CS2020
Data Structures and Algorithms

Welcome!

Quiz 1

Friday, February 12

Sorting, Part I

- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

Sorting, Part II

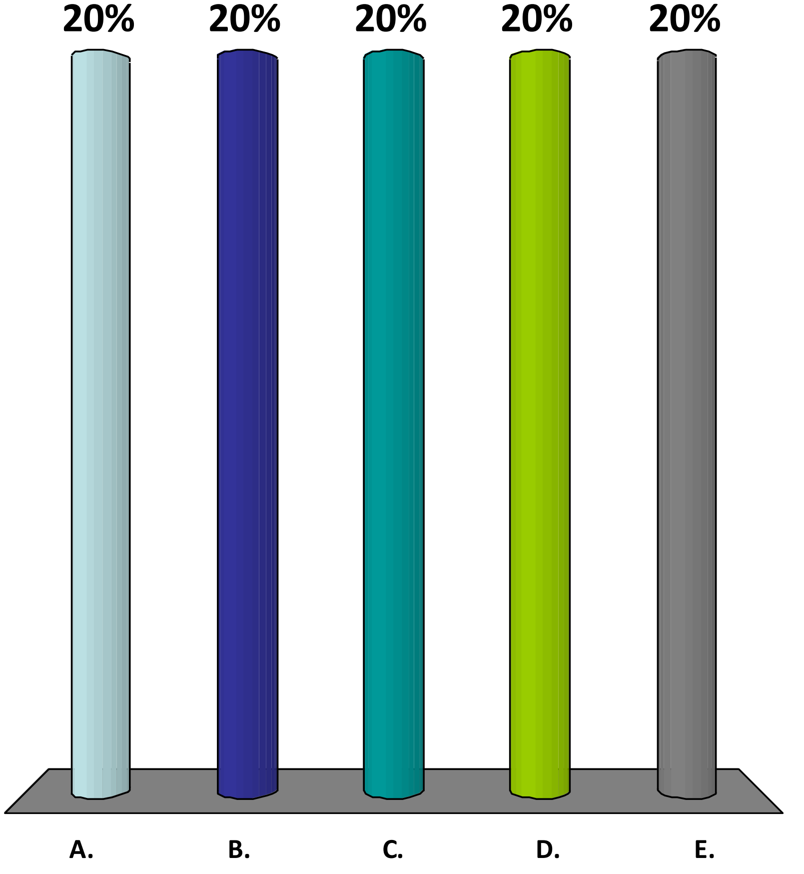
QuickSort

- Divide-and-Conquer
- Partitioning
- Duplicates
- Choosing a pivot

Probability Theory

You bet on red for 10 rounds and all lost because of all black. Next round you should:

A. Bet on black
B. Bet on red
C. Bet on both red and black
D. Let's just leave
E. Bananananana!



Probability Theory

Flipping a coin:

- $Pr(heads) = 1/2$
- $Pr(tails) = 1/2$

Coin flips are independent:

- $Pr(heads, heads) = 1/2 * 1/2 = 1/4$
- $Pr(heads, tails, heads) = 1/2 * 1/2 * 1/2 = 1/8$

Probability Theory

Flipping a coin:

- $\Pr(\text{heads}) = ½$
- $\Pr(\text{tails}) = ½$

Set of uniform events ($e_1, e_2, e_3, \ldots, e_k$):

- $\Pr(e_1) = 1/k$
- $\Pr(e_2) = 1/k$
- …
- $\Pr(e_k) = 1/k$

Probability Theory

Events A, B:

- $\Pr(A)$ = probability of A
- $\Pr(B)$ = probability of B

Pairwise independence:

- A and B are independent
  (e.g., unrelated random coin flips)
- $\Pr(A \text{ and } B) = \Pr(A)\Pr(B)$

Probability Theory

Expected value:

- Weighted average

Example: random variable **X** has two outcomes:

- Pr(**X** = 12) = ¼
- Pr(**X** = 60) = ¾

Expected value of **X**:

$E[X] = (¼)12 + (¾)60 = 48$

Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

In two coin flips: I <u>expect</u> one heads.

Probability Theory

Define variable **X**:

- **X** = number of heads in two coin flips

In two coin flips: I <u>expect</u> one heads.

– Pr(heads, heads) = ¼	2 heads * ¼	=	½
– Pr(heads, tails) = ¼	1 heads * ¼	=	¼
– Pr(tails, heads) = ¼	1 heads * ¼	=	¼
– Pr(tails, tails) = ¼	0 heads * ¼	=	0
			1

Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

In two coin flips: I <u>expect</u> one heads.

- If you repeated the experiment many times, on average after two coin flips, you will have one heads.

Goal: calculate <u>expected</u> time of QuickSort

Probability Theory

Set of outcomes for $X = (e_1, e_2, e_3, \ldots, e_k)$:

- $\Pr(e_1) = p_1$
- $\Pr(e_2) = p_2$
- …
- $\Pr(e_k) = p_k$

Expected outcome:

$$E[X] = e_1 p_1 + e_2 p_2 + \ldots + e_k p_k$$

Those were the days

Expected Value

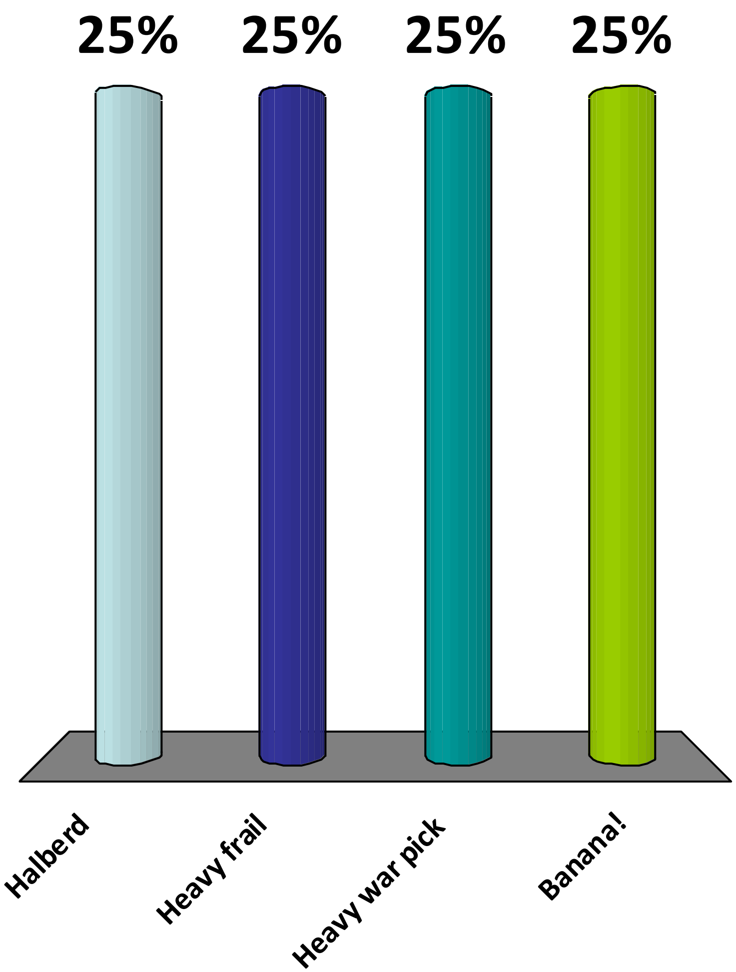
- Which one will you pick?

Name	Source	Prof	Damage
Falchion	PHB	+3	2d4
Glaive	PHB	+2	2d4
Greataxe	PHB	+2	1d12
Greatsword	PHB	+3	1d10
Halberd	PHB	+2	1d10
Heavy flail	PHB	+2	2d6
Heavy war pick	AV	+2	1d12

1. Halberd; 2. Longbow; 3. Handaxe; 4. Short sword; 5. Shortbow; 6. Longsword; 7. Maul; 8. Greataxe; 9. War pick; 10. Bastard sword; 11. Warhammer; 12. Flail; 13. Battleaxe; 14. Throwing hammer; 15. Scimitar; 16. Glaive

Which weapon(s) is/are better in the long run in terms on damage only?

A. Halberd
B. Heavy frail
C. Heavy war pick
D. Banana!



Response Counter

Probability Theory

Linearity of Expectation:

- $E[A + B] = E[A] + E[B]$

Example:

- X = # heads in 2 coin flips
- Y = # heads in 2 coin flips
- $X + Y$ = # heads in 4 coin flips

$E[X+Y] = E[X] + E[Y] = 1 + 1 = 2$

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

E[X]= expected number of flips to get one head

Example: X = 7

T T T T T T H

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

E[X]= Pr(heads after 1 flip)*1 +
Pr(heads after 2 flips)*2 +
Pr(heads after 3 flips)*3 +
Pr(heads after 4 flips)*4 +
…

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

E[X]= Pr(H)*1 +

Pr(T H)*2 +

Pr(T T H)*3 +

Pr(T T T H)*4 +

…

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = $(1 - p)$

How many flips to get at least one head?

$$
\begin{aligned}
\mathbf{E}[X] = {} & p(1) + \\
& (1 - p)(p)(2) + \\
& (1 - p)(1 - p)(p)(3) + \\
& (1 - p)(1 - p)(1 - p)(p)(4) + \\
& \dots
\end{aligned}
$$

Geometric series!

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

$\mathbf{E}[X] = (p)(1) + (1 - p)\,(1 + \mathbf{E}[X])$

How many <u>more</u> flips to get a head?

Idea: If I flip "tails," the expected number of additional flips to get a "heads" is <u>still</u> **E**[X]!!

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = $(1 - p)$

How many flips to get at least one head?

$$\mathbf{E}[X] = (p)(1) + (1 - p)\,(1 + \mathbf{E}[X])$$

$$= p + 1 - p + 1\mathbf{E}[X] - p\mathbf{E}[X]$$

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

$$\mathbf{E}[X] = (p)(1) + (1 - p)\,(1 + \mathbf{E}[X])$$

$$= p + 1 - p + 1\mathbf{E}[X] - p\mathbf{E}[X]$$

$$\mathbf{E}[X] - \mathbf{E}[X] + p\mathbf{E}[X] = 1$$

Probability Theory

Flipping an (unfair) coin:

- $\Pr(\text{heads}) = p$
- $\Pr(\text{tails}) = (1 - p)$

How many flips to get at least one head?

$$\mathbf{E}[X] = (p)(1) + (1 - p)\,(1 + \mathbf{E}[X])$$

$$= p + 1 - p + 1\mathbf{E}[X] - p\mathbf{E}[X]$$

$$p\mathbf{E}[X] = 1$$

$$\mathbf{E}[X] = 1/p$$

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = $(1 - p)$

How many flips to get at least one head?

If $p = ½$, the expected number of flips to get one head equals:

$$\mathbf{E}[X] = 1/p = 1/½ = 2$$

Paranoid QuickSort

```
ParanoidQuickSort(A[1..n], n)
    if (n == 1) then return;
    else
        repeat
            pIndex = random(1, n)
            p = partition(A[1..n], n, pIndex)
        until p > (1/10)n and p < (9/10)

        x = QuickSort(A[1..p–1], p–1)
        y = QuickSort(A[p+1..n], n–p)
```

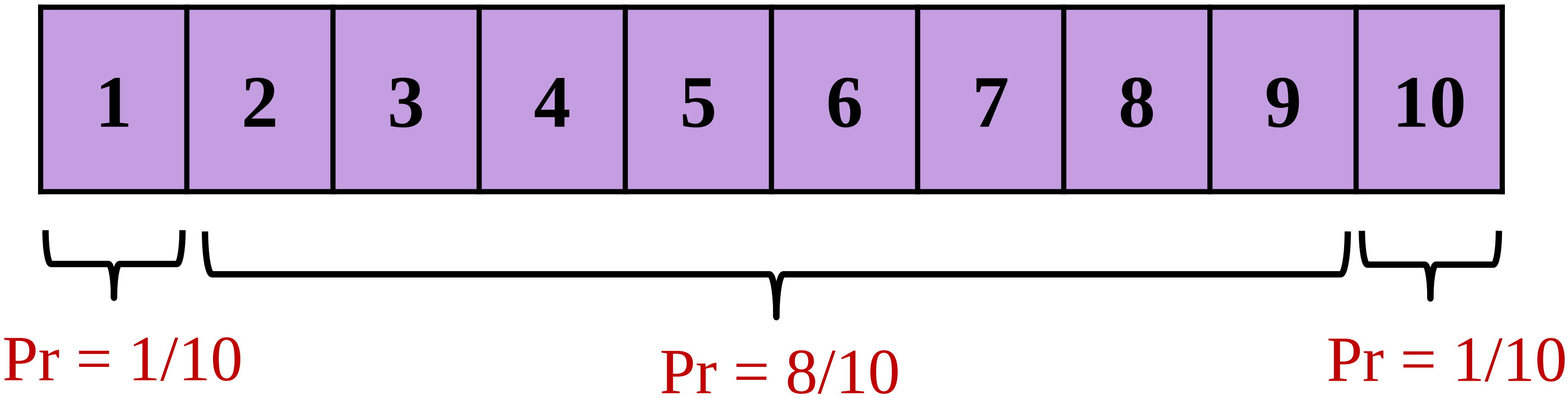
QuickSort Partition

Remember:

A *pivot* is **good** if it divides the array into two pieces, each of which is size at least $n/10$.

Choosing a Good Pivot

Imagine the array divided into 10 pieces:



Probability of a good pivot:

$p = 8/10$

$(1 - p) = 2/10$

Choosing a Good Pivot

Probability of a good pivot:

$$p = 8/10$$

$$(1 - p) = 2/10$$

Expected number of times to repeatedly choose a pivot to achieve a good pivot:

$$\mathbf{E}[\#\text{ choices}] = 1/p = 10/8 < 2$$

Paranoid QuickSort

```
QuickSort(A[1..n], n)
    if (n==1) then return;
    else
        repeat
            pIndex = random(1, n)
            p = partition(A[1..n], n, pIndex)
        until p > n/10 and p < n(9/10)
        x = QuickSort(A[1..p–1], p–1)
        y = QuickSort(A[p+1..n], n–p)
```

Paranoid QuickSort

Key claim:

We only execute the **repeat** loop O(1) times.

Then we know:

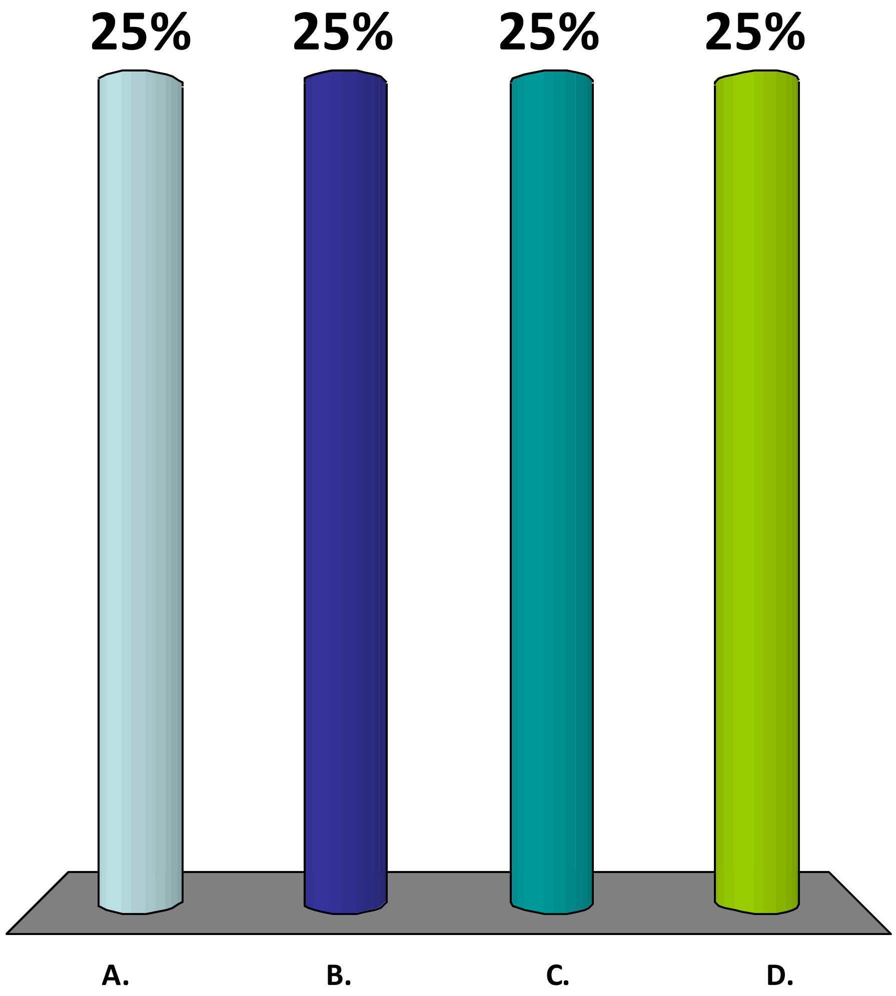
$$\mathbf{E}[\mathrm{T}(n)] = \mathbf{E}[\mathrm{T}(k)]+\mathbf{E}[\mathrm{T}(n-k)]+\mathbf{E}[\text{\# pivot choices}](n)$$

$$<= \mathbf{E}[\mathrm{T}(k)] + \mathrm{E}[\mathrm{T}(n-k)] + 2n$$

$$= O(n \log n)$$

You can approximate π by tossing….

