SORTING

MERGESORT

Ruben Acuña

Fall 2018

Last update: 3/5/2022

THE ALGORITHM

IN PRACTICE

- Realistically, you won't use Selection, or other $O(n^2)$ sorting algorithms. Too slow.
- Insertion sort you may still see – almost O(n) complexity for sorting is nice after all.
- Shellsort is still used, but is starting to be replaced by Quicksort. See Section 2.3 for information on Quicksort.

- In this section we discuss an O(nlog(n)) algorithm: **Mergesort**.
  - Where is it used? Right in the Java core for starters:
  - http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

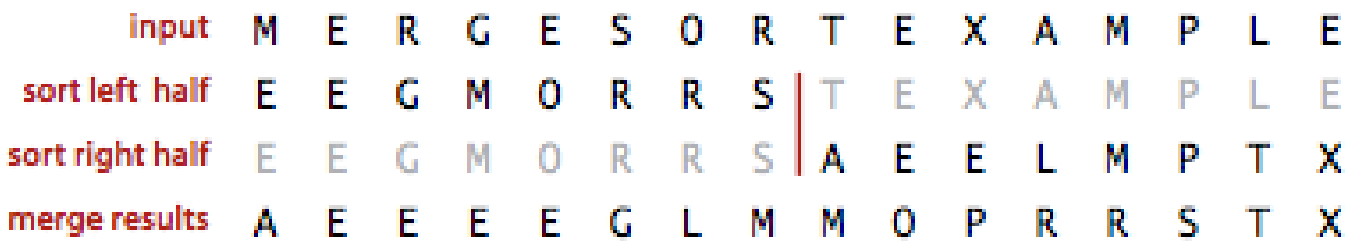
DIVIDE AND CONQUER ALGORITHMS

- Until we now, we haven't mentioned any "overall" algorithm design techniques – just existing data structures and algorithms.
  - Algorithm *design* is typically a follow up class to ours.
- *In a **divide and conquer**, an algorithm is designed to divide a problem into set of sub-problems, repeating until a trivial problem emerges, and then combine the result of the sub-problems to give a solution to the original problem.*
- This often gives an algorithm's complexity "log(n)" – this represents the algorithm performing a reduction on the input size.

- What is the relation between this idea and recursion?
- What would be harder to parallelizable, an DnC algorithm or normal iterative algorithm?

THE CONCEPT

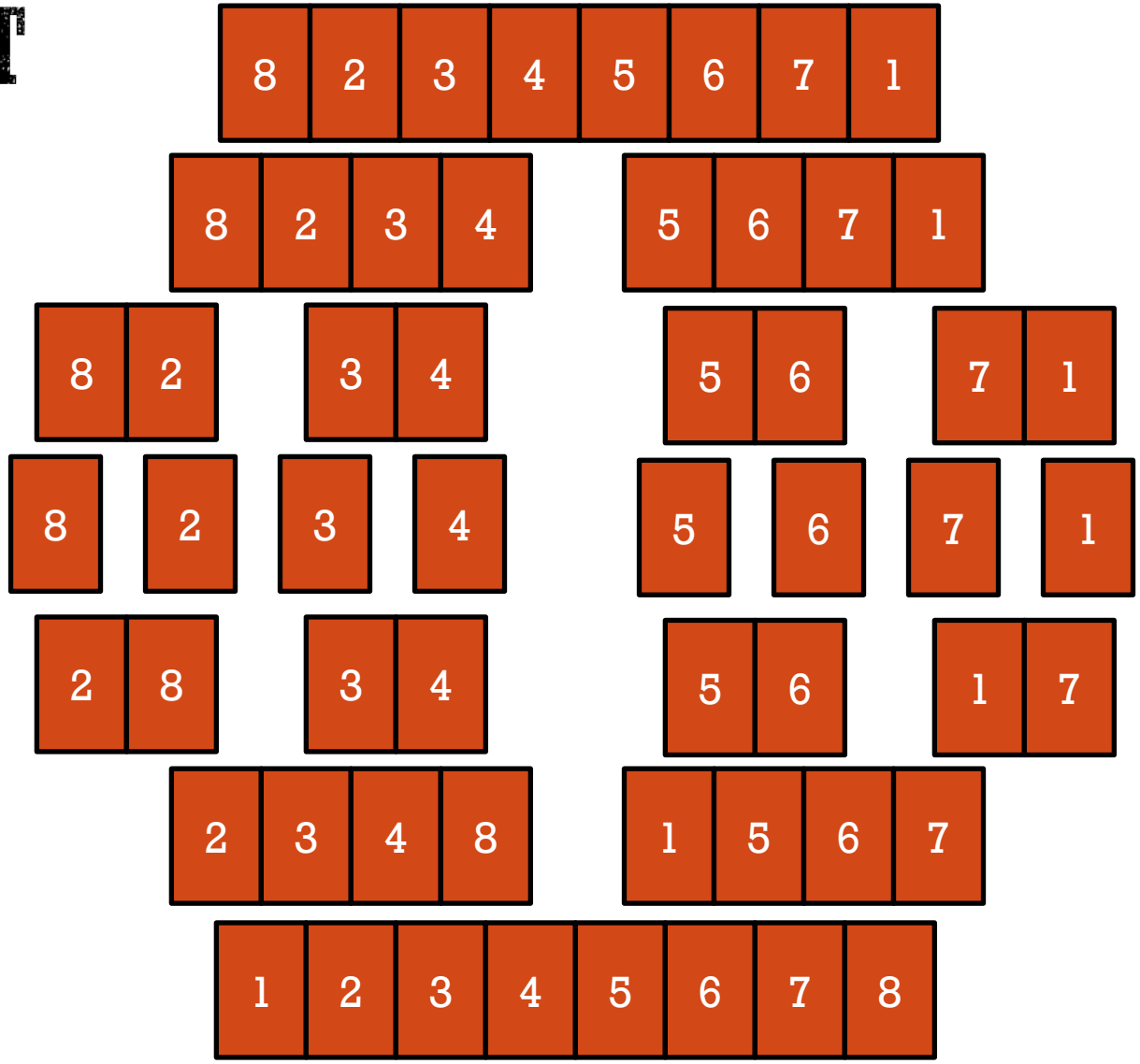
- Basic Idea:
  - Take an array, then split it into halves.
  - Sort each of the halves. (use recursion!)
  - Then combine the sorted halves to form a sorted whole.



