Lecture #1: Sorting

COSC 3020: Algorithms and Data Structures

Lars Kotthoff[1]
larsko@uwyo.edu

[1] with material from various sources

Outline

- Insertion Sort
- Heapsort
- Mergesort
- Quicksort
- Theoretical Bounds
- Additional Assumptions: Linear-Time Sort

Learning Goals

- Describe, apply, and compare various sorting algorithms.
- Analyze the complexity of these sorting algorithms.
- Explain the difference between the complexity of a problem (sorting) and the complexity of a particular algorithm for solving that problem.

Do try this at home

- https://visualgo.net/bn/sorting
- https://www.toptal.com/developers/sorting-algorithms

How to Measure Sorting Algorithms

- Computational complexity (a.k.a. runtime)
  - Worst case
  - Average case
  - Best case

    How often is the input sorted, reverse sorted, or "almost" sorted (k swaps from sorted where $k \ll n$)?
- Stability: What happens to elements with identical keys?
- Memory Usage: How much extra memory is used?

Insertion Sort

Insertion Sort

```
function insertionSort(arr) {
  for(var i = 1; i < arr.length; i++) {
    // Invariant: the elements in arr[0..i-1] are in sorted
       order.
    var val = arr[i];
    var j;
    for(j = i; j > 0 && arr[j-1] > val; j--) {
      arr[j] = arr[j-1];
    }
    arr[j] = val;
  }
}
```

Proving a Loop Invariant

Induction variable: number of times through the loop.

Base case: Prove the invariant true before the loop starts.

Induction hypothesis: Assume the invariant holds just before beginning some (unspecified) iteration.

Inductive step: Prove the invariant holds at the end of that iteration for the next iteration.

Extra bit (not part of proof): Make sure the loop will eventually end!

Insertion Sort

```
for(var i = 1; i < arr.length; i++) {
  // Invariant: the elements in arr[0..i-1] are in sorted
     order.
  var val = arr[i];
  var j;
  for(j = i; j > 0 && arr[j-1] > val; j--) {
    arr[j] = arr[j-1];
  }
  arr[j] = val;
}
```

Base case (at the start of the $(i = 1)$ iteration): `arr[0..0]` only has one element; so, it's always in sorted order.

Insertion Sort

```
for(var i = 1; i < arr.length; i++) {
  // Invariant: the elements in arr[0..i-1] are in sorted
     order.
  var val = arr[i];
  var j;
  for(j = i; j > 0 && arr[j-1] > val; j--) {
    arr[j] = arr[j-1];
  }
  arr[j] = val;
}
```

Induction Hypothesis: At the start of iteration i of the loop, `arr[0..i-1]` are in sorted order.

Insertion Sort

```
for(var i = 1; i < arr.length; i++) {
  // Invariant: the elements in arr[0..i-1] are in sorted
     order.
  var val = arr[i];
  var j;
  for(j = i; j > 0 && arr[j-1] > val; j--) {
    arr[j] = arr[j-1];
  }
  arr[j] = val;
}
```

Inductive Step: The inner loop places `val=arr[i]` at the appropriate index $j < i$ by shifting elements of `arr[0..i-1]` that are larger than `val` one position to the right. As `arr[0..i-1]` is sorted (by IH), `arr[0..i]` ends up in sorted order and the invariant holds at the start of the next iteration $(i = i + 1)$.

Insertion Sort

```
for(var i = 1; i < arr.length; i++) {
  // Invariant: the elements in arr[0..i-1] are in sorted
     order.
  var val = arr[i];
  var j;
  for(j = i; j > 0 && arr[j-1] > val; j--) {
    arr[j] = arr[j-1];
  }
  arr[j] = val;
}
```

Loop termination: The loop ends after `length-1` iterations. When it ends, we were about to enter the (`i=length`) iteration.

Therefore, by the newly proven invariant, when the loop ends, `arr[0..length-1]` is in sorted order, which means `arr` is sorted!