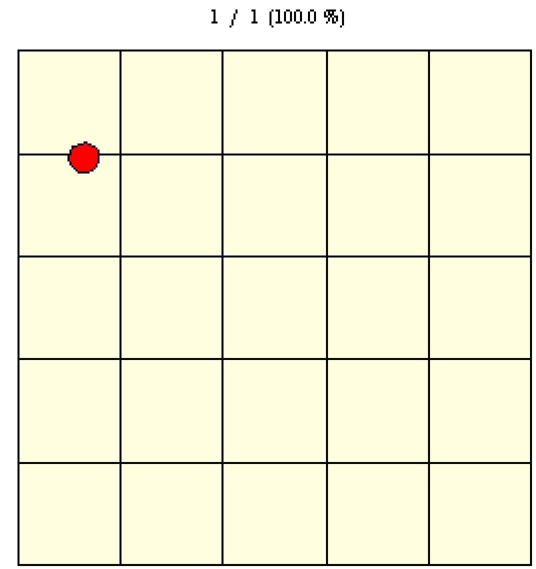
A. Sausages
B. Coins
C. Bananas
D. All of the above

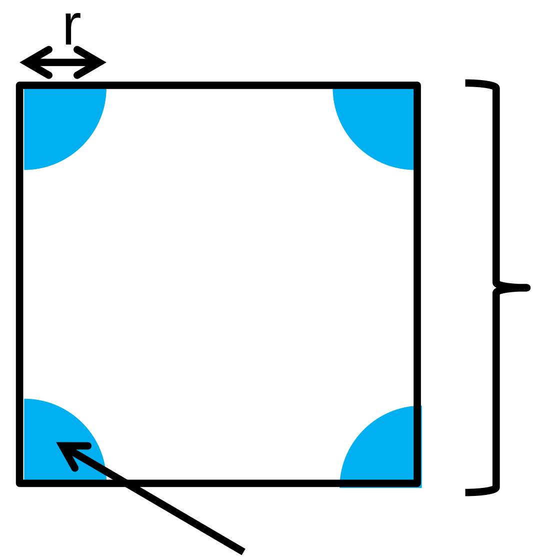


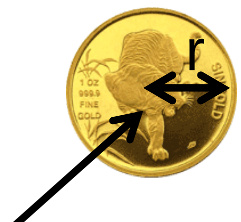
Response Counter

- Probability of coins hitting the crossing
$$= \frac{2\pi r^2}{l^2}$$

$$\pi \approx \frac{l^2 \times \#\,\mathrm{hit}}{2r^2 \times \#\,\mathrm{coins}}$$



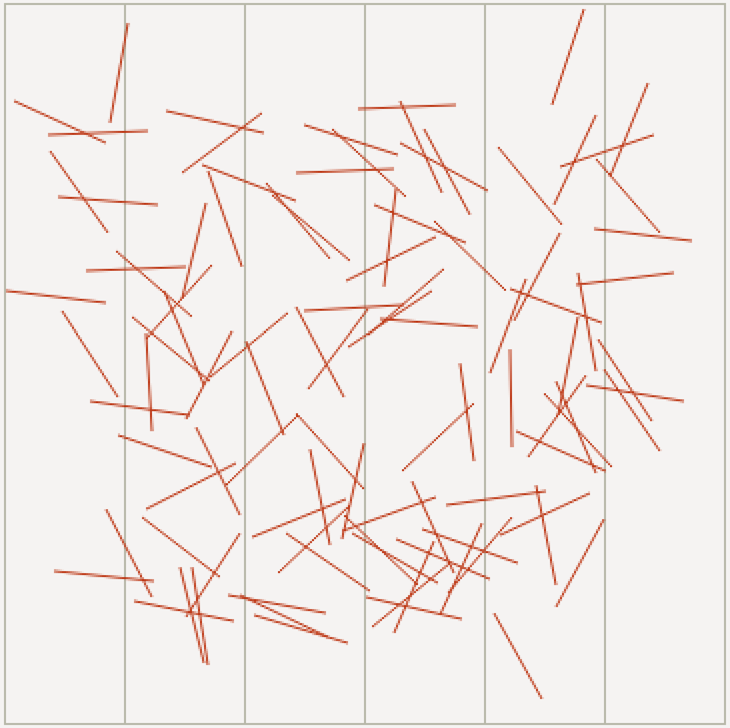




Coin with radius r

Landing zone of coin center

Pickup Sticks & Pi



Number of sticks dropped: 12210
Number that crossed a line: 6399

Your estimate of π: 3.146184

Error: 0.15%

Number of sticks to drop: 100

Click on the board! Each time you click, 100 sticks will be dropped and the number that cross a line will be counted. I dropped 10 to start you off.

Does the estimate get better as you drop more sticks (i.e. does the error get smaller)? How close to 3.1415927… can you get after 100,000 sticks, or 1,000,000?

$$\pi \approx \frac{2 \text{ x stick length x \# sticks tossed}}{\text{distance between lines x \# sticks crossing a line}}$$

http://www.sciencefriday.com/articles/estimate-pi-by-dropping-sticks/
http://www.wikihow.com/Calculate-Pi-by-Throwing-Frozen-Hot-Dogs

QuickSort Tips

- Optimize the partition routine
  - Most important aspect of a good QuickSort is partitioning.
- Choose a pivot carefully (e.g., at random)
  - Bad pivots lead to bad performance.
- Plan for arrays with duplicate values.
  - Equal elements can cause bad performance.

QuickSort Optimizations

For small arrays, use InsertionSort.

- Recursion has overhead.
- QuickSort is slow on small arrays.
- Idea: if the array is small, switch to InsertionSort

Better idea: leave small arrays unsorted and do one big InsertionSort on the whole array

Details:

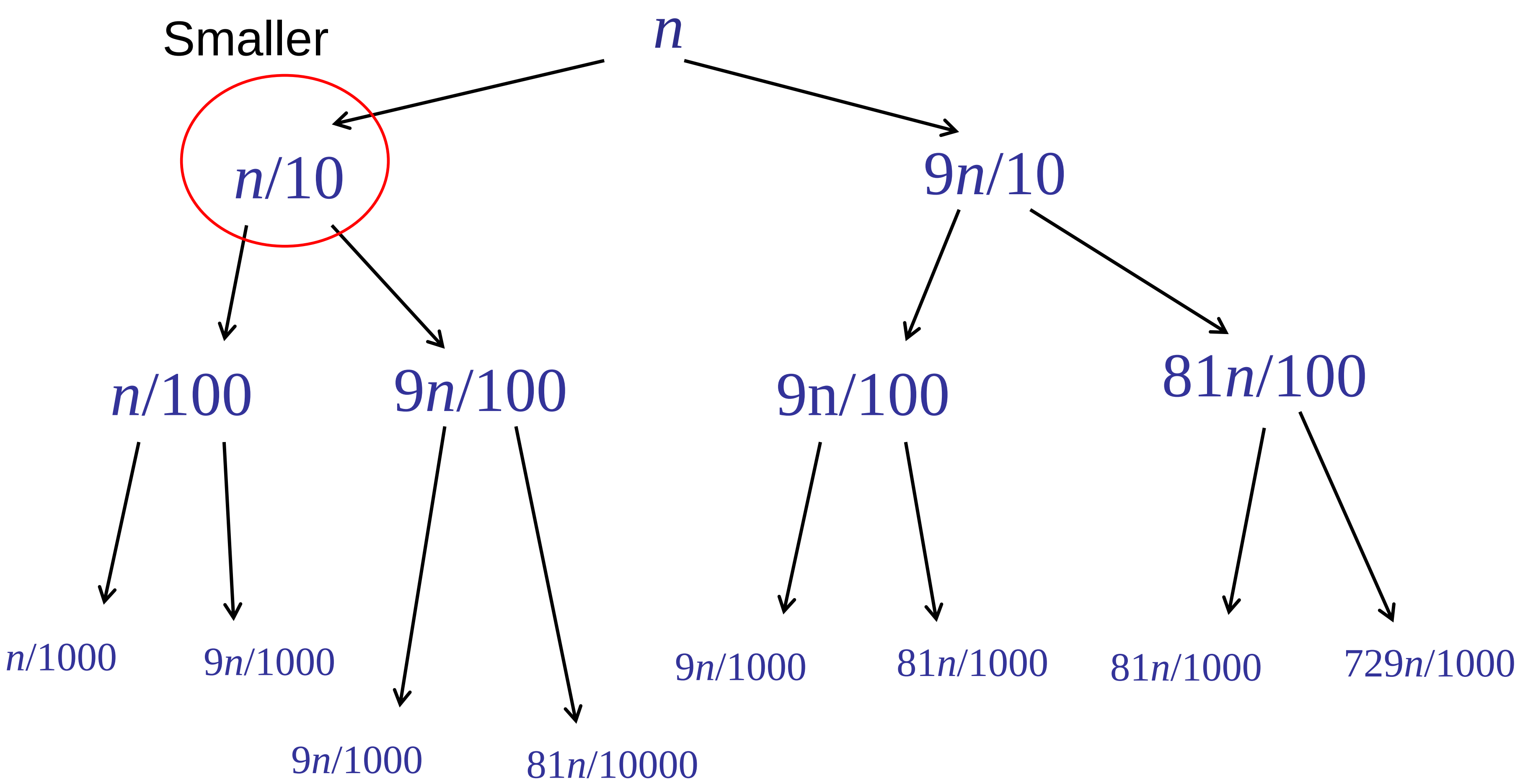
- Once recursion reaches a small array, use InsertionSort (instead of partition/recurse).
- Once recursion reaches 8 elements, hand-code?

QuickSort Optimizations

To save space, recurse into smaller half first.

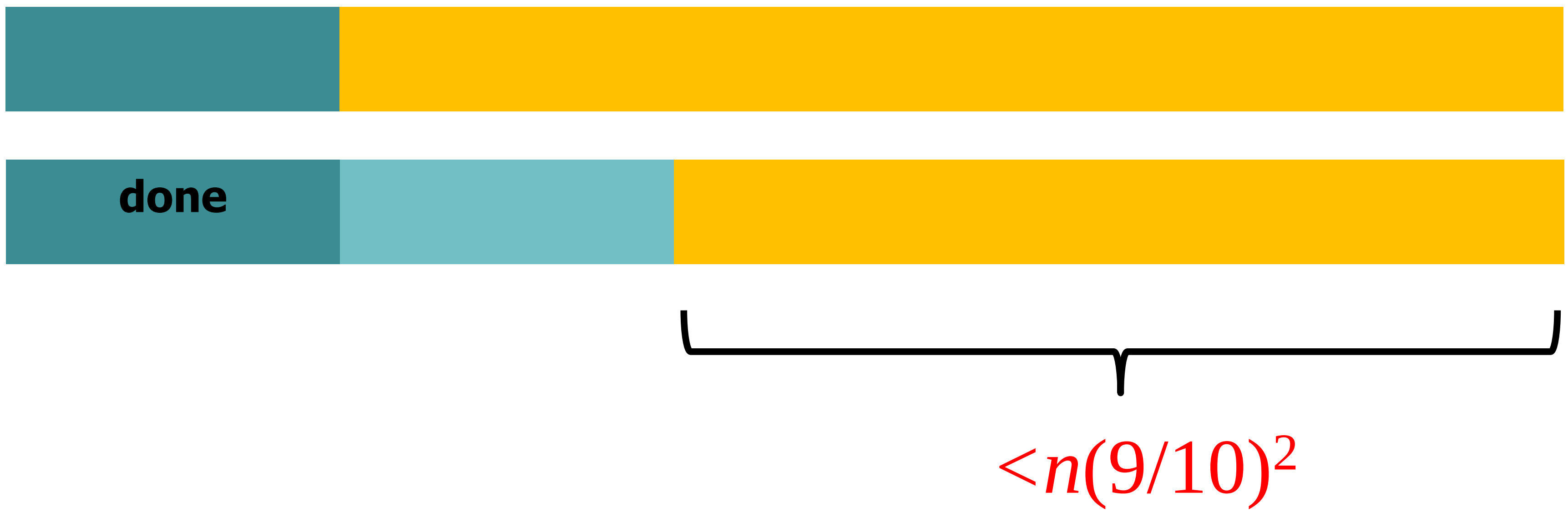
- When you recurse, you have to save everything on the stack during the first recursive call.
- During the second recursive call, you can optimize and not save anything. (See: tail recursion.)
- You can only recurse into the smaller side log(n) times.
- Only need O(log n) extra space in the worst case.

QuickSort Analysis



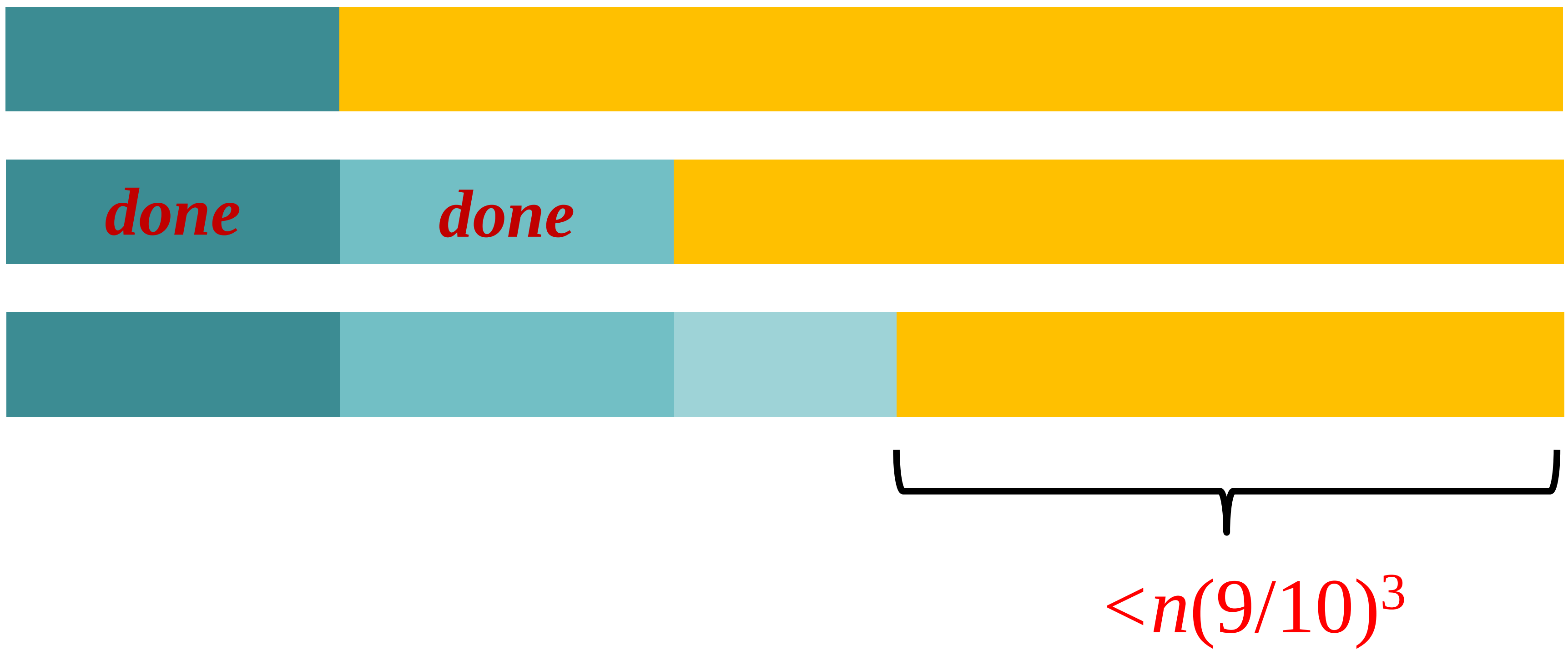
QuickSort Analysis

Assume larger part shrinks by at least 9/10 every iteration:



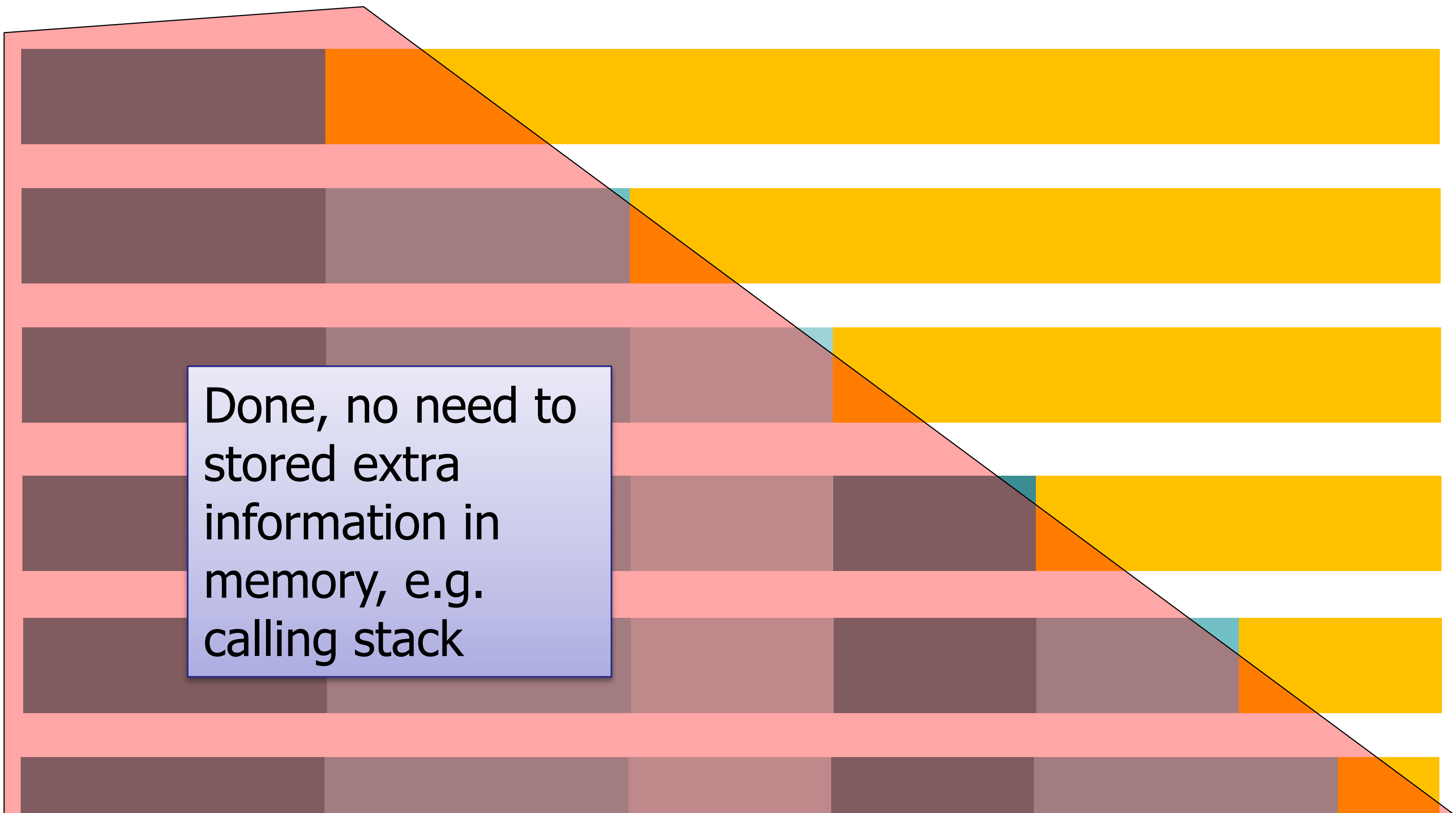
QuickSort Analysis

Assume larger part shrinks by at least 9/10 every iteration:



QuickSort Analysis

Assume larger part shrinks by at least 9/10 every iteration:



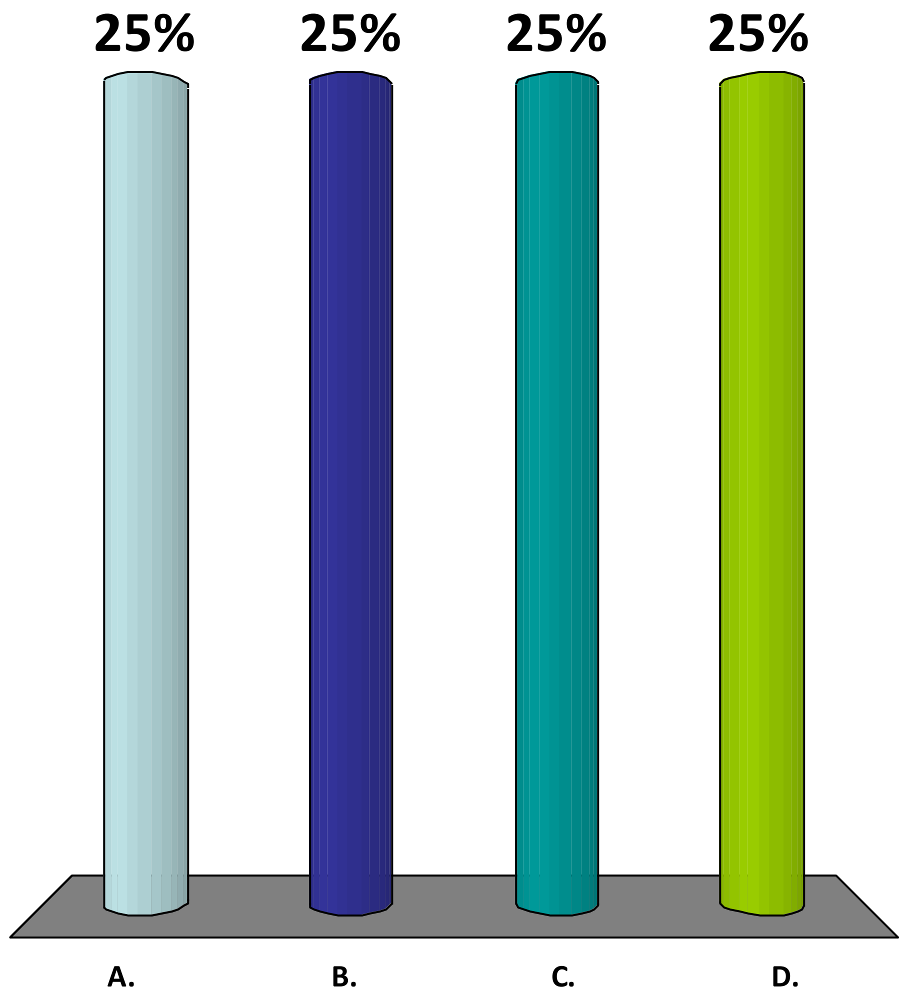
QuickSort Optimizations

Two-pivot Quicksort

- Recently shown that two pivots is faster than one!
- Choose two pivots, partition around both.
- What about three pivots? Four?
- Experiment!

But aren't all these still O(n log n)? What is the reason of being faster?

A. They bluff
B. Some can achieve O(n)
C. It's about the constant c in O
D. They divided into O(n) partition



Response Counter

Summary

QuickSort:

- Algorithm basics: divide-and-conquer
- How to partition an array in $O(n)$ time.
- How to choose a good pivot.
- Paranoid QuickSort.
- Randomized analysis.

Today: Sorting, Part III

- Selection and Order Statistics
  - QuickSelect

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

$\mathbf{x_{10}}$	$\mathbf{x_2}$	$\mathbf{x_4}$	$\mathbf{x_1}$	$\mathbf{x_5}$	$\mathbf{x_3}$	$\mathbf{x_7}$	$\mathbf{x_8}$	$\mathbf{x_9}$	$\mathbf{x_6}$

E.g.: Find the median ($k = n/2$)

Find the 7^{th} element ($k = 7$)

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

x_{10}	x_2	x_4	x_1	x_5	x_3	x_7	x_8	x_9	x_6

Option 1:

- Sort the array.
- Count to element number k.

Running time: O(n log n)

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$	$\mathbf{x}_6$	$\mathbf{x}_7$	$\mathbf{x}_8$	$\mathbf{x}_9$	$\mathbf{x}_{10}$

Option 1:

- Sort the array.
- Count to element number k.

Running time: O(n log n)

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

$\mathbf{x_{10}}$	$\mathbf{x_2}$	$\mathbf{x_4}$	$\mathbf{x_1}$	$\mathbf{x_5}$	$\mathbf{x_3}$	$\mathbf{x_7}$	$\mathbf{x_8}$	$\mathbf{x_9}$	$\mathbf{x_6}$

Option 2:

– Only do the minimum amount of sorting necessary

Order Statistics

Key Idea: partition the array

$\mathbf{x_2}$	$\mathbf{x_4}$	$\mathbf{x_1}$	$\mathbf{x_3}$	$\mathbf{x_5}$	$\mathbf{x_7}$	$\mathbf{x_8}$	$\mathbf{x_9}$	$\mathbf{x_6}$	$\mathbf{x_{10}}$

Now continue searching in the correct half.

E.g.: Partitioned around x_5 so search for x_3 in left half…

Order Statistics

Example: search for 5th element

9	**22**	**13**	**17**	**5**	**3**	**100**	**6**	**19**	**8**

Order Statistics

Example: search for 5th element

9	**22**	**13**	**17**	**5**	**3**	**100**	**6**	**19**	**8**

Random pivot: 17

9	**8**	**13**	**5**	**3**	**6**	**17**	**100**	**19**	**22**
1	2	3	4	5	6	**7**	**8**	**9**	**10**

Order Statistics

Example: search for 5th element

9	**8**	**13**	**5**	**3**	**6**	**17**	**100**	**19**	**22**
1	2	3	4	5	6	**7**	**8**	**9**	**10**

Search for 5th element in left half.

9	**8**	**13**	**5**	**3**	**6**				
1	2	3	4	5	6				

Order Statistics

Example: search for 5th element

9	**8**	**13**	**5**	**3**	**6**				

Random pivot: 8

6	**3**	**5**	**8**	**13**	**9**				
1	2	3	**4**	5	6				

Order Statistics

Example: search for 5th element

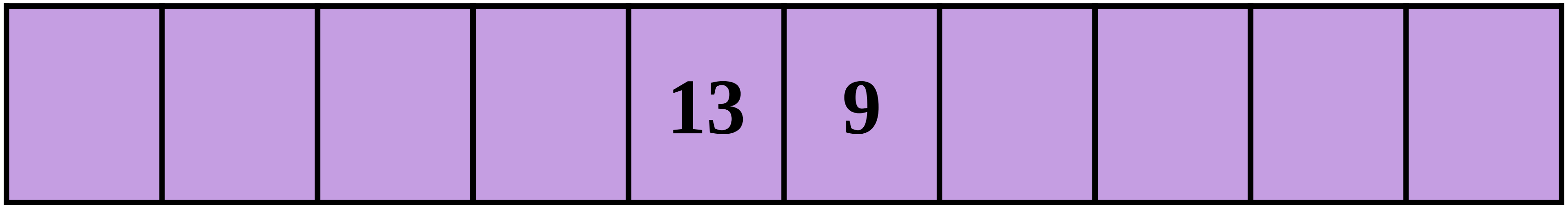
9	**8**	**13**	**5**	**3**	**6**				

Search for: 5 – 4 = 1 in right half

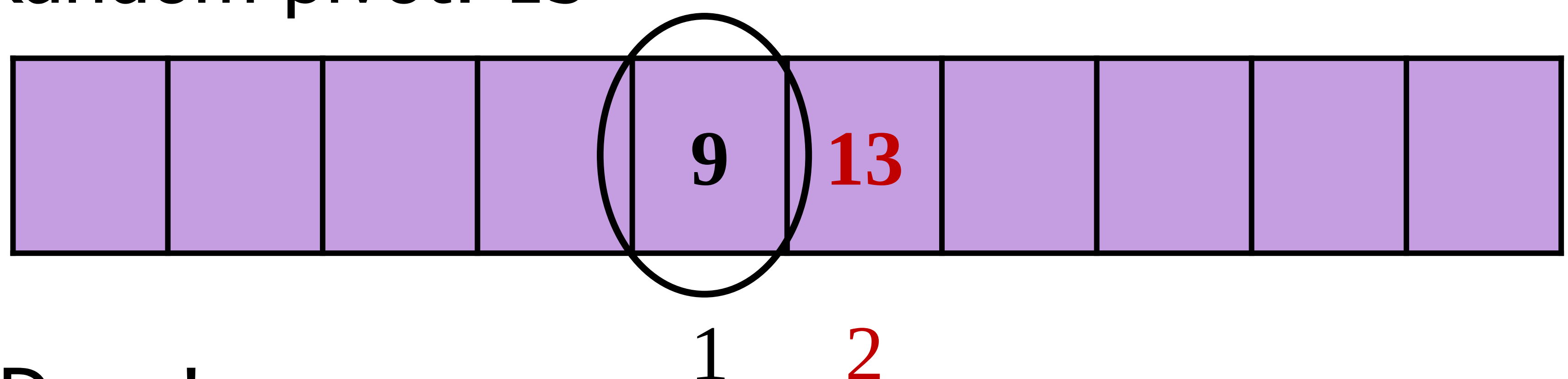
6	**3**	**5**	**8**	**13**	**9**				
1	2	3	**4**	5	6				

Order Statistics

Search for: 5 – 4 = 1 in right half



Random pivot: 13



Done!

Finding the kth smallest element

```
Select(A[1..n], n, k)

    if (n == 1) then return A[1];
    else  Choose random pivot index pIndex.
          p = partition(A[1..n], n, pIndex)
          if (k == p) then return A[p];
          else if (k < p) then
                return Select(A[1..p–1], k)
          else if (k > p) then
                return Select(A[p+1], k – p)
```

Finding the k^{th} smallest element

Key point:

- Only recurse once!
- Why not recurse twice?
  - Does not help---the correct element is on one side.
  - You do not need to sort both sides!
  - Makes it run a lot faster.

Analysis

Paranoid-Select:

- Repeatedly partition until at least $n/10$ in each half of the partition.

```
repeat
        p = partition(A[1..n], n, pIndex)
until (p > n/10) and (p < 9n/10)
```

Analysis

Paranoid-Select:

- Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$\mathbf{E}[T(n)] \le \mathbf{E}[T(9n/10)] + \mathbf{E}[\#\ \text{partitions}](n)$$

cost of partitioning

Analysis

Paranoid-Select:

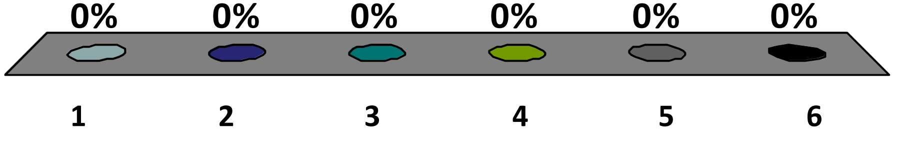
- Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$
\begin{aligned}
\mathbf{E}[\mathrm{T}(\mathrm{n})] &\le \mathbf{E}[\mathrm{T}(9\mathrm{n}/10)] + \mathbf{E}[\#\ \mathrm{partitions}](n) \\
&\le \mathbf{E}[\mathrm{T}(9\mathrm{n}/10)] + 2n
\end{aligned}
$$

The expected running time of paranoid select is in TOTAL:

1. $O(\log n)$
2. $O(n)$
3. $O(n \log n)$
4. $O(n^2)$
5. $O(n^{\log\log(n)})$
6. I have no idea.



Analysis

Paranoid-Select:

- Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$
\begin{aligned}
\mathbf{E}[\mathrm{T(n)}] &\le \mathbf{E}[\#\ \mathrm{partitions}](n) + \mathbf{E}[\mathrm{T(9n/10)}] \\
&\le 2n+\mathbf{E}[\mathrm{T(9n/10)}] \\
&\le 2n+2n\,(9/10)+(9/10)\,\mathbf{E}[\mathrm{T(9n/10)}] \\
&\le 2n+2n\,(9/10)+2n\,(9/10)^2+\ldots\ldots
\end{aligned}
$$

Analysis

Paranoid-Select:

- Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$
\begin{aligned}
\mathbf{E}[\mathrm{T(n)}] &\le \mathbf{E}[\mathrm{T(9n/10)}] + \mathbf{E}[\#\ \text{partitions}](n) \\
&\le \mathbf{E}[\mathrm{T(9n/10)}] + 2n \\
&\le \mathrm{O}(n)
\end{aligned}
$$

Recurrence: $T(n) = T(n/2) + O(n)$

Other problems

Uniqueness testing

- Input:
  - Array A
- Output:
  - Does array A contain any duplicate items? YES/NO?