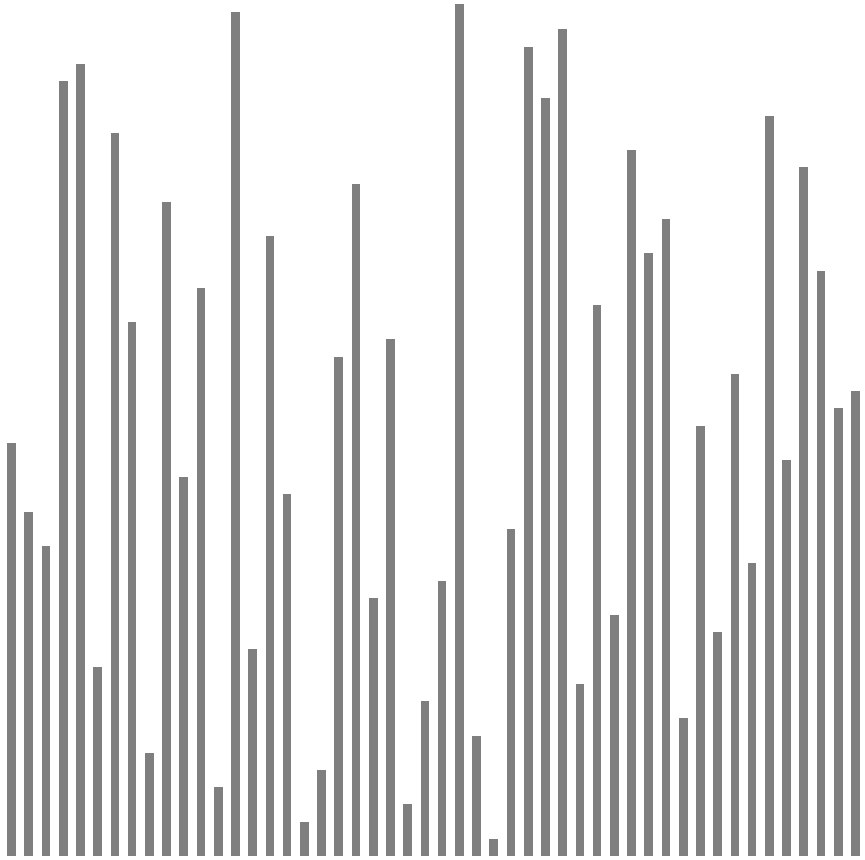
Mergesort overview

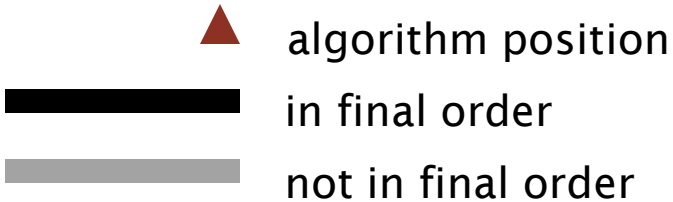
Image from Sedgewick and Wayne.