Insertion Sort: Running Time

At the start of iteration i, the first i elements in the array are sorted, and we insert the $(i+1)$st element into its proper place.

i $=$ 3

Worst case:

4	5	6	**3**	...	...	...	...	...	...

Insertion Sort: Running Time

At the start of iteration i, the first i elements in the array are sorted, and we insert the $(i+1)$st element into its proper place.

i = 3

Worst case:

4	5	6	**3**	...	...	...	...	...	...

$\Theta(n)$ per iteration $\rightarrow \Theta(n^2)$

Insertion Sort: Running Time

At the start of iteration i, the first i elements in the array are sorted, and we insert the $(i+1)$st element into its proper place.

i $=$ 3

Worst case:

4	5	6	**3**	...	...	...	...	...	...

$\Theta(n)$ per iteration $\rightarrow \Theta(n^2)$

Best case:

0	1	2	**3**	...	...	...	...	...	...

Insertion Sort: Running Time

At the start of iteration i, the first i elements in the array are sorted, and we insert the $(i+1)$st element into its proper place.

i = 3

Worst case:

4	5	6	**3**	...	...	...	...	...	...

$\Theta(n)$ per iteration $\rightarrow \Theta(n^2)$

Best case:

0	1	2	**3**	...	...	...	...	...	...

$\Theta(1)$ per iteration $\rightarrow \Theta(n)$

Insertion Sort: Running Time

At the start of iteration i, the first i elements in the array are sorted, and we insert the $(i+1)$st element into its proper place.

i = 3

Worst case:

4	5	6	**3**	...	...	...	...	...	...

$\Theta(n)$ per iteration $\rightarrow$ $\Theta(n^2)$

Best case:

0	1	2	**3**	...	...	...	...	...	...

$\Theta(1)$ per iteration $\rightarrow$ $\Theta(n)$

Average case? $\rightarrow$ lab

Insertion Sort: Stability & Memory

At the start of iteration `i`, the first `i` elements in the array are sorted, and we insert the `(i+1)`st element into its proper place.

Easily made stable:

"proper place" is largest `j` such that `arr[j-1]` $\leq$ new element.

Memory:

Sorting is done in-place, meaning only a constant number of extra memory locations are used.

Heapsort

Heapsort

1. Heapify input array.
2. Repeat n times: Perform `deleteMin`

Worst case: $\Theta(n \log n)$

[2] Schaffer and Sedgewick, The Analysis of Heapsort, *J. Algorithms* **15** (1993), 76–100.

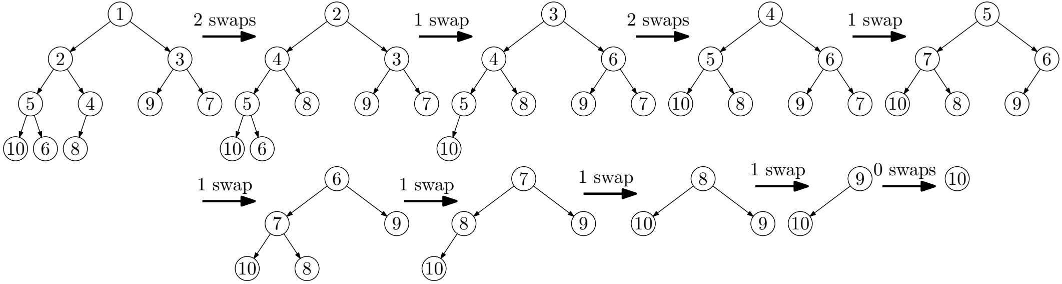
Heapsort

1. Heapify input array.
2. Repeat n times: Perform `deleteMin`

Worst case: $\Theta(n \log n)$

Best case[2]: $\Theta(n \log n)$

1	2	3	5	4	9	7	10	6	8



[2]Schaffer and Sedgewick, The Analysis of Heapsort, *J. Algorithms* **15** (1993), 76–100.