7	2	4	3	5	6	9	8	1	0

⇨ No duplicates

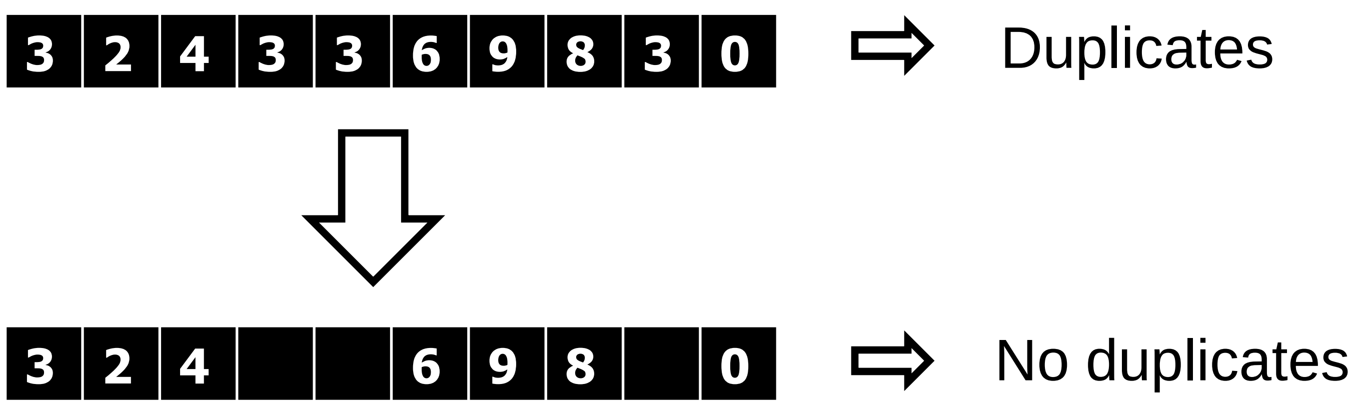
3	2	4	3	3	6	9	8	3	0

⇨ Duplicates

Other problems

Deleting duplicates

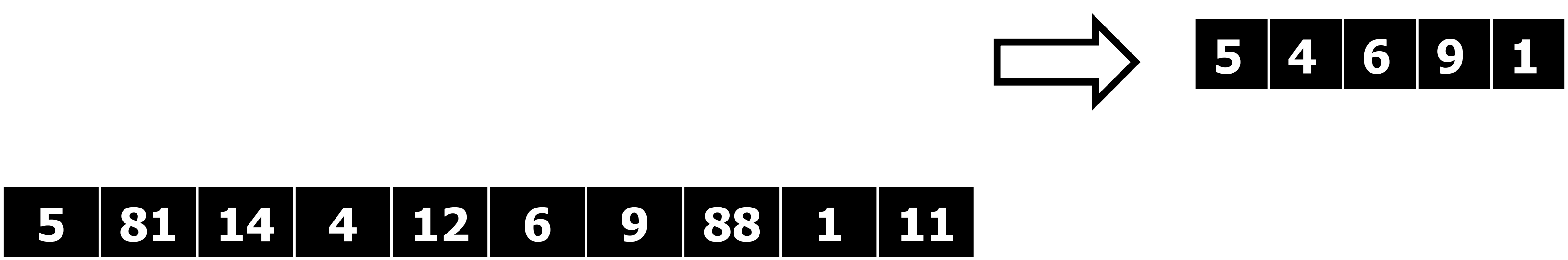
- Input:
  - Array A
- Output:
  - Array A with all the duplicates removed, same order.



Other problems

Set intersection:

- Input:
  - Array A, B
- Output:
  - Array C containing all items in both A and B.



Other problems

Target pair:

- Input:
  - Array A, target
- Output:
  - Two elements (x,y) in A where (x+y) = target.

5	81	14	4	12	6	9	88	1	11

⇨ (4, 9)

target = 13?

Summary

QuickSort: O(n log n)

- Partitioning an array
- Deterministic QuickSort
- Paranoid Quicksort

Order Statistics: O(n)

- Finding the k^{th} smallest element in an array.
- Key idea: partition
- Paranoid Select

Faster Sorting Algorithms

So far:

- Sorting algorithm only do two things:
  - Compare items.
  - Swap/move items that are out of place.
- Require: implements Comparable

What if we can do more?

- What if we are sorting integers?
- What if we are sorting real numbers?

Faster Sorting Algorithms

Counting Sort:

- Linear time, lots of space

Radix Sort:

- Linear time, more efficient space

CS3230

Integer Sorts:

- O(n loglog n) time
- Efficient space
- Complicated and mostly theoretical

CS6234

Auxiliary Data

Typically, databases contain pairs:

[key, data]

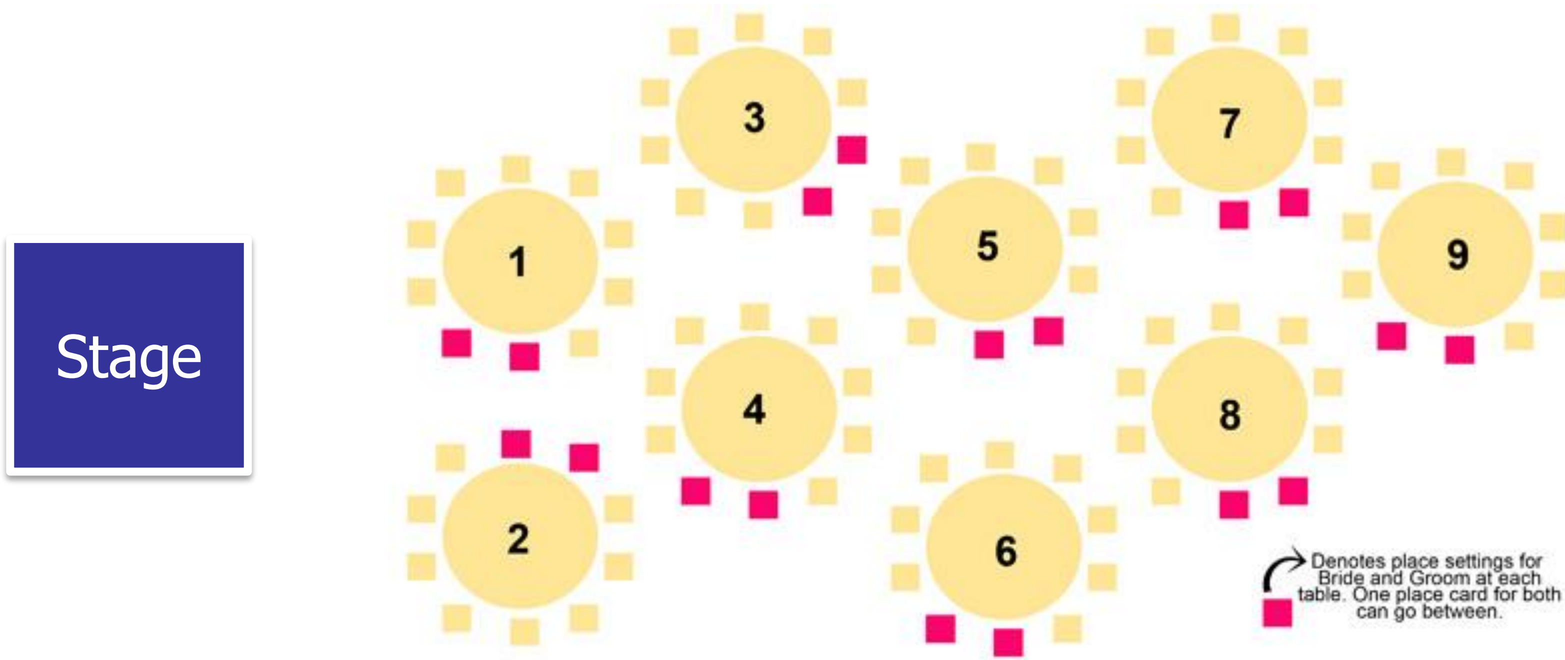
For example:

Age	Name
18	John
18	Mary
19	Bob
32	Sam

Sort by age!

Counting Sort

Order of importance	Guest type	Number
1	Direct family	30
2	Bosses	15
3	Indirect family	50
4	Friends	40
5	People who we don't care….	60



Counting Sort

Key properties:

- Only for sorting integers.
- Assume that all the integers in the input are in the range: [1, k]

Counting Sort

Input: A[1..n]

– Assume $A[j] \in \{1, 2, \ldots, k\}$

Output: B[1..n], sorted

Extra space: C[1..k]

– Initially $C[j] = 0$

Counting Sort

Step 1: Counting

```
for (j=1; j<=n; j++){
    C[A[j]] = C[A[j]] + 1;
}
```

Counting Sort

Step 1:

```
for (j=1; j<=n; j++){
    C[A[j]] = C[A[j]] + 1;
}
```

Example: initial state

A	=	4	1	3	4	3

		1	2	3	4
C	=	0	0	0	0

Counting Sort

Step 1:

```
for (j=1; j<=n; j++){
    C[A[j]] = C[A[j]] + 1;
}
```

Example: (j=1)

A =	4	1	3	4	3

	1	2	3	4
C =	0	0	0	1

Counting Sort

Step 1:

```
for (j=1; j<=n; j++){
    C[A[j]] = C[A[j]] + 1;
}
```

Example: (j=2)

A =	4	1	3	4	3

	1	2	3	4
C =	1	0	0	1

Counting Sort

Step 1:

```
for (j=1; j<=n; j++){
    C[A[j]] = C[A[j]] + 1;
}
```

Example: (j=3)

A =	4	1	3	4	3

	1	2	3	4
C =	1	0	1	1

Counting Sort

Step 1:

```
for (j=1; j<=n; j++){
    C[A[j]] = C[A[j]] + 1;
}
```

Example: (j=4)

A	=	4	1	3	4	3

		1	2	3	4
C	=	1	0	1	2

Counting Sort

Step 1:

```
for (j=1; j<=n; j++){
    C[A[j]] = C[A[j]] + 1;
}
```

Example: (j=5)

A =	4	1	3	4	3

	1	2	3	4
C =	1	0	2	2

Counting Sort

Step 2: Produce Output

B	=					

		1	2	3	4
C	=	1	0	2	2

Counting Sort

Step 2: Produce Output

B =	1				

	1	2	3	4
C =	1	0	2	2

Counting Sort

Step 2: Produce Output

B	=	1	3	3		

		1	2	3	4
C	=	1	0	2	2

Counting Sort

Step 2: Produce Output

B	=	1	3	3	4	4

		1	2	3	4
C	=	1	0	2	2

Counting Sort

Step 2: Produce Output

B	=	1	3	3	4	4

		1	2	3	4
C	=	1	0	2	2

Counting Sort

Are we done?

Is this a good sorting algorithm?

When does this not satisfy our needs?

Auxiliary Data

Typically, databases contain pairs:

[key, data]

Age	Name
32	Sam
18	Mary
19	Bob
18	John

Auxiliary Data

...	18	19	...	32	...
...	2	1	...	1	...

Age	Name
32	Sam
18	Mary
19	Bob
18	John

Auxiliary Data

...	18	19	...	32	...
...	2	1	...	1	...

Age	Name
18	?
18	?
19	?
32	?

Auxiliary Data

...	18	19	...	32	...
...	2	1	...	1	...

Age	Name
32	Sam
18	Mary
19	Bob
18	John

Age	Name
18	?
18	?
19	?
32	?

Use binary search to fill in the missing data?

Counting Sort

Step 1:

```
for (j=1; j<=n; j++){
    C[A[j]] = C[A[j]] + 1;
}
```

A =	4	1	3	4	3

	1	2	3	4
C =	1	0	2	2

Counting Sort

Step 2: Accumulating

```
for (j=1; j<=k; j++){
    C[j] = C[j] + C[j-1];
}
```

Goal: C[j] = #(keys $\le$ j)

Counting Sort

Step 2: Accumulating

```
for (j=2; j<=k; j++){
    C[j] = C[j] + C[j-1];
}
```

Example: initial state

	1	2	3	4
C =	1	0	2	2

Counting Sort

Step 2: Accumulating

```
for (j=2; j<=k; j++){
    C[j] = C[j] + C[j-1];
}
```

Example: (j=2)

	1	2	3	4
C =	1	**1**	2	2

Counting Sort

Step 2: Accumulating

```
for (j=2; j<=k; j++){
    C[j] = C[j] + C[j-1];
}
```

Example: (j=3)

	1	2	3	4
C =	1	1	**3**	2

Counting Sort

Step 2: Accumulating

```
for (j=2; j<=k; j++){
    C[j] = C[j] + C[j-1];
}
```

Example: (j=3)

	1	2	3	4
C =	1	1	3	**5**

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Goal: Copy each input in *A* to output in *B*.

Note: Also copy auxiliary data.

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: initial state

		1	2	3	4
C	=	1	1	3	5

		1	2	3	4	5
A	=	4	1	3	4	3
B	=					

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=5)

		1	2	3	4
C	=	1	1	3	5

		1	2	3	4	5
A	=	4	1	3	4	3
B	=					

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=5)

		1	2	3	4
C	=	1	1	**3**	5

		1	2	3	4	5
A	=	4	1	3	4	**3**
B	=			**3**		

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=5)

		1	2	3	4
C	=	1	1	2	5

		1	2	3	4	5
A	=	4	1	3	4	3
B	=			3		

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=4)

		1	2	3	4
C	=	1	1	2	**5**

		1	2	3	4	5
A	=	4	1	3	**4**	3
B	=			3		

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=4)

		1	2	3	4
C	=	1	1	2	**5**

		1	2	3	4	5
A	=	4	1	3	**4**	3
B	=			3		**4**

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=4)

		1	2	3	4
C	=	1	1	2	**4**

		1	2	3	4	5
A	=	4	1	3	4	3
B	=			3		4

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=3)

		1	2	3	4
C	=	1	1	2	4

		1	2	3	4	5
A	=	4	1	3	4	3
B	=			3		4

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=3)

		1	2	3	4
C	=	1	1	**2**	4

		1	2	3	4	5
A	=	4	1	**3**	4	3
B	=		**3**	3		4

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=3)

		1	2	3	4
C	=	1	1	**1**	4

		1	2	3	4	5
A	=	4	1	3	4	3
B	=		3	3		4

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=2)

		1	2	3	4
C	=	1	1	1	4

		1	2	3	4	5
A	=	4	1	3	4	3
B	=		3	3		4

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=2)

		1	2	3	4
C	=	1	1	1	4

		1	2	3	4	5
A	=	4	1	3	4	3
B	=	1	3	3		4

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=2)

		1	2	3	4
C	=	**0**	1	1	4

		1	2	3	4	5
A	=	4	1	3	4	3
B	=	1	3	3		4

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=1)

		1	2	3	4
C	=	0	1	1	4

		1	2	3	4	5
A	=	4	1	3	4	3
B	=	1	3	3		4

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=1)

		1	2	3	4
C	=	0	1	1	**4**

		1	2	3	4	5
A	=	**4**	1	3	4	3
B	=	1	3	3	**4**	4

Counting Sort

Step 3: Copying Output

```
for (j=n; j>0; j--){
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1;
}
```

Example: (j=1)

		1	2	3	4
C	=	0	1	1	**3**

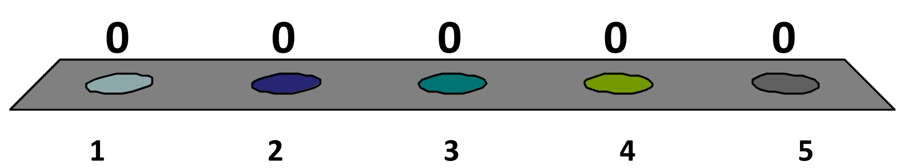
		1	2	3	4	5
A	=	4	1	3	4	3
B	=	1	3	3	4	4

Counting Sort

```
Counting-Sort(A, B, n, k)
   for (j=1; j<=n; j++){
      C[A[j]] = C[A[j]] + 1;
   }
   for (j=2; j<=k; j++){
      C[j] = C[j] + C[j-1];
   }
   for (j=n; j>0; j--){
      B[C[A[j]]] = A[j];
      C[A[j]] = C[A[j]]-1;
   }
```

What is the running time of Counting Sort?