MERGESORT CONCEPT



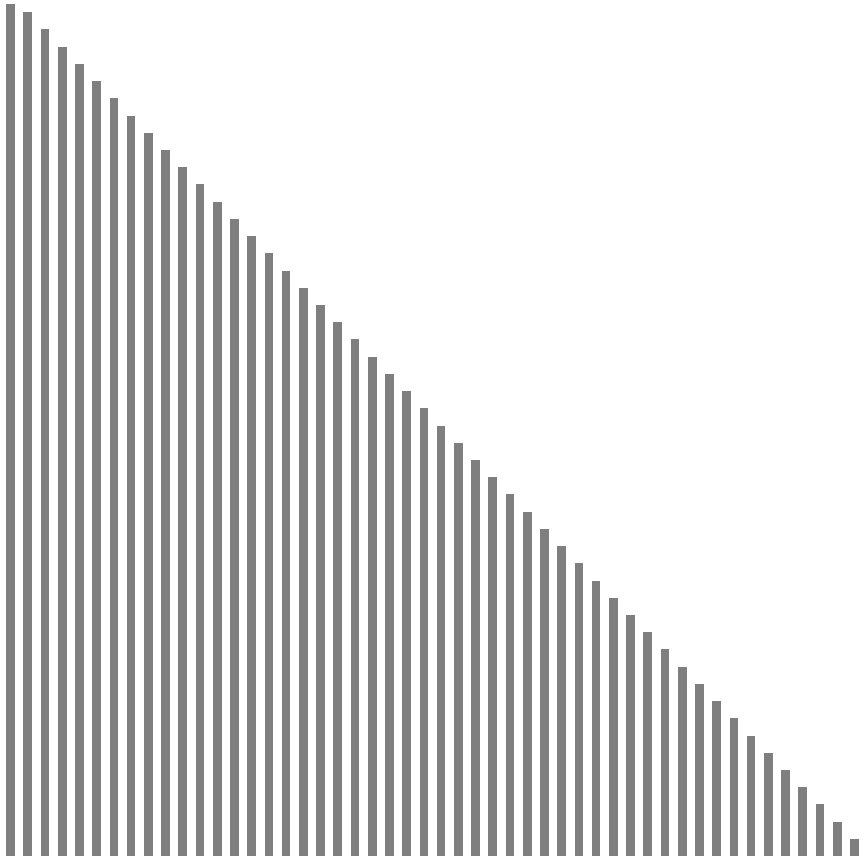
TESTING: RANDOM ORDER





From Sedgewick and Wayne, sorting-algorithms.com.

TESTING: REVERSED



From Sedgewick and Wayne, sorting-algorithms.com.

IMPLEMENTATION

```
public static void sort(Comparable[] a) {
        Comparable[] aux = new Comparable[a.length];
        sort(a, aux, 0, a.length-1);

        assert isSorted(a);
    }

    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;

        sort(a, aux, lo, mid);
        sort(a, aux, mid + 1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
    {
        //...
    }
```

What about just writing (lo + hi)/2?

IMPLEMENTING MERGE

```
public static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    assert isSorted(a, lo, mid);
    assert isSorted(a, mid+1, hi);

    int i = lo, j = mid+1;

    for(int k = lo; k <= hi; k++)
        aux[k]=a[k];

    // merge back to a[]
    for (int k = lo; k <= hi; k++) {
        if      (i > mid)                a[k] = aux[j++];
        else if (j > hi)                 a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else                             a[k] = aux[i++];
    }

    assert isSorted(a, lo, hi);
}
```

Why asserts?

IMPLEMENTING MERGE

	lo	hi	a[] 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	0	1	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	2	3	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 1, 3)	0	3	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	4	5	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	6	7	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 5, 7)	4	7	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, 0, 3, 7)	0	7	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	8	9	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	10	11	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, 8, 9, 11)	8	11	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 12, 12, 13)	12	13	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 14, 14, 15)	14	15	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	12	15	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	8	15	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, 0, 7, 15)	0	15	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for top-down mergesort

CONTEXT

	Insertion Sort			Mergesort		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

- What gives better performance, a better computer or a better algorithm?

Table from Sedgewick and Wayne.