Heapsort: Stability & Memory

1. Heapify input array.
2. Repeat n times: Perform `deleteMin`

Not stable:

Hack: Use index in input array to break comparison ties.

Memory:

- in-place. You can avoid using another array by storing the result of the ith `deleteMin` in heap location $n - i$, except the array is then sorted in reverse order, so use a Max-Heap (and `deleteMax`).
- Far-apart array accesses ruin cache performance.

Mergesort

Mergesort

Mergesort is a "divide and conquer" algorithm.

1. If the array has 0 or 1 elements, it's sorted. Stop.
2. Split the array into two approximately equal-sized halves.
3. Sort each half recursively (using Mergesort).
4. Merge the sorted halves to produce one sorted result:
   - Consider the two halves to be queues.
   - Repeatedly dequeue the smaller of the two front elements (or dequeue the only front element if one queue is empty) and add it to the result.

Mergesort Example

3	-4	7	5	9	6	2	1

Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

3	-4

7	5

Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

3	-4

7	5

3

-4

Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

3	-4

7	5

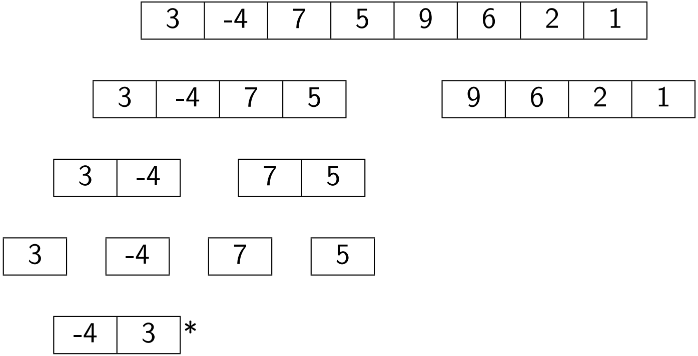
3

-4

-4	3

*

Mergesort Example



Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

3	-4

7	5

3

-4

7

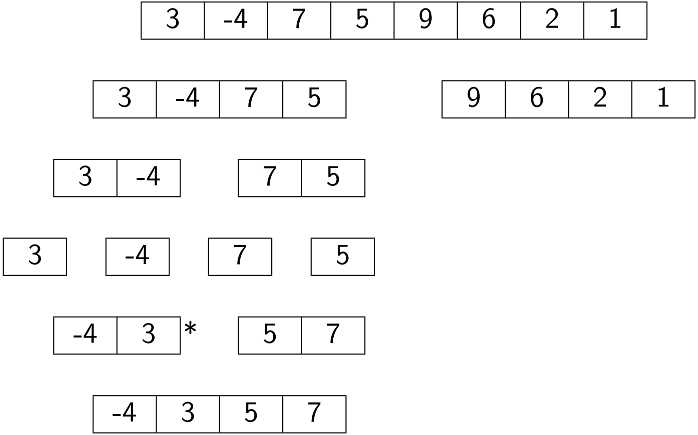
5

-4	3

*

5	7

Mergesort Example



Mergesort Example

3 -4 7 5 9 6 2 1

3 -4 7 5   9 6 2 1

3 -4   7 5   9 6   2 1

3   -4   7   5   9   6   2   1

-4 3 *   5 7   6 9   1 2

-4 3 5 7   1 2 6 9 **

-4 1 2 3 5 6 7 9

Mergesort Code

```
function msort(x, lo, hi, tmp) {
  if(lo >= hi) return;
  var mid = Math.floor((lo+hi)/2);
  msort(x, lo, mid, tmp);
  msort(x, mid+1, hi, tmp);
  merge(x, lo, mid, hi, tmp);
}

function mergesort(x) {
  var tmp = [];
  msort(x, 0, x.length - 1, tmp);
}
```