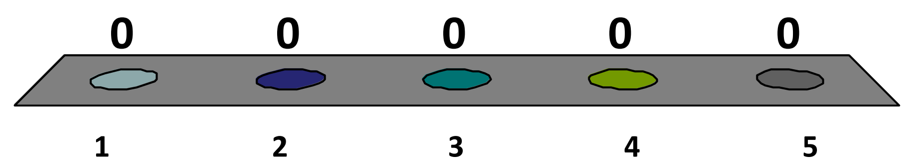
1. $O(k)$
2. $O(n)$
3. $O(n + k)$
4. $O(nk)$
5. $O(n \log k)$



0 of 30

What is the space usage of Counting Sort?

1. $O(k)$
2. $O(n)$
3. $O(n + k)$
4. $O(nk)$
5. $O(n \log k)$



0 of 30

Notes on Counting Sort

Counting Sort is *good* when: ($k \cong n$)

- Time: $O(n)$
- Space: $O(n)$

Counting Sort is *bad* when: ($k >> n$)

- For example: sort a set of 32-bit words?
- No! Space required: $2^{32} > 4$ billion

Auxiliary Data

Typically, databases contain pairs:

[key, data]

For example:

Age	Name
18	John
32	Sam
18	Mary
19	Bob

John precedes Mary.

Stability

We say that a sorting algorithm is <u>stable</u> if:

- Assume $[k_1, data_1]$ precedes $[k_2, data_2]$ in the input.
- Assume $k_1 = k_2$.
- Then: $[k_1, data_1]$ precedes $[k_2, data_2]$ in the output.

A stable algorithms does not change the order of data with equivalent keys.

Stability

Typically, databases contain pairs:

[key, data]

For example:

Age	Name
18	John
18	Mary
19	Bob
32	Sam

John precedes Mary.

Stability

Counting Sort is stable

- When copying data from input to output, it moves from right to left.
- At the same time, it decrements the location count in C.
- Therefore, keys stay in the same order.

Faster Sorting Algorithms

Counting Sort:

- Linear time, lots of space

Radix Sort:

- Linear time, more efficient space

CS3230

Integer Sorts:

- O(n loglog n) time
- Efficient space
- Complicated and mostly theoretical

CS6234

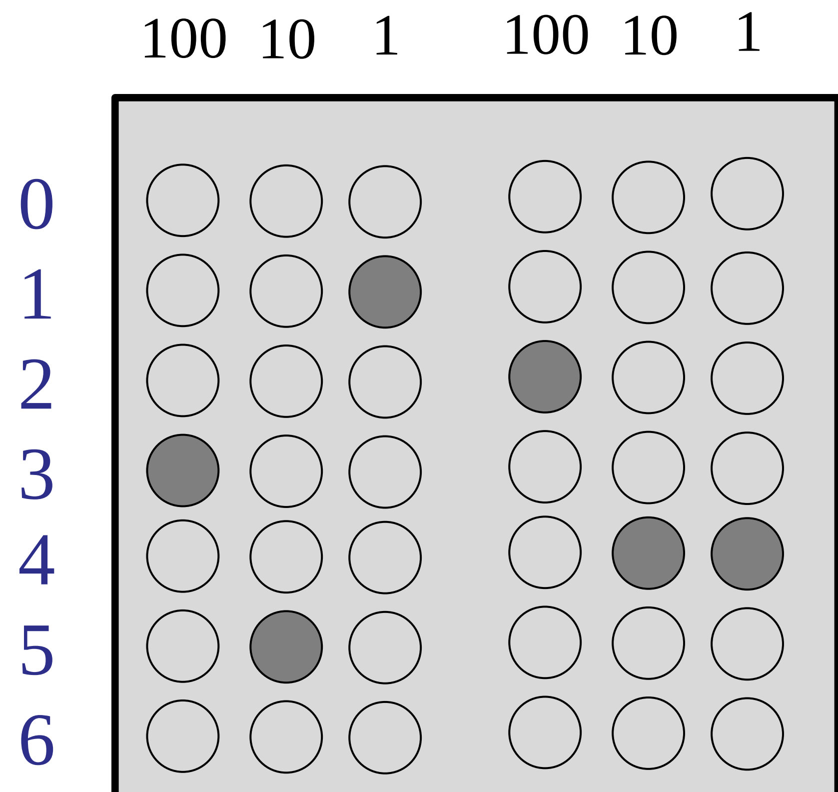
Radix Sort

Digit-by-digit sorting:

- Originated at IBM
- Large numbers of punch-cards to sort
- Each pass through the machine can sort by only one digit.

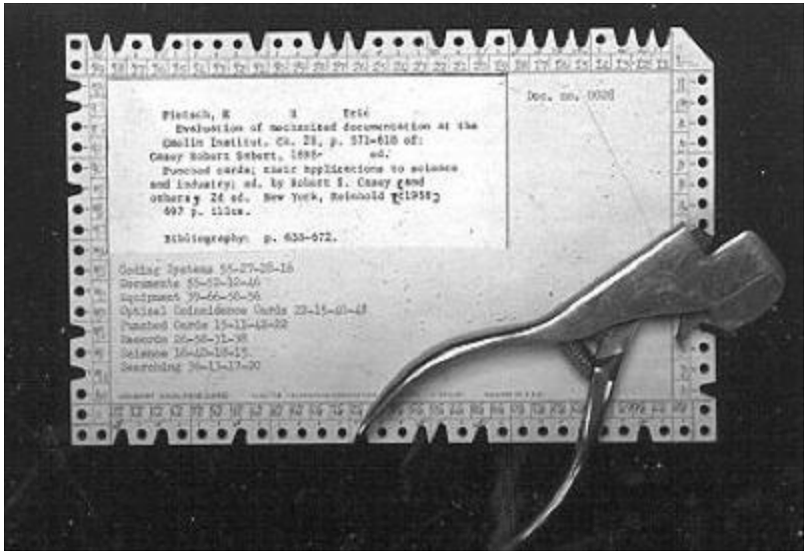
351

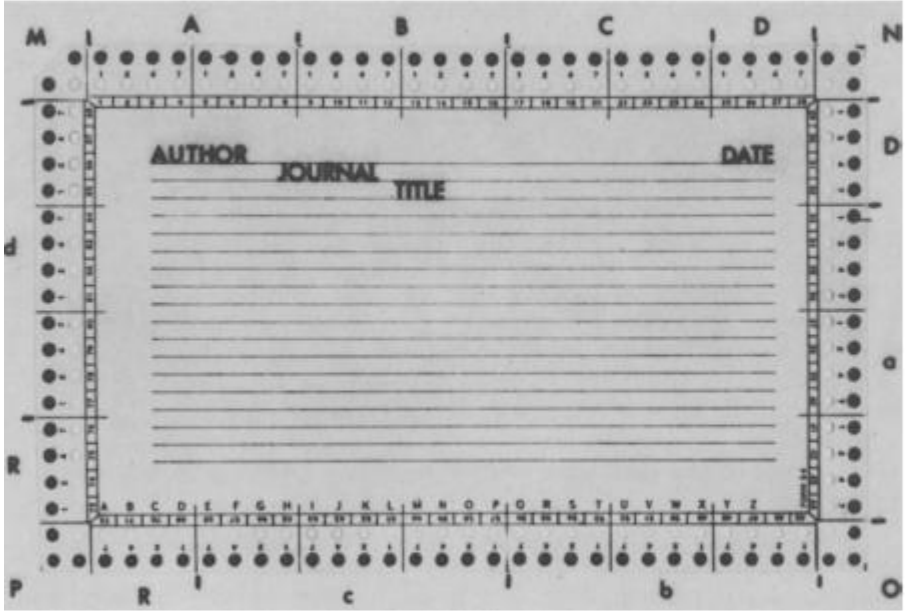
244 =

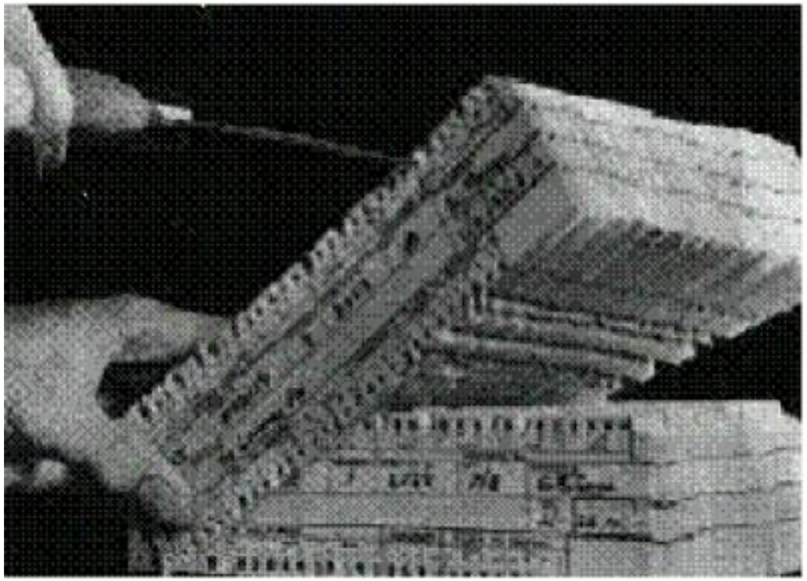


Punch cards

- More precisely
  - Edge-notched card







Radix Sort

Example:

3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5

Radix Sort

Example:

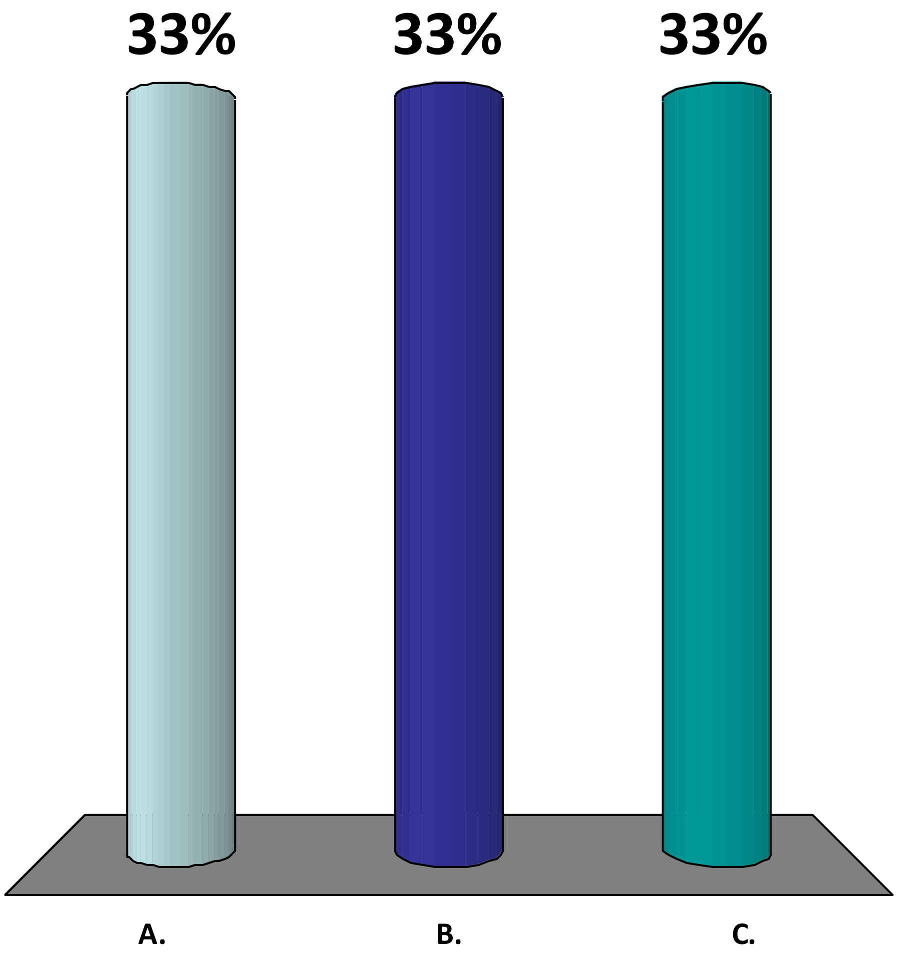
3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5

Which to sort first:

- least significant bit?
- most significant bit?
- doesn't matter?

Better sort by

A. Most Significant bit
B. Least Significant bit
C. Depends



LSB vs MSB

LSB

- Don’t need recursion

MSB

- Handle strings/numbers with various length better
- Can skip less SB, i.e. faster

MSB first?

Example:

3 5 5	3 5 5
4 5 7	3 2 9
6 5 7	4 5 7
8 3 9	4 3 6
4 3 6	6 5 7
7 2 0	7 2 0
3 2 9	8 3 9

MSB first?

Example:

3 5 5	**3** 5 5	3 **2** 9
4 5 7	**3** 2 9	7 **2** 0
6 5 7	**4** 5 7	4 **3** 6
8 3 9	**4** 3 6	8 **3** 9
4 3 6	**6** 5 7	3 **5** 5
7 2 0	**7** 2 0	4 **5** 7
3 2 9	**8** 3 9	6 **5** 7

Problem: have to sort subparts separately for next column. Namely, recursion

Radix Sort: LSB first

Example:

3 5 5	7 2 **0**
4 5 7	3 5 **5**
6 5 7	4 3 **6**
8 3 9	4 5 **7**
4 3 6	6 5 **7**
7 2 0	8 3 **9**
3 2 9	3 2 **9**

First sort 1's column….

Radix Sort: LSB first

Example:

3 5 5	7 2 0	7 **2** 0
4 5 7	3 5 5	3 **2** 9
6 5 7	4 3 6	4 **3** 6
8 3 9	4 5 7	8 **3** 9
4 3 6	6 5 7	3 **5** 5
7 2 0	8 3 9	4 **5** 7
3 2 9	3 2 9	6 **5** 7

Next sort 10's column….

Radix Sort: LSB first

Example:

3 5 5	7 2 0	7 2 0	**3** 2 9
4 5 7	3 5 5	3 2 9	**3** 5 5
6 5 7	4 3 6	4 3 6	**4** 3 6
8 3 9	4 5 7	8 3 9	**4** 5 7
4 3 6	6 5 7	3 5 5	**6** 5 7
7 2 0	8 3 9	4 5 7	**7** 2 0
3 2 9	3 2 9	6 5 7	**8** 3 9

Last sort 100's column….

Radix Sort: LSB first

Example:

3 5 5	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	8 3 9	4 5 7	7 2 0
3 2 9	3 2 9	6 5 7	8 3 9

Key property: use *stable* sort for each column.

Radix Sort

Why does it work?

If 2 elements differ on most significant column *t*, then:

1. Prior to digit *t*, doesn’t matter.
2. At digit *t*, they are put in the right order.
3. After digit *t*, all higher-order digits are the same and since the sort is stable, they stay in the same order.

Radix Sort

Analysis:

- Use **<u>Counting Sort</u>** for each column.
- Sort n <u>words</u> of b bits each.
- Each <u>digit</u> has r bits.
- Each <u>word</u> has b/r digits.

Running time: $$O\left(\frac{b}{r}\left(n+2^r\right)\right)$$

- b/r digits
- For each digit: $O(n + 2^r)$

Radix Sort

Running time: $\mathrm{O}\left(\frac{b}{r}\left(n + 2^r\right)\right)$

- b/r digits
- For each digit: $\mathrm{O}(n + 2^r)$

Extra space: $\mathrm{O}(2^r)$

Radix Sort

Running time: $O\left(\frac{b}{r}\left(n+2^{r}\right)\right)$

Space usage: $O(2^r)$

Example: Sorting n 32-bit words

	Time	Extra Space
Counting Sort	$O(n + 2^{32})$	2^{32} words
Radix Sort 4 digits, 8 bits each	O(4(n+256))	256 words

Faster Sorting Algorithms

Counting Sort:

- Linear time, lots of space

Radix Sort:

- Linear time, more efficient space

Integer Sorts:

- O(n loglog n) time
- Efficient space
- Complicated and mostly theoretical

Summary

QuickSort: $O(n \log n)$

- Partitioning an array
- Deterministic QuickSort
- Paranoid Quicksort

Order Statistics: $O(n)$

- Finding the k^{th} smallest element in an array.
- Key idea: partition
- Paranoid Select

Comparison Sorting

What is a comparison?