PROPERTIES: COMPARISONS

For the previous sorting algorithms, we analyzed the number of comparison and exchanges they made on average:

- Selection Sort:

  Comparisons: $\sim \frac{n^2}{2}$

  Exchanges: $\sim n$

- Insertion Sort:

  Comparisons: $\sim \frac{n^2}{4}$

  Exchanges: $\sim \frac{n^2}{4}$

- Shellsort:

  Comparisons: $O(n^{\frac{3}{2}}), \Omega(nlogn)$

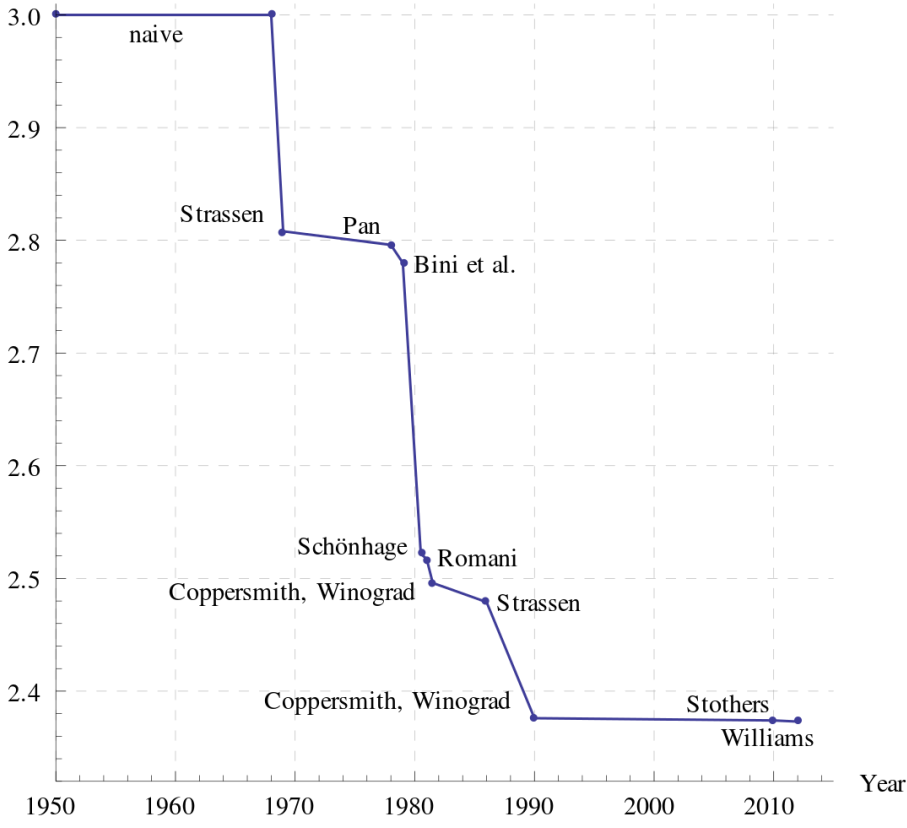
- Selection and insertion sort aren't that bad to analyze – they are iterative.
- Shellsort is more tricky because the gap sequence (choice of h) is really specifying a family of sorting algorithms (think: h=1 is insertion sort).

- Mergesort on the other hand is recursive. Time to break out our knowledge of recurrences.

INTERMISSION

MATRIX MULTIPLICATION

- Matrix multiplication is a classic problem.
- Many major problems have reductions to matrix multiplication. Remember: it serves as a general constraint system.
- Multiplying matrices is pretty much what HPC is all about.
- The naive algorithm is $O(n^k)$, k=3 where one simply has nested loops.
- Since matrix multiplication is so common people try to devise faster algorithms.
- Where will it stop?



$\Omega(n^{2.3728639})\ldots$?

Image from Wikipedia Commons.

ALGORITHM ANALYSIS

THE CLAIM

- "Top-down mergesort uses between $\frac{1}{2}nlg(n)$ and $nlg(n)$ compares to sort any array of length N" (Sedgewick and Wayne).
  - Our goal is to show this statement is true.
  - (FYI: The statement is in a sense arbitrary. For learning purposes we are starting with a truth that is known. In practice, one would make informed guesses from the structure of algorithm and try to prove them in order to learn more about the structure of the problem. Guesses improve over time and one finally finds the correct solution. This messy work is, sadly, excluded from most proof descriptions.)

- Proof via direct construction:
  - Let C(n) represent the number of comparisons for an input of n elements.

PROOF FORMULATION

- First: write recurrence for C(n).
  - Need to get out the code…

Trivial:

C(0)=0 loop in merge doesn't run

C(1)=0 2nd branch only in merge

Recurrence:

$C(n) \leq C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$ upper

$C(n) \geq C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + \left\lfloor \frac{n}{2} \right\rfloor$ lower

```
private static void sort(Comparable[] a, ...) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;

    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

public static void merge(Comparable[] a, ...) {
    assert isSorted(a, lo, mid);
    assert isSorted(a, mid+1, hi);

    int i = lo, j = mid+1;

    for(int k = lo; k <= hi; k++)
        aux[k]=a[k];

    // merge back to a[]
    for (int k = lo; k <= hi; k++) {
        if      (i > mid)                  a[k] = aux[j++];
        else if (j > hi)                   a[k] = aux[i++];
        else if (less(aux[j], aux[i]))a[k] = aux[j++];
        else                               a[k] = aux[i++];
    }

    assert isSorted(a, lo, hi);
}
```

PROOF FORMULATION

Let's choose to work with the upper bound – more practical.