Merge Code

```
function merge(x, lo, mid, hi, tmp) {
  var a = lo, b = mid + 1;
  for(var k = lo; k <= hi; k++) {
    if(a <= mid && (b > hi || x[a] < x[b])) {
      tmp[k] = x[a++];
    } else {
      tmp[k] = x[b++];
    }
  }
  for(var k = lo; k <= hi; k++) {
    x[k] = tmp[k];
  }
}
```

Sample Merge Steps

```
merge(x, 0, 0, 1, tmp); // step *
```

x :	3	-4	7	5	9	6	2	1
tmp :	-4	3	?	?	?	?	?	?
x :	-4	3	7	5	9	6	2	1

```
merge(x, 4, 5, 7, tmp); // step **
```

x :	-4	3	5	7	6	9	1	2
tmp :	?	?	?	?	1	2	6	9
x :	-4	3	5	7	1	2	6	9

```
merge(x, 0, 3, 7, tmp); // final step
```

Mergesort Running Time

1. If the array has 0 or 1 elements, it's sorted. Stop. $T(1) = 1$
2. Split the array into two approximately equal-sized halves. 1
3. Sort each half recursively (using Mergesort). $2T(n/2)$
4. Merge the sorted halves to produce one sorted result: n
   - Consider the two halves to be queues.
   - Repeatedly dequeue the smaller of the two front elements (or dequeue the only front element if one queue is empty) and add it to the result.

Mergesort Running Time

Recurrence relation:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

Mergesort Running Time

Recurrence relation:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

Solve by substitution:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &\dots \\ &= 2^i T(n/2^i) + in \end{aligned}$$

for $i = \lg n$

$$= nT(1) + n \lg n = n + n \lg n \in \Theta(n \log n)$$

Mergesort: Stability & Memory

Stable:
Dequeue from the left queue if the two front elements are equal.

Memory:
Not easy to implement without using $\Omega(n)$ extra space, so it is not viewed as an in-place sort.

Quicksort

Quicksort (C.A.R. Hoare 1961)

In practice, one of the fastest sorting algorithms (although usually combined with insertion sort for smaller arrays).

1. Pick a pivot

2	-4	6	1	5	-3	3	7

2. Reorder the array such that all elements $<$ pivot are to its left, and all elements $\geq$ pivot are to its right.

-4	1	-3	2	6	5	3	7

left partition pivot right partition

3. Recursively sort each partition.

Base case?

Quicksort (C.A.R. Hoare 1961)

In practice, one of the fastest sorting algorithms (although usually combined with insertion sort for smaller arrays).

1. Pick a pivot

2	-4	6	1	5	-3	3	7

2. Reorder the array such that all elements $<$ pivot are to its left, and all elements $\geq$ pivot are to its right.

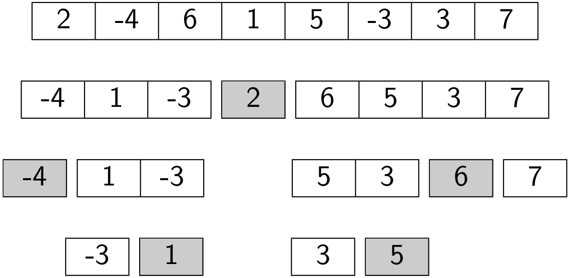
-4	1	-3	2	6	5	3	7

left partition pivot right partition

3. Recursively sort each partition.

Base case? size ≤ 1

Quicksort Visually



Quicksort by Jon Bentley

```
function qsort(x, lo, hi) {
  var i, p;
  if(lo >= hi) return;
  p = lo;
  for(i = lo + 1; i <= hi; i++)
    if(x[i] < x[lo]) swap(x[++p], x[i]);
  swap(x[lo], x[p]);
  qsort(x, lo, p - 1);
  qsort(x, p + 1, hi);
}

function quicksort(x) {
  qsort(x, 0, x.length - 1);
}
```