- if (a < b) then …
- if (a > b) then …
- if (a == b) then …

Comparison Sorting

In Java:

- Interface java.lang.Comparable

  public int **compareTo**(Object o)

```
class Counter implements Comparable<Counter> {
        int iCount;
        public int compareTo(Counter thatC) {
                if (iCount == thatC.iCount) return 0;
                else if (iCount < thatC.iCount) return -1;
                else if (iCount > thatC.iCount) return 1;
        }
}
```

Comparison Sorting

We say that a sorting algorithm is a

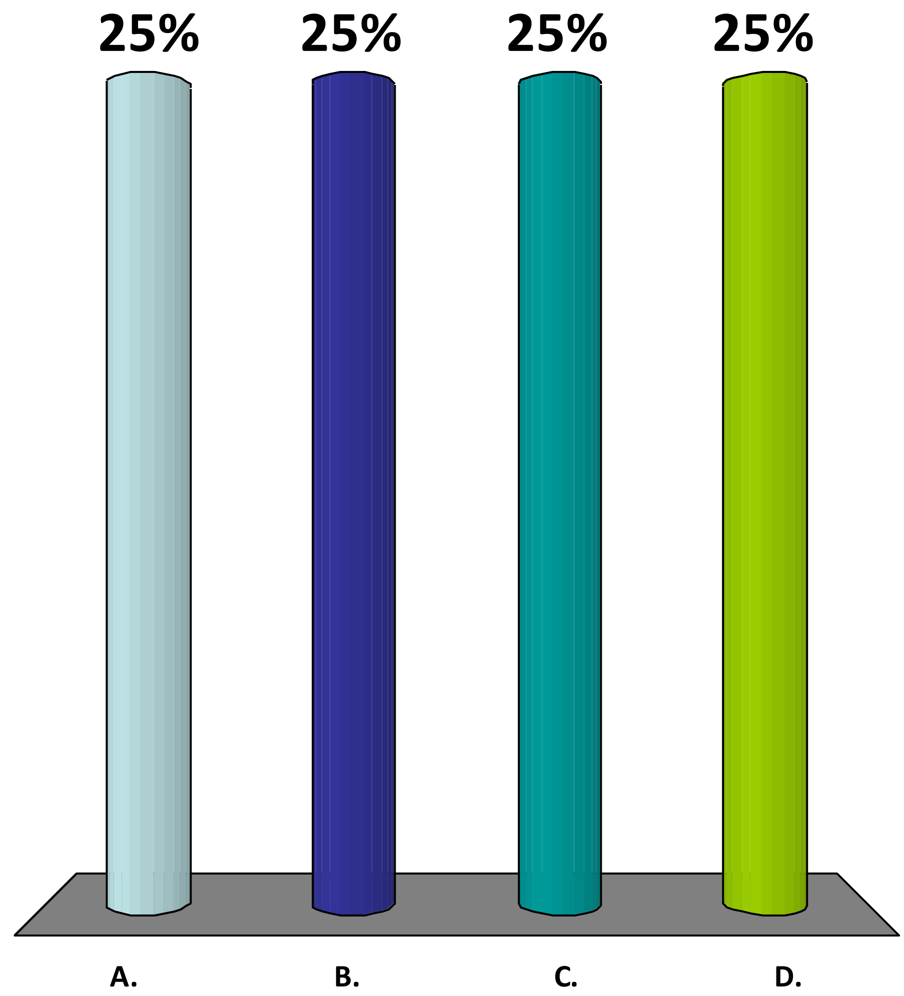
comparison sort

if only comparisons are used to determine the order of the elements.

Examples: MergeSort, Heapsort,
DQuickSort, InsertionSort,
etc.

Our best speed for the worst case in all **<u>Comparison</u>** sorts is O(n log n) because…

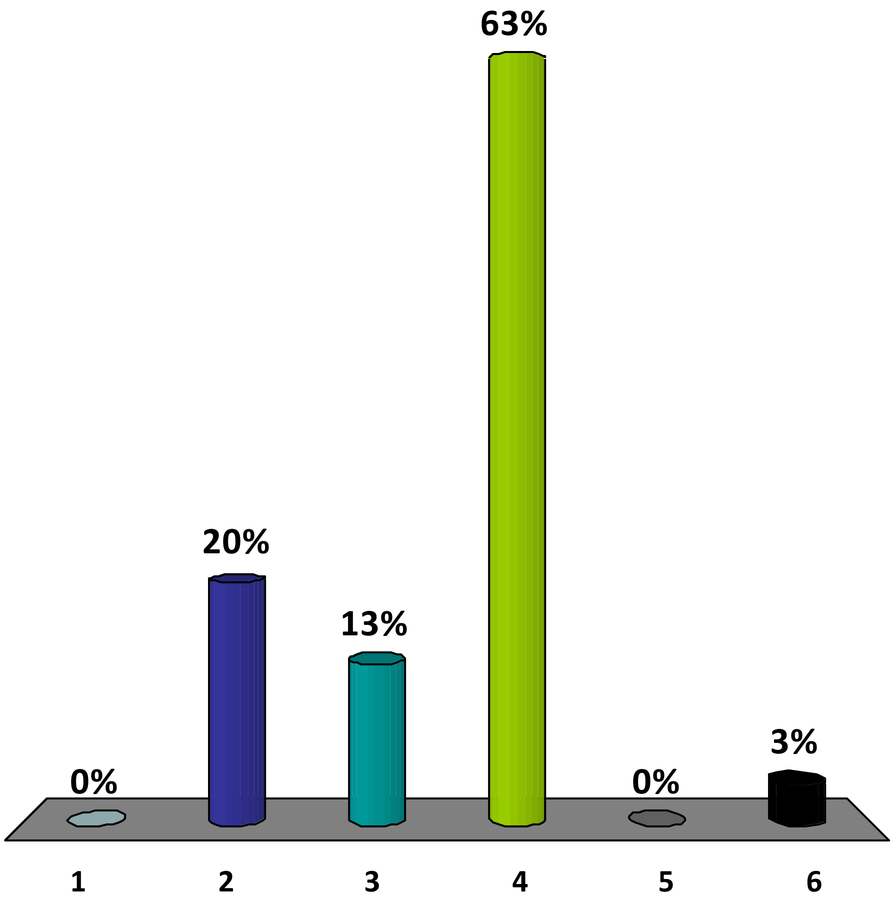
A. That is the best running time
B. There exists an algorithm better than O(n log n)
C. I don't like oxymoron. I *<u>clearly misunderstood</u>* it.
D. I am not sure and no one on Earth is sure about that



How many comparisons to sort?

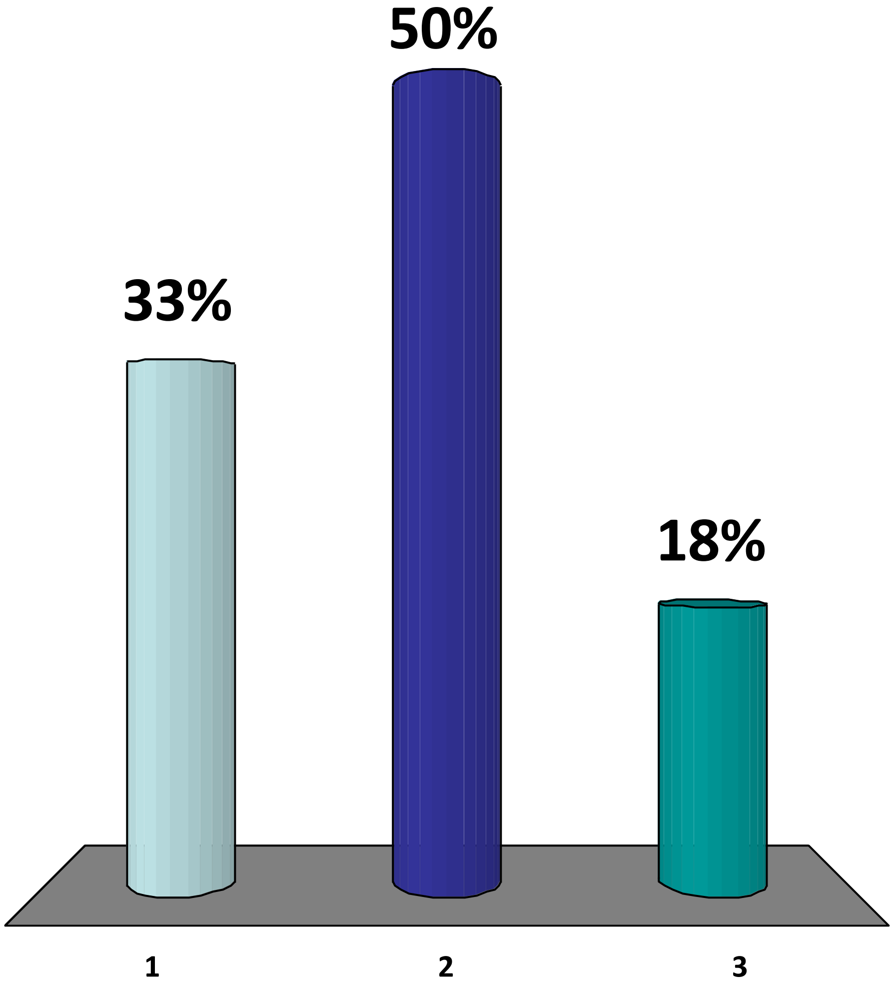
If (a, b, c) are elements, which of the following is illegal in a comparison sort?

1. if (a < b) then…
2. c = a
3. if (b==c) then…
4. if (a == b/2) then…
5. if (a > b) then…
6. None of the above.



Can you sort 5 elements {a, b, c, d, e} using only 3 comparisons?

1. Yes
2. No
3. Maybe



How many comparisons to sort?

Can you sort 5 elements $\{a, b, c, d, e\}$ using only 3 comparisons?

- There must be one or two elements that are not compared to the others!
- Ex: compare (a,b), (b,c), (d,e)---c and d not compared
- Ex: compare (a,b), (b,c), (c,d)---d and e not compared

Lower bound: sorting requires $> (n - 2) = \Omega(\mathrm{n})$ comparisons.

How many comparisons to sort?

Can you sort 5 elements $\{a, b, c, d, e\}$ using only 4 comparisons? 5 comparisons? 6 comparisons?

How many comparisons to sort?

Can you sort 5 elements $\{a, b, c, d, e\}$ using only 4 comparisons? 5 comparisons? 6 comparisons?

Theorem: Sorting 5 elements requires 7 comparisons!

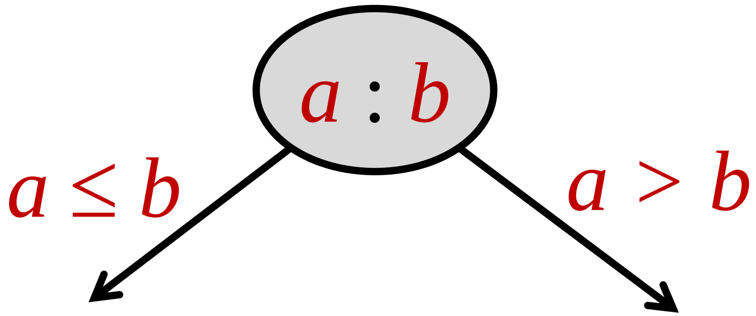
Algorithm as a Decision-Tree

Consider sorting: $\{a, b, c\}$

Algorithm as a Decision-Tree

Consider sorting: $\{a, b, c\}$

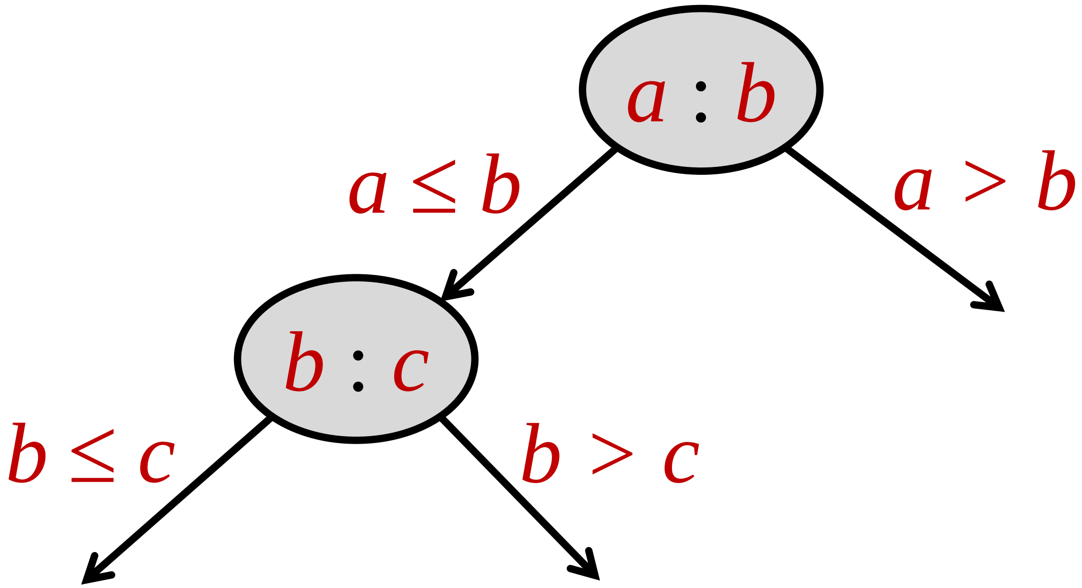
- Step 1: compare *a* and *b*



Algorithm as a Decision-Tree

Consider sorting: $\{a, b, c\}$

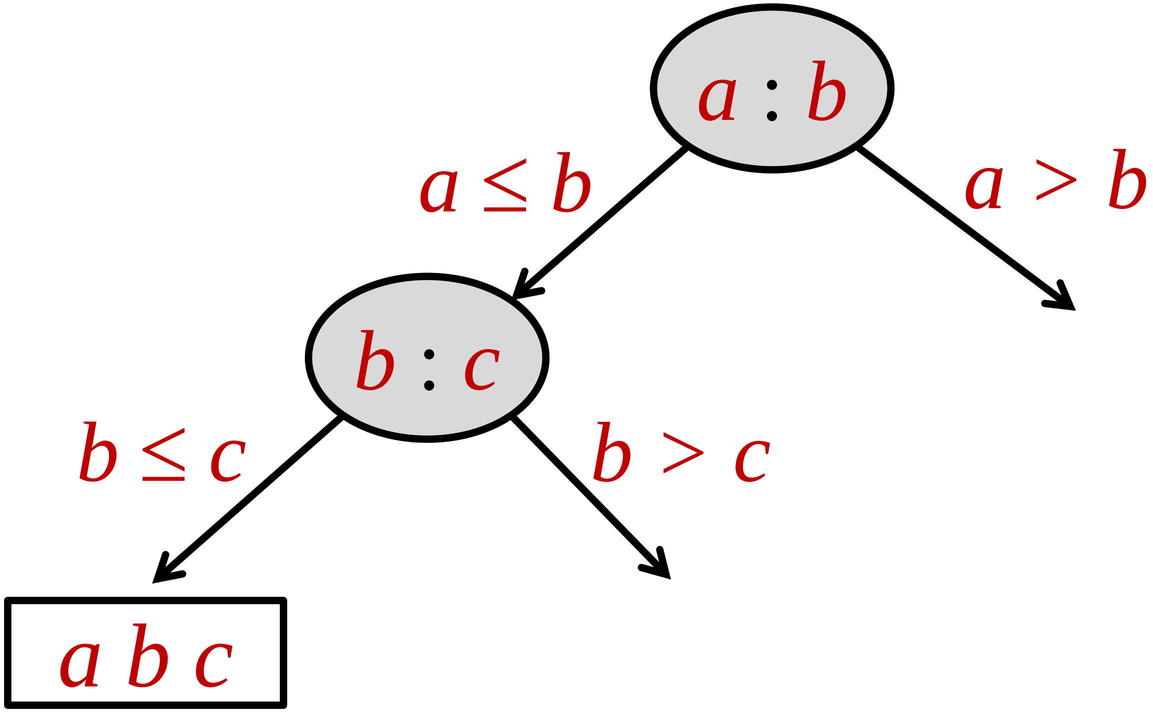
- Step 2: if ($a < b$) then compare b and c



