y has lost a comparison its weight becomes zero. Once its weight becomes it can never be compared with a key x whose weight w(x) is less than w(y). Hence w(y) never changes hereafter.

# Observations

- The sum of the weights is always n. This is true initially and it is preserved when weights are updated.
- When the algorithm stops, only one key has a non-zero weight. Otherwise there would be at least two keys (say a and b) that never lost a comparison. The adversary can give arbitrarily high and distinct values to these two keys.
  - Now, if the algorithm declares a as largest and b as second largest, adversary can always choose the values the other way round and make the algorithm's choice of largest and second largest incorrect.

- Claim: In any algorithm to compute the *largest* , the *largest* must be compared with at least ceil(log n) keys in the worst case.
- Proof: Let x be the key that has non-zero weight when the algorithm stops. Then clearly x = largest. We'll show that x has directly won over at least ceil(log n) distinct keys.

- Let $w_k = w(x)$ be the weight of x after the kth comparison won by x against a previously undefeated key. Then, we'll prove that

$$w_k \leq 2w_{k-1}$$

  - If x has won against few previously defeated keys in between, its weight does not change. So,
  - $w_k = w_{k-1} + w(y)$ where y is the previously undefeated key
  - Clearly, $w(y) \leq w_{k-1}$ for else y wouldn't loose to x.
  - Thus, $w_k \leq 2w_{k-1}$

- Let K be the number of comparisons x wins against previously undefeated keys, then

$$n = w_K \leq 2^K w_0 = 2^K$$

Thus $K \geq \log n$ or $K \geq \text{ceil}(\log n)$

# General Selection

- General Selection or just the Selection problem is to find an element of rank k in a given array.
- Randomized algo : Similar to Qsort except that instead of recursing on both the subproblems, we recurse only on one which is expected to contain the required element : Hence, Expected  Time is linear instead of n log n:
- IDEA: Pick a pivot, divide the problem into two subproblems as earlier, compute the number say q of elements in the left sublist, if it is = k – 1, our pivot is the required element. If it > k-1, we recurse in the left sublist else(q < k-1) we recurse in the right sublist looking for an element of rank k-q-1.
- Analysis is on the same line : do it yourself.

# Selection: Worst Case Linear