CS2020
Data Structures and Algorithms

Welcome!

Problem Set

Problem Set 3:

- Due yesterday.

Problem Set 4:

- Released today.
- Due next Wednesday.
- A bit easier…

Quiz 1

Details:

- Friday, in-class
- Quick review at the end of lecture today

Textbooks on reserve at the library:

- Introduction to Algorithms (CLRS)
- Java Foundations (Lewis, DePasquale, Chase)
- Competitive Programming (Halim)

Discussion Group

This week:

- Problems posted
- Problem set review
- Quiz review

Let you tutor know which topics to cover… ☺

Today’s Plan

Order Statistics

– QuickSort Review

– Paranoid Select

– Deterministic Median

Linear-time Sorting

– How fast can you sort?

– Counting Sort, Radix Sort

Quiz Review

Recall: QuickSort

QuickSort($A[1..n]$, n)

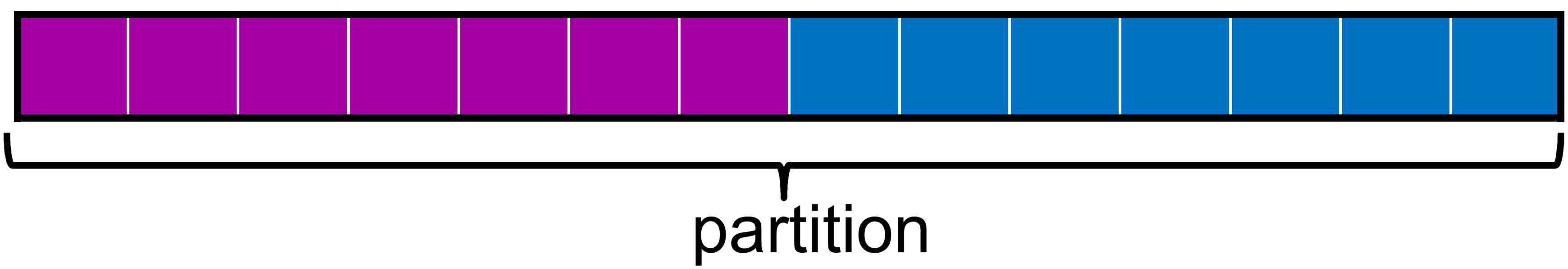
    if (n==1) **then** return

    else $pIndex$ = **random**(1, n)

        p = partition($A[1..n]$, n, $pIndex$)

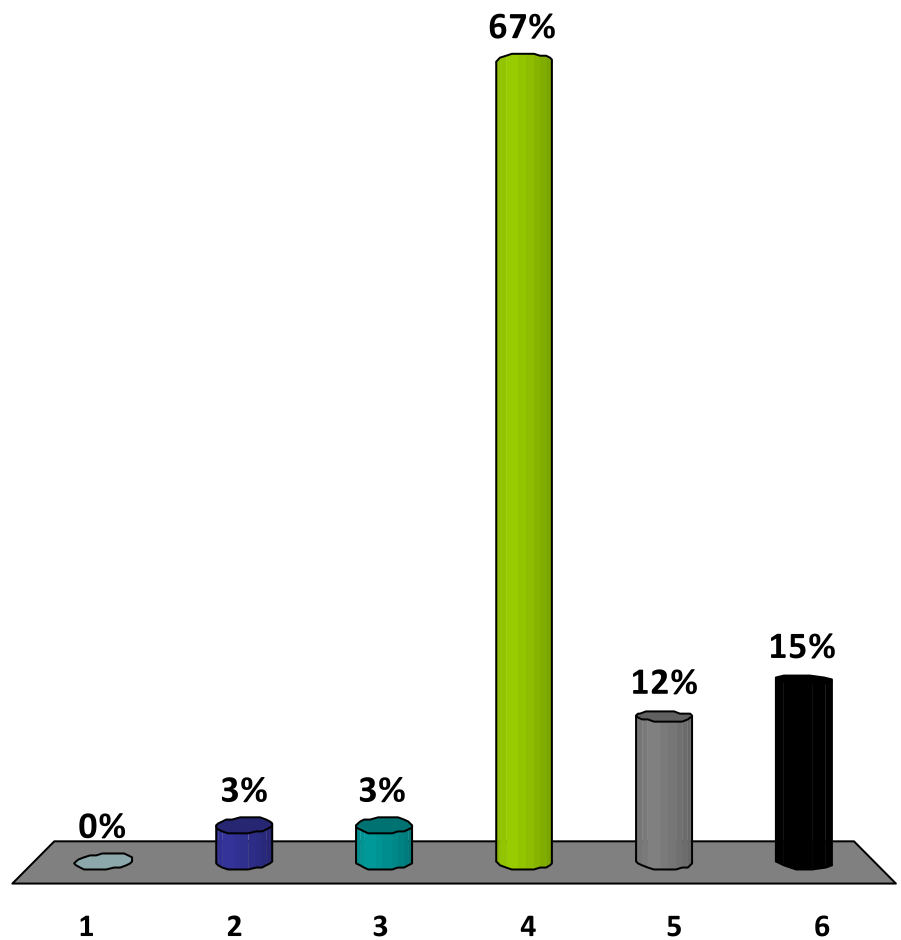
        x = QuickSort($A[1..p-1]$, $p-1$)