Algorithm as a Decision-Tree

Consider sorting: $\{a, b, c\}$

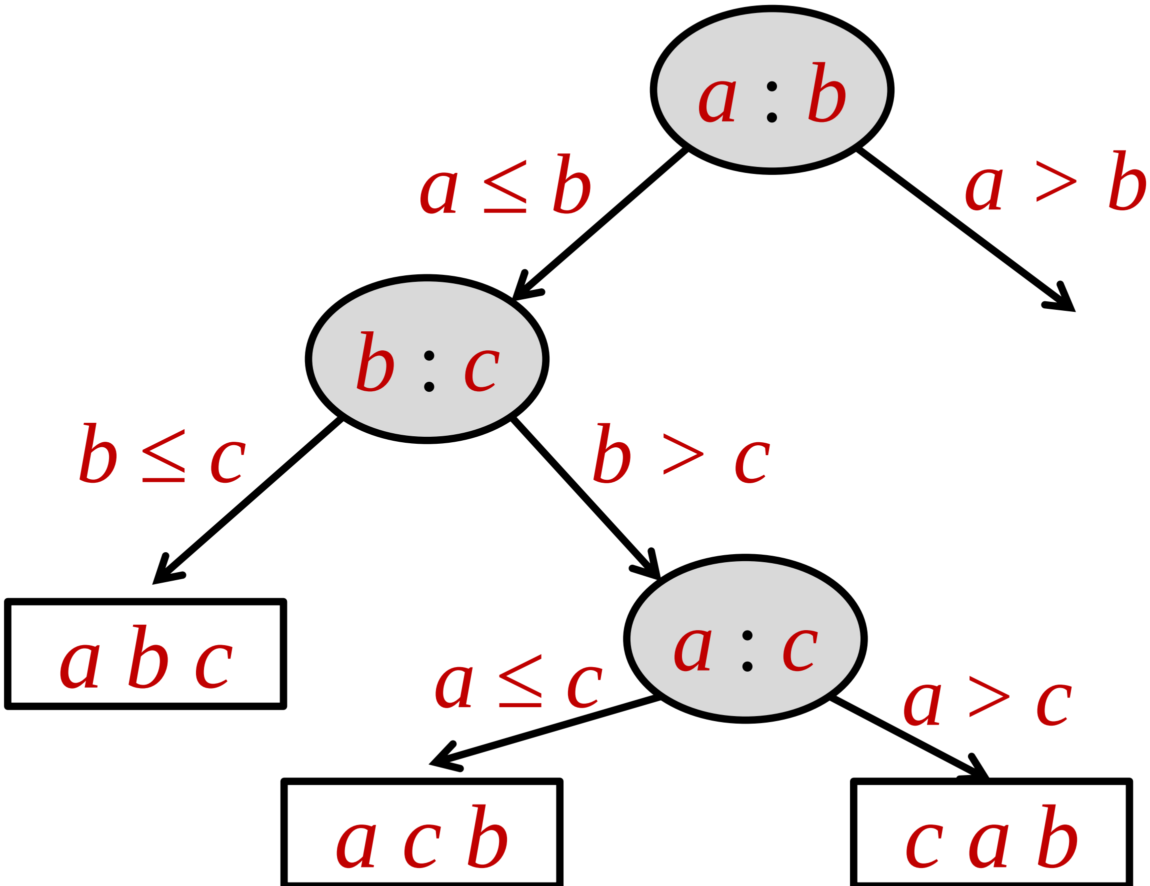
- Step 3: if ($a < b$) and ($b < c$) then output $<a, b, c>$



Algorithm as a Decision-Tree

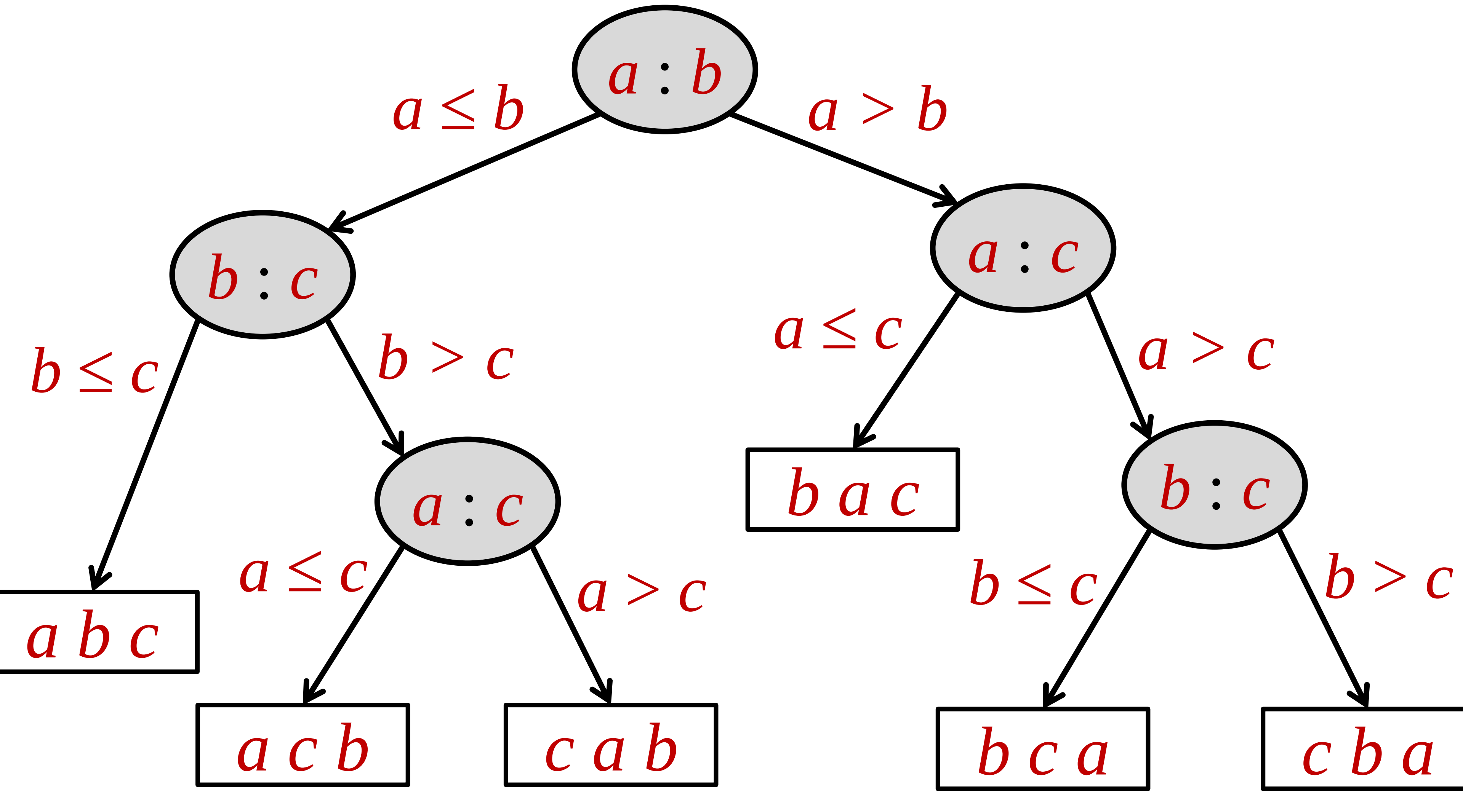
Consider sorting: $\{a, b, c\}$

– Step 4: if ($a < b$) and ($b > c$) then compare a and c



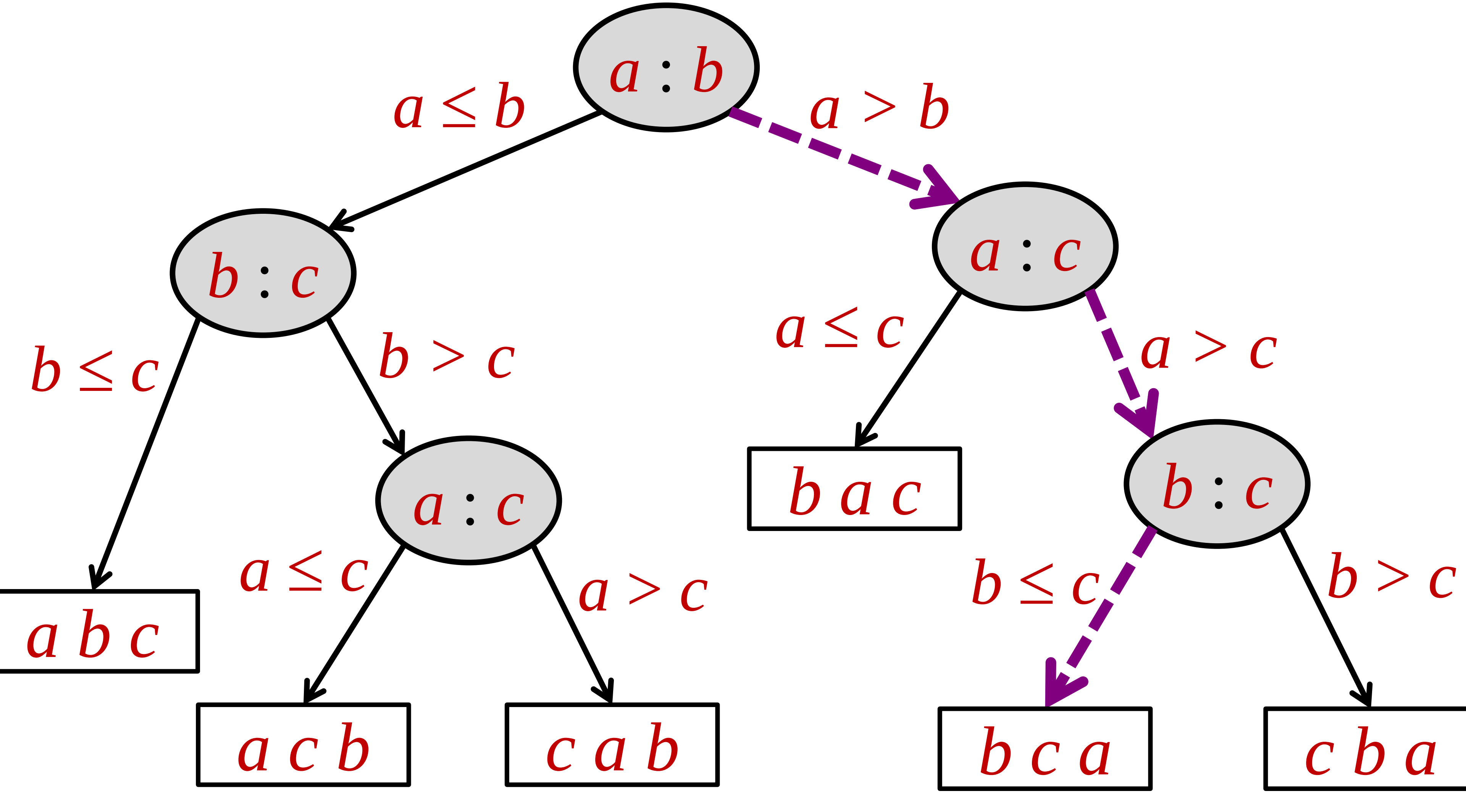
Algorithm as a Decision-Tree

Consider sorting: $\{a, b, c\}$



Algorithm as a Decision-Tree

Consider sorting: $\{a=9,\ b=2,\ c=6\}$



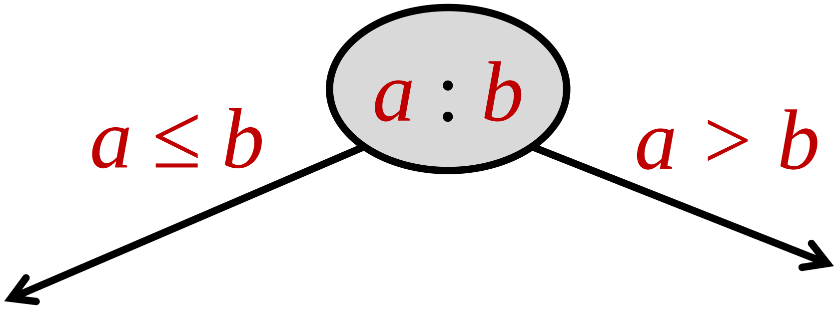
Algorithm as a Decision-Tree

A comparison-sort consists of:

A tree where:

Each node specifies two elements to compare.

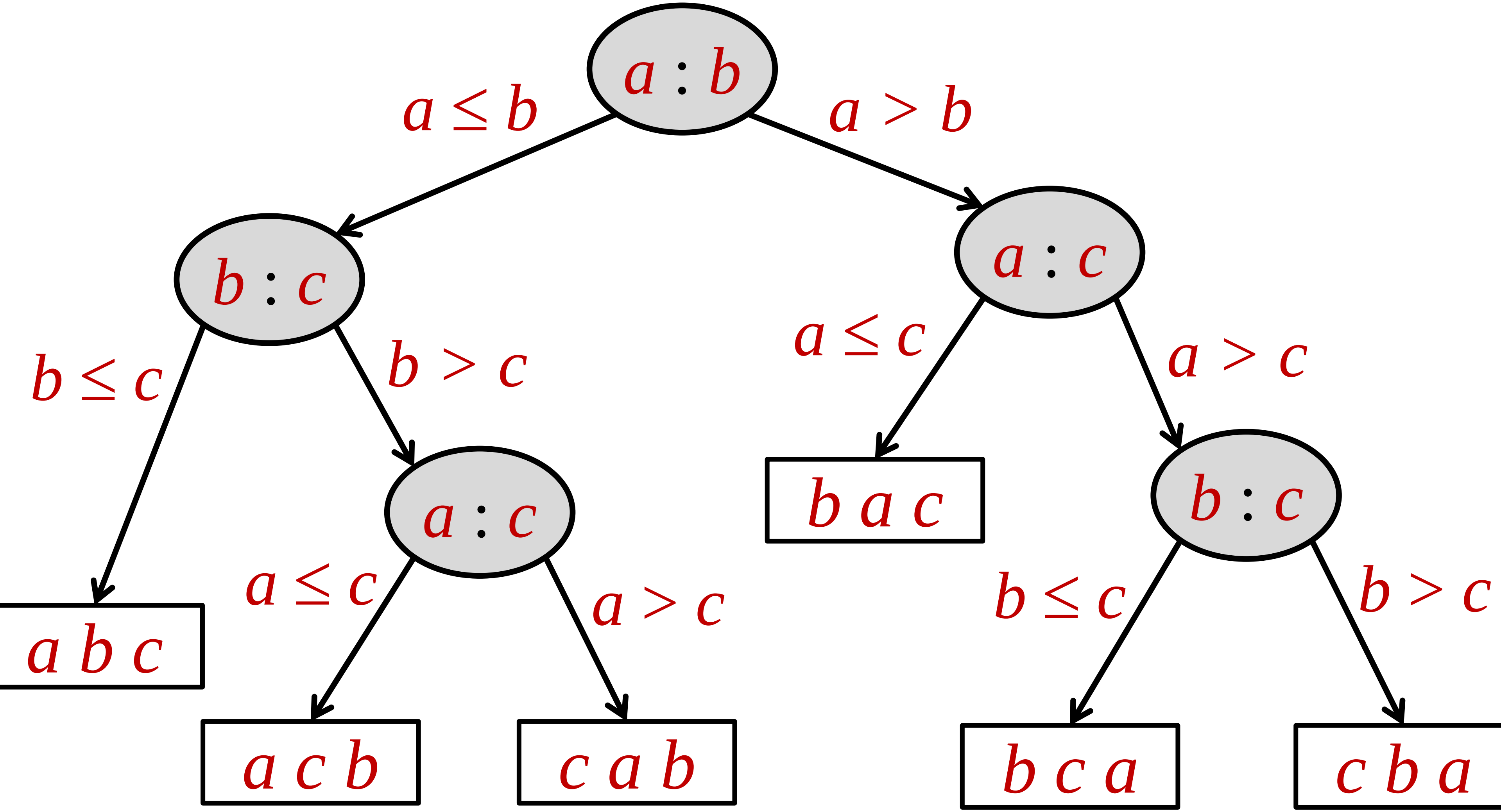
Each edge indicates which element is larger.



Every comparison-sort can be written this way.