Our recurrence is then:

- $C(0) = C(1) = 0$
- $C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$

Much algebra follows…

Let n look like a power of two. (WLOG: it is necessary that C work for all 2^k, and we assume C looks "continuous".)

Thus, $\mathrm{n} = 2^{\mathrm{k}}$.

So, $\left\lfloor \frac{n}{2} \right\rfloor = \left\lfloor \frac{2^{\mathrm{k}}}{2} \right\rfloor = \left\lfloor 2^{\mathrm{k}-1} \right\rfloor = 2^{\mathrm{k}-1}$. Ditto for the ceiling. This gives:

- $C\left(2^{\mathrm{k}}\right) = 2C\left(2^{\mathrm{k}-1}\right) + 2^{\mathrm{k}}$

Goal: eliminate RHS use of C by expansion.

PROOF

Have:

$$C(2^k) = 2C(2^{k-1}) + 2^k$$

Divide by 2^k:

$$\frac{C(2^k)}{2^k} = \frac{C(2^{k-1})}{2^{k-1}} + 1$$ (RHS denom is 2^{k-1}, not 2^k, because of the coefficient.)

Recursively apply formula:

$$\frac{C(2^k)}{2^k} = \left(\frac{C(2^{k-2})}{2^{k-2}} + 1\right) + 1$$

$$\frac{C(2^k)}{2^k} = \left(\left(\frac{C(2^{k-3})}{2^{k-3}} + 1\right) + 1\right) + 1$$ (and so on…)

$$\frac{C(2^k)}{2^k} = \frac{C(2^0)}{2^0} + k$$

Simplify and multiply by 2^k:

$$C(2^k) = k2^k$$

Since $n = 2^k$, and $\lg(n) = k$:

$$C(n) = \lg(n)n$$

WHAT'S NEXT?

- We have shown that Mergesort is O(nlogn), or ~nlogn (worse case).
- Is it possible to do better?
- What would we have to do to argue the point either way?

THE SORTING LOWER BOUND

THE BEST POSSIBLE ALGORITHM

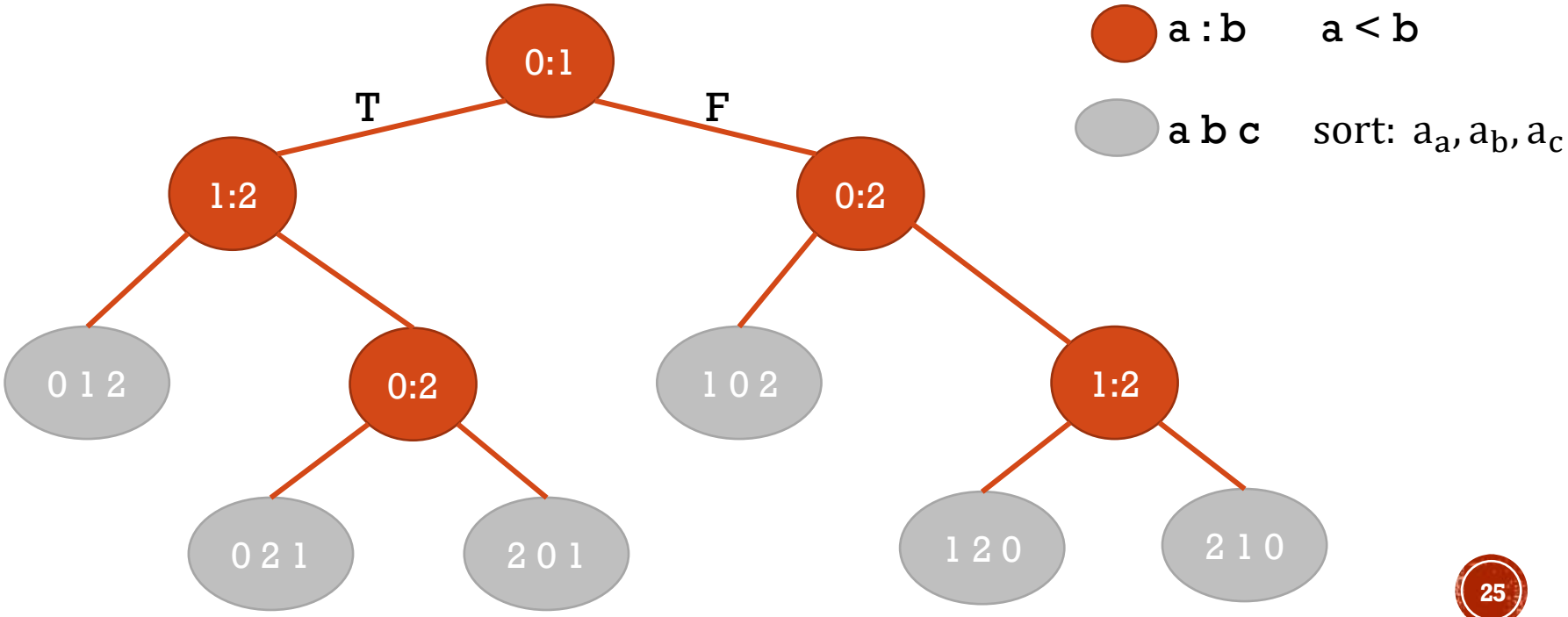
- For a moment, let's pretend we have no clue what the *lower bound* for sorting is.
- By the consistency of computation, there must exist some lower bound for sorting.
- To find it, we must propose a general way to model the ordered nature of many elements (call it an array *a*): comparisons.
- We call this the ***decision tree*** model of sorting.
- Our goal is to find out how many comparisons it takes to explore the ***search space*** of the sorting problem