Quicksort Example (using Bentley's Algorithm)

```
if(x[i] < x[lo]) swap(x[++p], x[i]);
```

lo							hi
2	-4	6	1	5	-3	3	7
p	i						

2	-4	6	1	5	-3	3	7
	p	i					

2	-4	6	1	5	-3	3	7
	p		i				

2	-4	1	6	5	-3	3	7
		p		i			

Quicksort Example (using Bentley's Algorithm)

```
if(x[i] < x[lo]) swap(x[++p], x[i]);
```

lo							hi
2	-4	1	6	5	-3	3	7
		p			i		

2	-4	1	-3	5	6	3	7
			p			i	

2	-4	1	-3	5	6	3	7
			p				i

2	-4	1	-3	5	6	3	7
			p				i

Quicksort Example (using Bentley's Algorithm)

lo							hi
2	-4	1	-3	5	6	3	7
			p				i

```
swap(x[lo], x[p]);
```

lo							hi
-3	-4	1	2	5	6	3	7
			p				i

```
qsort(x, lo, p-1);
qsort(x, p+1, hi);
```

-4	-3	1	2	3	5	6	7

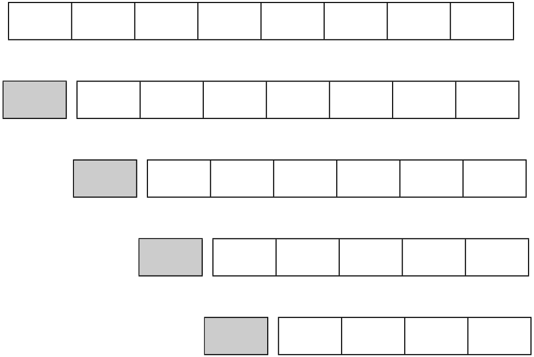
Quicksort: Running Time

Running time is proportional to number of comparisons.

1. Pick a pivot.
   Zero comparisons
2. Reorder (partition) array around the pivot.
   Quicksort compares each element to the pivot.
   $n - 1$ comparisons
3. Recursively sort each partition.
   Depends on the size of the partitions.

- ▷ If the partitions have size $n/2$ (or any constant fraction of n), the runtime is $\Theta(n \log n)$ (like Mergesort).
- ▷ In the worst case, however, we might create partitions with sizes 0 and $n - 1$.

Quicksort Visually: Worst case



Quicksort: Worst Case

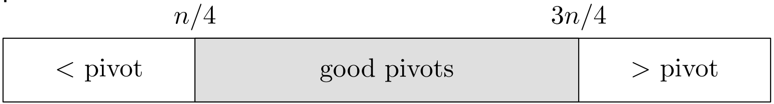
If this happens at every partition, quicksort makes $n-1$ comparisons in the first partition and recurses on a problem of size 0 and size $n-1$:

$$\begin{aligned} T(n) &= (n-1) + T(0) + T(n-1) = (n-1) + T(n-1) \\ &= (n-1) + (n-2) + T(n-2) \\ &\vdots \\ &= \sum_{i=1}^{n-1} i = (n-1)(n-2)/2 \end{aligned}$$

This is $\Theta(n^2)$ comparisons.

Quicksort: Average Case (Intuition)

- On an average input (i.e. random order of n items), our chosen pivot is equally likely to be the ith smallest for any $i = 1, 2, \ldots, n$.
- With probability 1/2, our pivot will be from the middle $n/2$ elements – a good pivot.



- Any good pivot creates two partitions of size at most $3n/4$.
- We expect to pick one good pivot every two tries.
- Expected number of splits is at most $2\log_{4/3} n \in \Theta(\log n)$.
- $\Theta(n \log n)$ total comparisons.