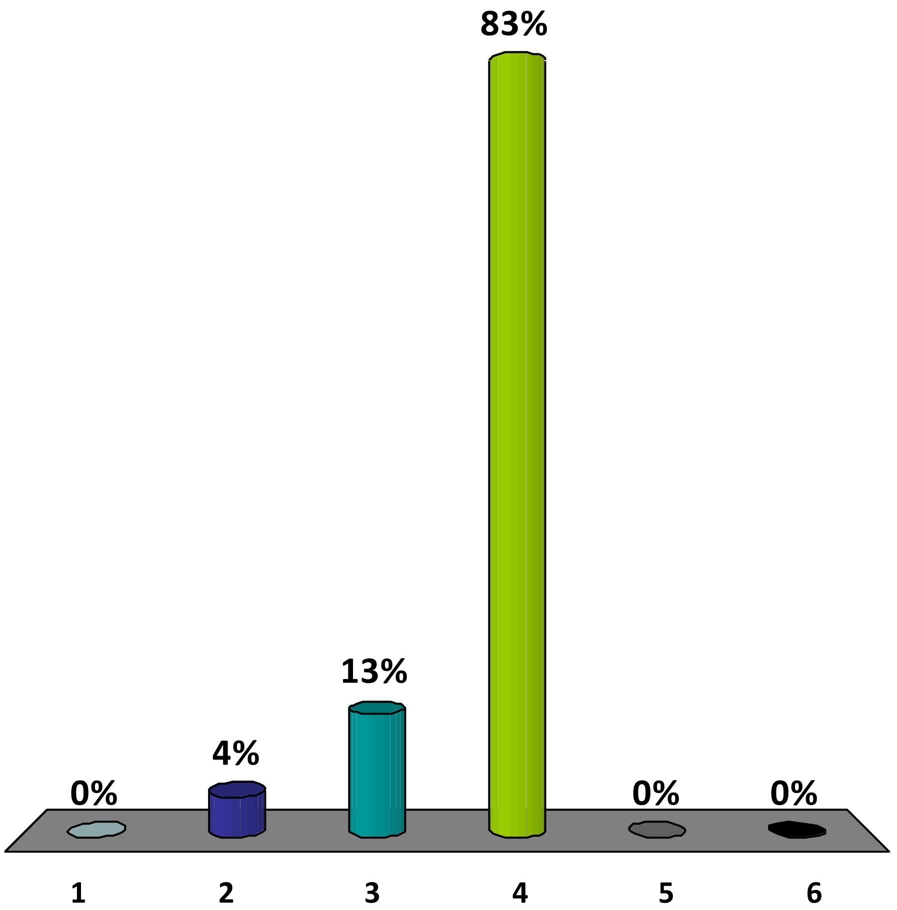
Algorithm as a Decision-Tree

Consider sorting: $\{a, b, c\}$



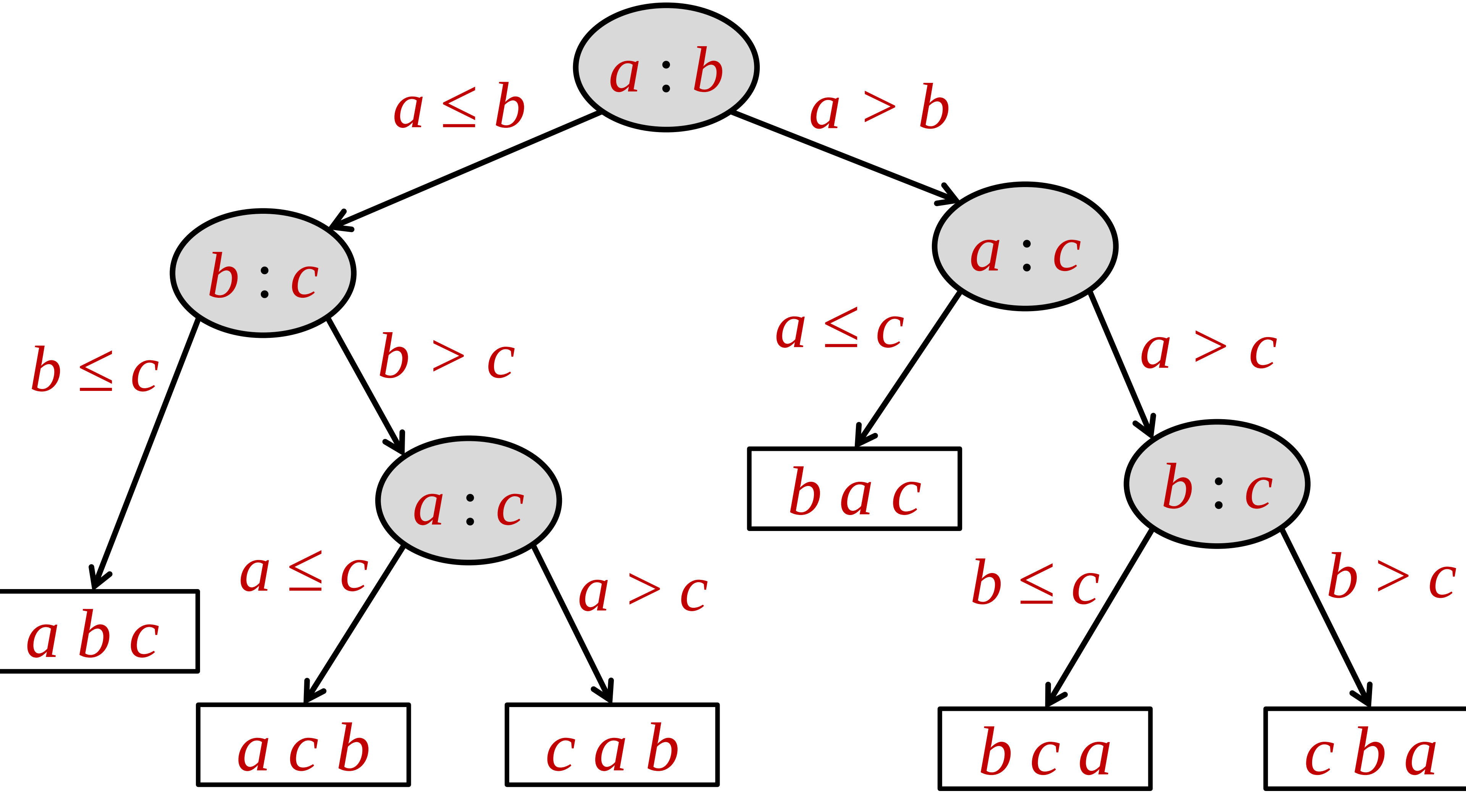
How many leaves are there in the *comparison-tree* for sorting {*a*, *b*, *c*, *d*, *e*}?

1. 5
2. 20
3. 60
4. 120
5. 256
6. 1024



Algorithm as a Decision-Tree

Consider sorting: $\{a, b, c\}$



Algorithm as a Decision-Tree

Sorting 5 elements: {a, b, c, d, e}

- Outputs: Every possible permutation!

  a b c d e

  a b c e d

  a b d c e

  a b d e c

  a b e c d

  a b e d c

  …

- Number of permutations: $n! = 5*4*3*2*1 = 120$

Algorithm as a Decision-Tree

Sorting n elements: $\{a_1, a_2, \ldots, a_n\}$

- Outputs: every possible permutation!
- Every sorting tree has $n!$ leaves.

Algorithm as a Decision-Tree

Sorting n elements: $\{a_1, a_2, \ldots, a_n\}$

- Outputs: every possible permutation.
- Every sorting tree has $n!$ leaves.

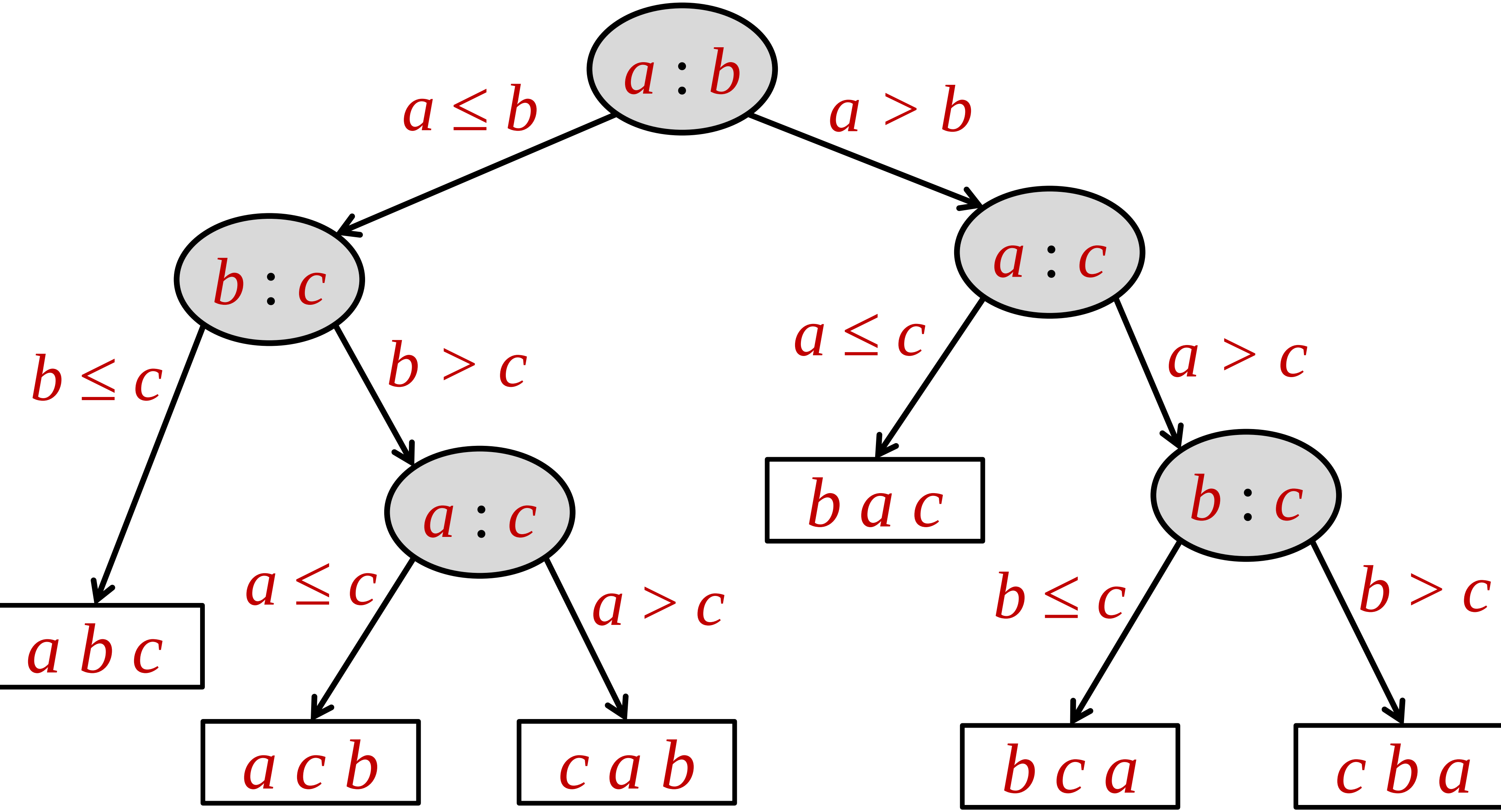
Running time of an algorithm:

- How many comparisons to get from root to leaf?
- Time = height of tree.

Key question: how high is a tree with $n!$ leaves?

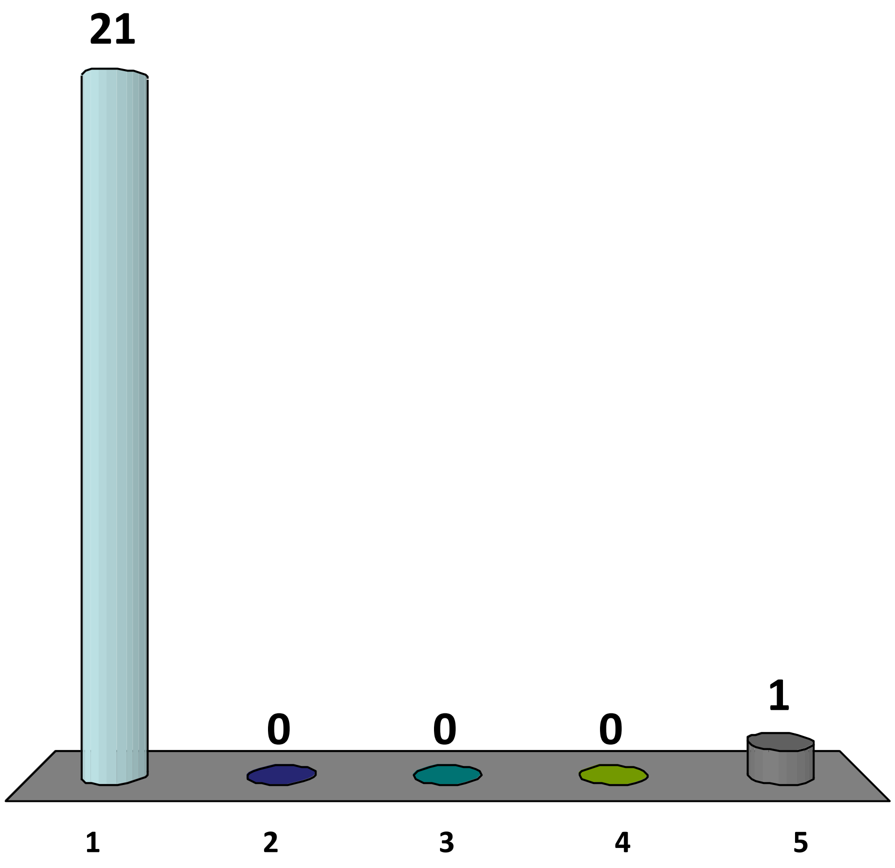
Algorithm as a Decision-Tree

Consider sorting: {*a*, *b*, *c*}



If a tree has *n* leaves, what is the minimum height?

1. $\log(n)$
2. $2\log(n)$
3. $\log^2(n)$
4. n
5. $n \log n$



Algorithm as a Decision-Tree

A tree with height h has $\leq 2^h$ leaves

A tree with n leaves has:

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

Height	Number of Leaves
0	1
1	≤ 2
2	≤ 4
3	≤ 8
…	…
h	$\leq 2^h$

Algorithm as a Decision-Tree

Claim: Every sorting tree has height $\geq \log(n!)$.

Proof:

1. Every sorting tree has $n!$ leaves, one for every possible output permutation.
2. A sorting tree with k leaves has height $\geq \log(k)$.

Algorithm as a Decision-Tree

Claim: Every sorting tree has height $\geq \log(n!)$.

Proof:

1. Every sorting tree has $n!$ leaves, one for every possible output permutation.
2. A sorting tree with k leaves has height $\geq \log(k)$.

Conclusion: Every comparison sort has running time $\geq \log(n!)$.

Algorithm as a Decision-Tree

Stirling's Approximation:

$$n! \approx \sqrt{2\pi \cdot n}\left(\frac{n}{e}\right)^n > \left(\frac{n}{e}\right)^n$$

$$\begin{aligned} \log(n!) &> \log[(n/\mathrm{e})^n] \\ &\geq n \log(n/\mathrm{e}) \\ &= \Omega(n \log n) \end{aligned}$$

How many comparisons to sort?

Theorem: Sorting 5 elements requires 7 comparisons!

Proof:

- If algorithm *A* is a comparison sort, the sorting tree for *A* has 5! = 120 leaves.
- A tree of height 6 has at most 2^6=64 leaves.
- Thus the sorting tree must be of height at least 7.
- Thus algorithm *A* has running time at least 7.

How many comparisons to sort?

Theorem: If A is a comparison sort, then sorting n elements requires time $\Omega(n \log n)$.

Proof:

- If algorithm A is a comparison sort, the sorting tree for A has $n!$ leaves.
- Thus the sorting tree must be of height at least $\log(n!)$.
- By Stirling's approximation, $\log(n!) > \Omega(n \log n)$.
- Thus the running time of A is $\Omega(n \log n)$.

How many comparisons to sort?

Theorem: If A is a comparison sort, then sorting n elements requires time $\Omega(n \log n)$.

Corollary: MergeSort is an asymptotically optimal comparison sort.

Have we shown that **QuickSort** is asymptotically optimal?

1. Yes
2. No
3. Maybe, it depends on the choice of pivot.

How many comparisons to sort?

Theorem: If A is a comparison sort, then sorting n elements requires time $\Omega(n \log n)$.

What about randomized algorithms, i.e,. QuickSort?

- We have assumed the algorithm can be represented as a binary tree.
- How do we represent random choices?
- You **can** adapt the decision-tree argument for randomized algorithms.
- More advanced, not in this class.

Summary

Comparison Sorting Algorithms

- Examples: MergeSort, InsertionSort, etc.
- For objects that implement `Comparable` interface.
- Every comparison sort requires time $\Omega(n \log n)$.
- MergeSort is asymptotically optimal.

Summary

QuickSort: $O(n \log n)$

- Partitioning an array
- Deterministic QuickSort
- Paranoid Quicksort

Order Statistics: $O(n)$

- Finding the k^{th} smallest element in an array.
- Key idea: partition
- Paranoid Select

Other fun things about sorting

- See what Obama says about sorting
  - https://www.youtube.com/watch?v=k4RRi_ntQc8
- Bogosort
  - Or called stupid sort, slowsort, shotgun sort or monkey sort

Xkcd:1185

https://gkoberger.github.io/stacksort/