- Investigation (proof):
  - To start, lets make our life a little easier with an assumption:
    - Assume *a*'s elements are distinct. WLOG: easy to add/remove distinctness.

DECISION TREES

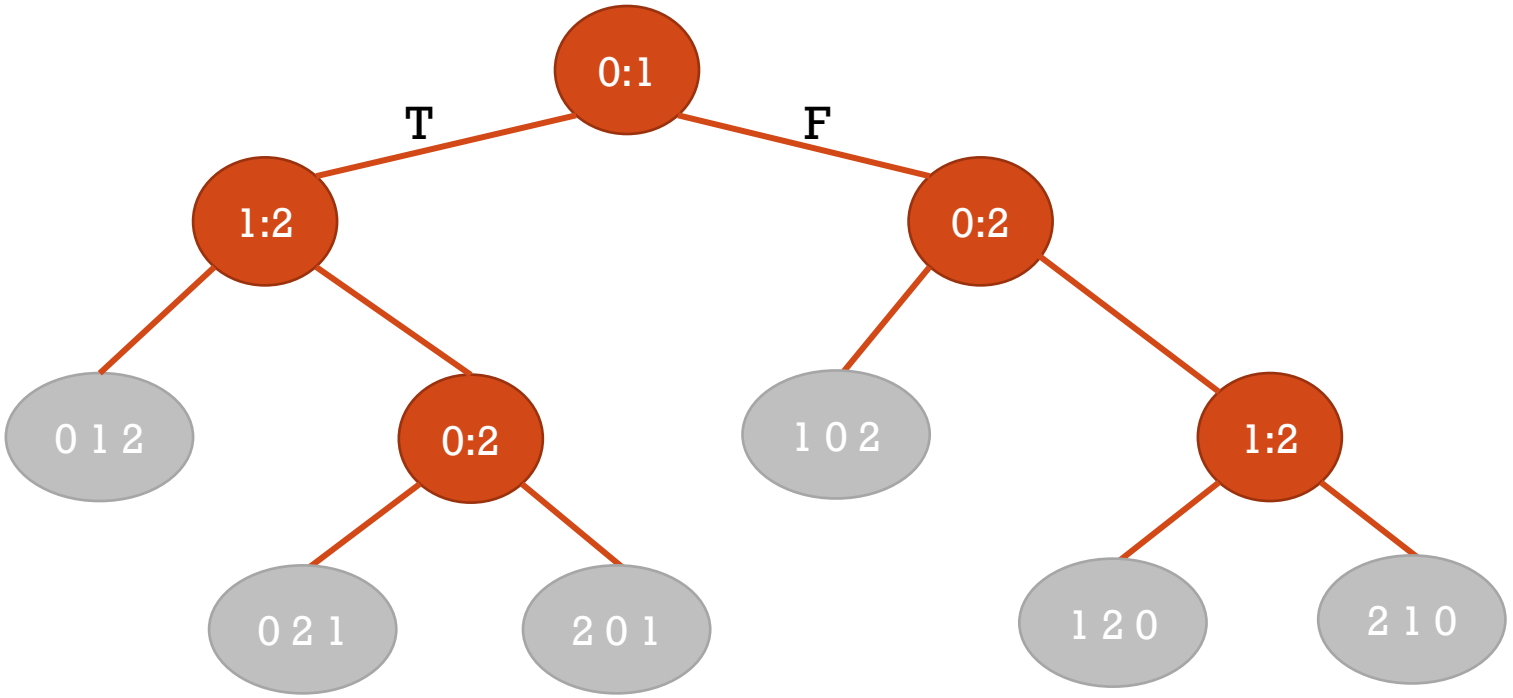
The key! Brings us from assumptions (comparison based) to structure w/ properties.