Choosing the Pivot

- first element
- random element
- median of three
- median of nine
- dual pivots

Quicksort: Stability & Memory

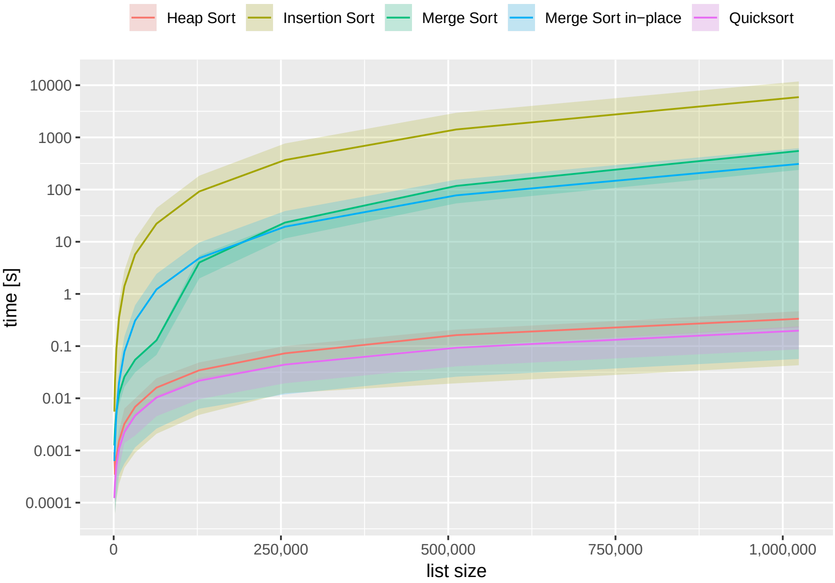
Stable:

Can be made stable, most easily by using more memory.

Memory:

In-place sort.

Compare: Average Case Running Times



Compare: Quick, Merge, Heap, and Insertion Sort

Running Time

	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Best case:	Insertion	Quick, Merge, Heap	
Average case:		Quick, Merge, Heap	Insertion
Worst case:		Merge, Heap	Quick, Insertion
"Real" data:	Quick $<$ Merge, Heap $<$ Insertion		

Some Quick/Merge implementations use Insertion on small arrays (base cases).
Some results depend on the implementation! For example, an initial check whether the last element of the left subarray is less than the first of the right can make Merge's best case linear.

Compare: Quick, Merge, Heap, and Insertion Sort

Stability

Stable (easy):	Insertion, Merge (prefer the left of the two sorted subarrays on ties)
Stable (with effort):	Quick
Unstable:	Heap

Memory use

- Insertion, Heap, Quick $<$ Merge

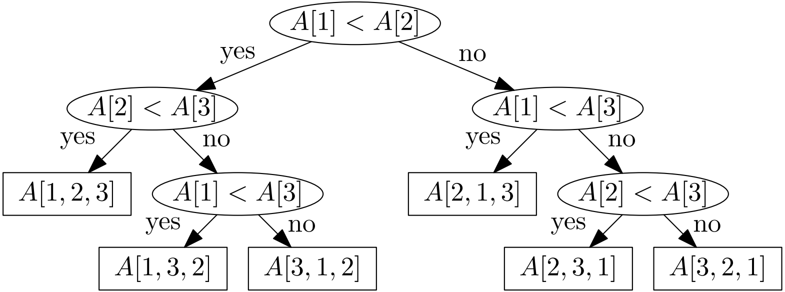
Theoretical Bounds

Complexity of the Sorting Problem

- complexity of a problem is the complexity of the best possible algorithm for that problem
- only considered comparison-based algorithms – compare two elements in constant time
- do not assume anything beyond comparison, e.g. that elements are numbers and we can perform arithmetic operations
- insertion, heap, merge, and quicksort are comparison-based