- Let us represent the decision process that finds a sorted order of elements with a binary tree of comparisons.
  - We'll work with an array of length n=3 for illustrative purposes.



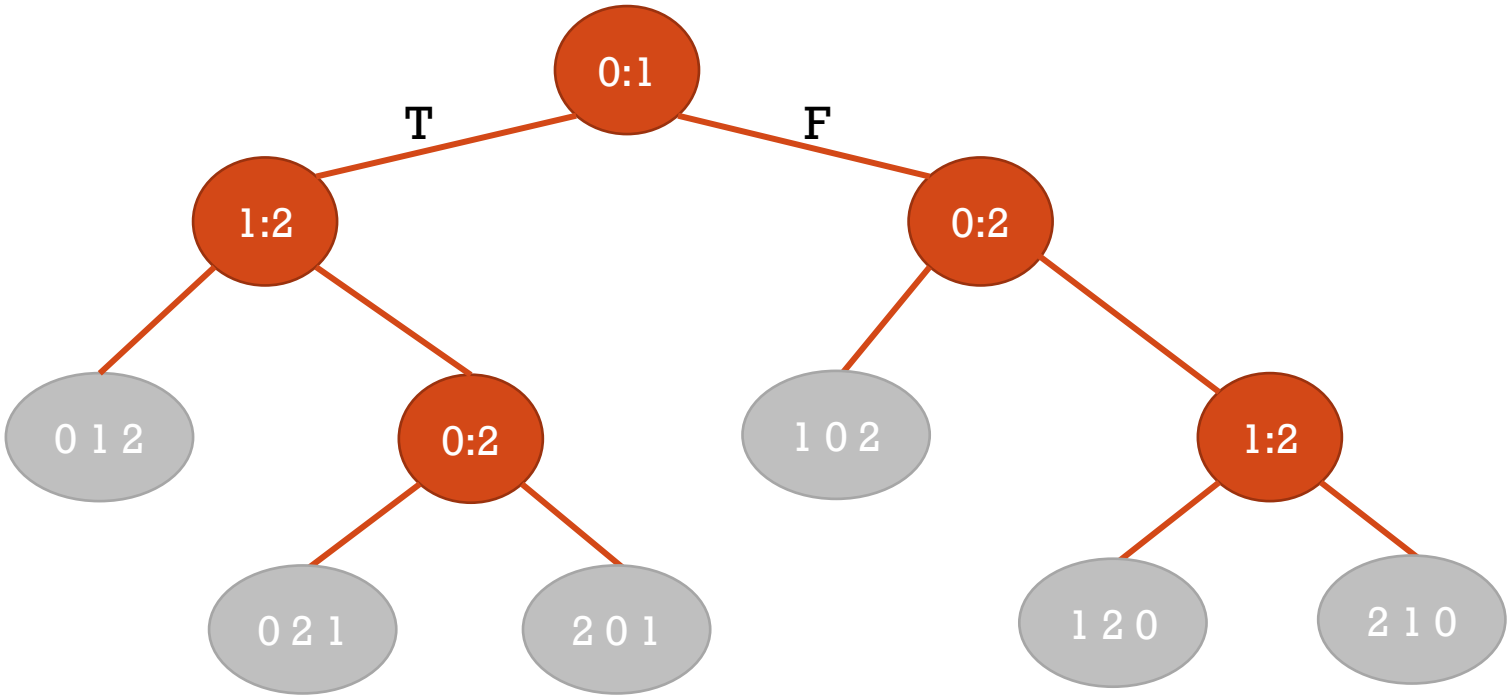
LOWER BOUND FOR SORTING

- For an array of N elements, how many leaves must the tree have?

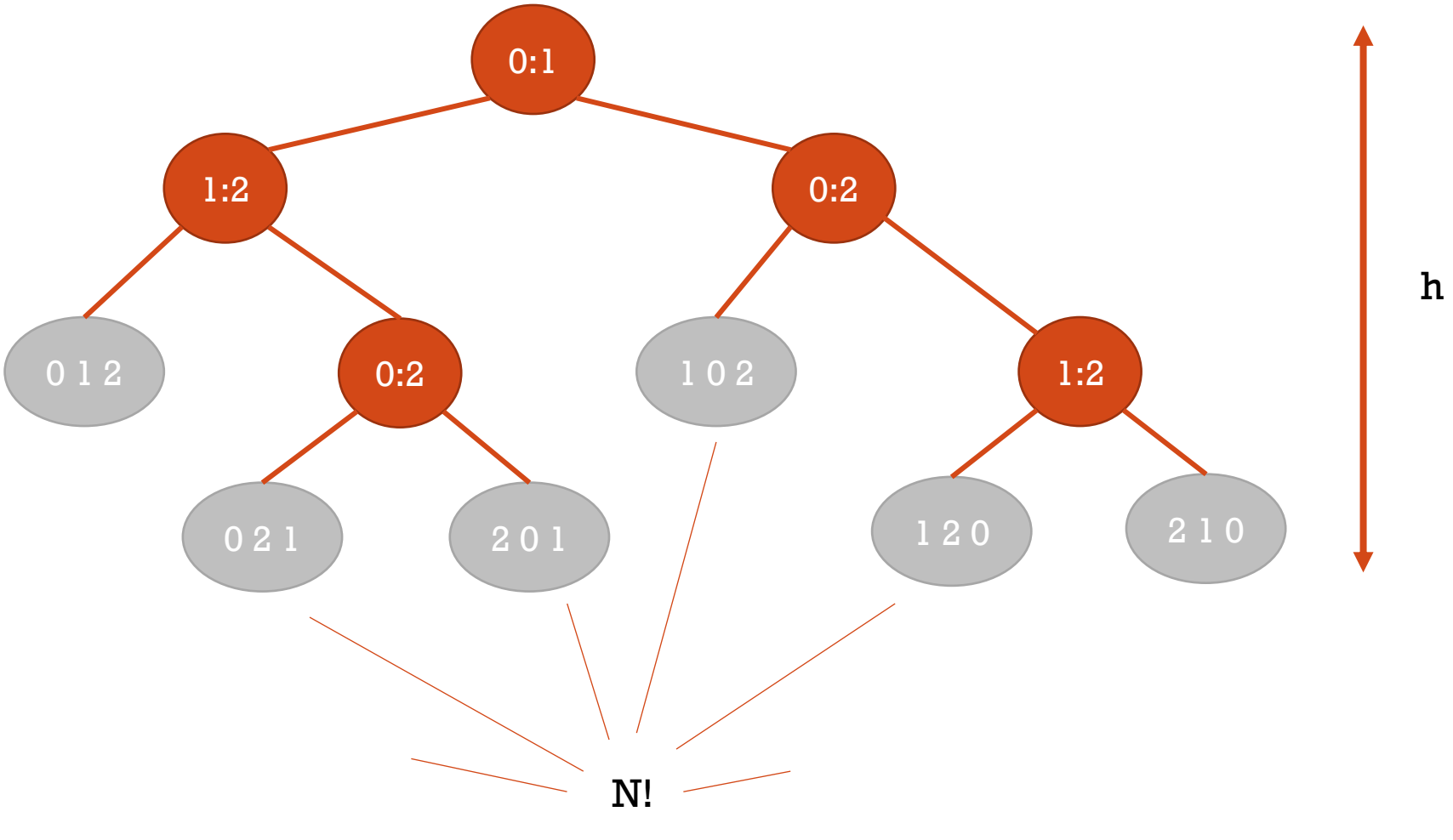


LOWER BOUND FOR SORTING

- Observation: the number of nodes on a path from the root to a leaf is number of comparisons needed to sort.
- What then is the path that gives the largest number of comparisons?
- Does the length of this path have a name?



LOWER BOUND FOR SORTING



PUTTING IT TOGETHER

- Subgoal: to characterize h based on what we know about the leaves.
- From a couple of slides ago we get:
  $N! \leq \#\ of\ leaves$
- Since we structured our knowledge as a binary tree, we get to use the properties of binary trees in our proof. Specially, that a tree of height h can have no more than 2^h leaves:
- $\#\ of\ leaves \leq 2^h$
- Combining them, we get:
  $N! \leq \#\ of\ leaves \leq 2^h$
- Now need to solve for h:
  $N! \leq 2^h$
  $\lg(N!) \leq h$ (take lg)
  $Nlg(N) \leq h$ (via Stirling's approximation: $\lg(N!) \sim Nlg(N)$)

Notice that we have no idea how many leaves there are – we have used the value only symbolically.

h is the height of our tree, and the number of comparisons needed.

Thus, any sorting algorithm needs at least $Nlg(N)$ compares, so: Ω(NlgN).

THE RESULT

- Merge is an asymptotically optimal compare-based sorting algorithm.

- Notice that our only restriction is "comparison-based".
- There is no requirement for programming model…
- The proof is on structure of the problem itself!

- As long as compares are the optimal way to sort, one cannot build **any** computer or any write **any** algorithm to do general sorting in less than nlogn time.