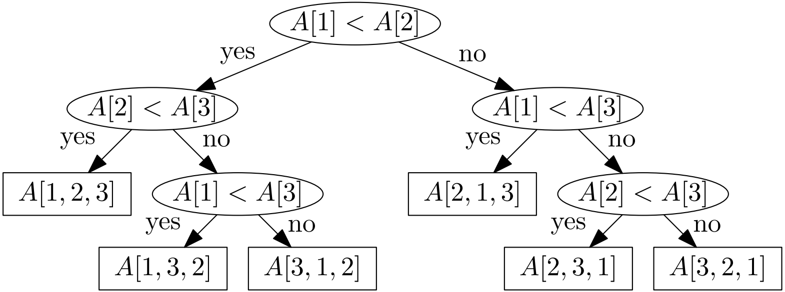
Comparison-based algorithms using a Decision Tree model

Each comparison is a "choice point" in the algorithm: the algorithm can do one thing if the comparison is true and another if false. So, the algorithm is like a binary tree...



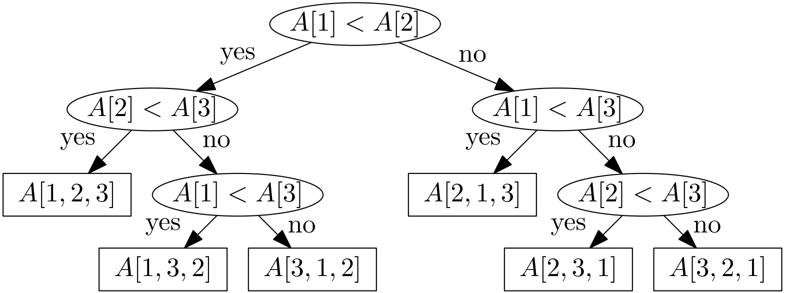
Complexity of the Sorting Problem

- ▷ This is the decision tree representation of Insertion Sort on inputs of size $n = 3$.
- ▷ Each leaf outputs the input array in some particular order. For example, $arr[3, 1, 2]$ means output $arr[3]$, $arr[1]$, $arr[2]$.



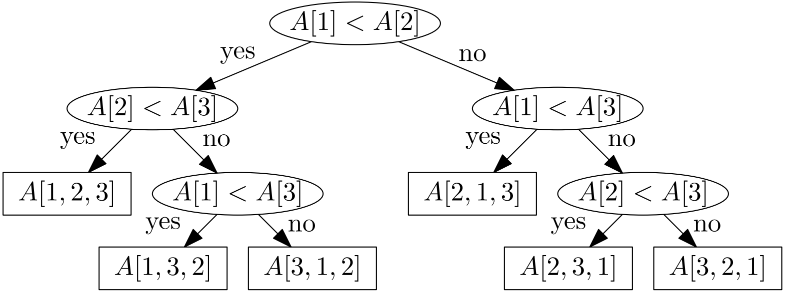
Complexity of the Sorting Problem

- There are $n!$ possible output orderings of an input array of size n.
- There must be a leaf for each one, otherwise the algorithm fails to sort.
    - For example, if leaf $arr[2,3,1]$ doesn't exist then the algorithm cannot sort $[\texttt{cat}, \texttt{ant}, \texttt{bee}]$.



Complexity of the Sorting Problem

- The number of leaves is at least $n!$.
- The height of the decision tree is at least $\lceil \lg(n!) \rceil$.
- The number of comparisons made *in the worst case* is at least $\lceil \lg(n!) \rceil$.
- This is true for any comparison-based sorting algorithm so the complexity of the sorting problem is $\Omega(n \log n)$ ($\lg(n!) = n \lg n$ according to Stirling's approximation).



Additional Assumptions: Linear-Time Sort

Additional Assumptions

- previously: only assume we can compare elements
- in many cases, we know more (e.g. we're sorting positive numbers)
- can exploit this to get more efficient algorithms

Bucket Sort

- distinct items known (e.g. sorting reviews by stars)
- create n buckets for n distinct items (e.g. hash table)
- scan array once, putting each element into its bucket
- time complexity $\Theta(n)$

Radix Sort

- elements composed of smaller parts, e.g. numbers of digits, strings of characters
- sort elements by number of parts, parts at each position
- iterated bucket sort
- time complexity $\Theta(wn)$