        y = QuickSort($A[p+1..\mathrm{n}]$, $n-p$)



If you choose the pivot as A[n/2], then QuickSort runs in time:

1. O(1)
2. O($\log^2 n$)
3. O(n)
4. O(n log n)
5. ✓ O(n^2)
6. I don't remember.



QuickSort (review)

Fact 1:

- If you choose the pivot badly, you get bad performance.
- Worst-cast running time: $O(n^2)$

QuickSort (review)

Fact 1:

- If you choose the pivot badly, you get bad performance.
- Worst-cast running time: $O(n^2)$

Fact 2:

- If you choose the pivot as the median, you get good performance.
- Recurrence: $T(n) = 2T(n/2) + O(n) = O(n \log n)$

QuickSort (review)

Idea: choose the pivot at random

Paranoid-QuickSort($A[1..n]$, n)

if (n==1) **then** return;

else **repeat**

$pIndex$ = **random**(1, n)

p = partition($A[1..n]$, n, $pIndex$)

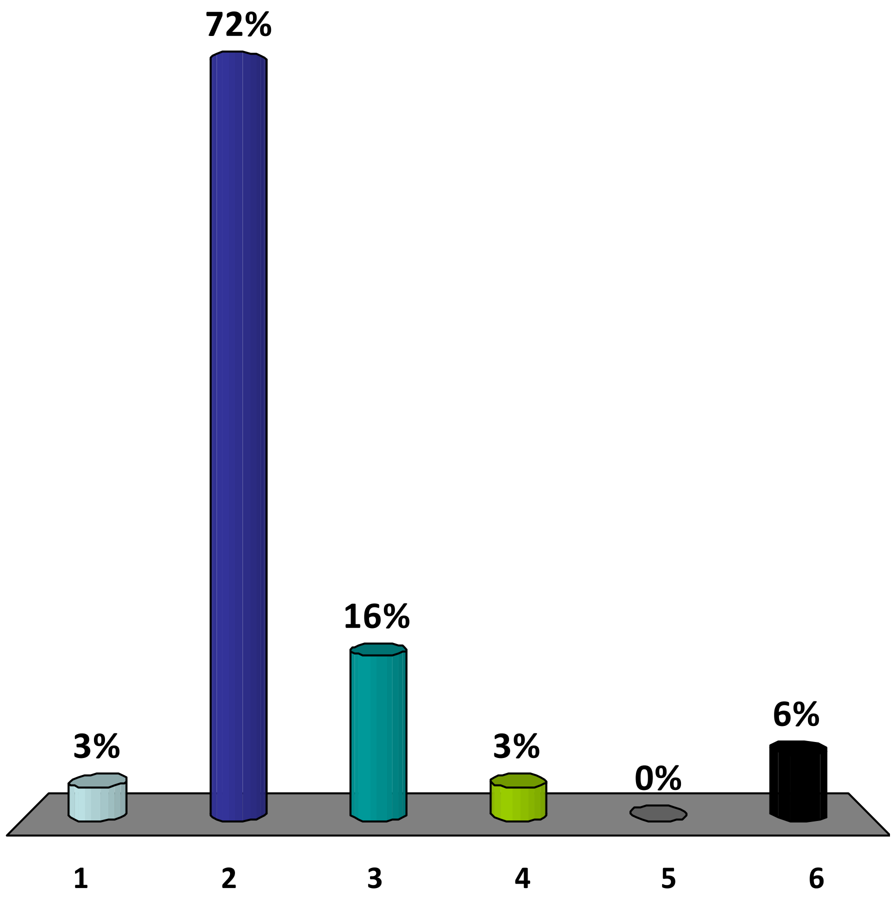
until $p > n/10$ **and** $p < n(9/10)$

x = QuickSort($A[1..p-1]$, $p-1$)

y = QuickSort($A[p+1..\text{n}]$, $n-p$)

How many times do you have to choose a pivot at random to get a good split?

1. E[# choices] = 1
2. E[# choices] < 2
3. E[# choices] < log n
4. E[# choices] < n
5. E[# choices] < n log n
6. I don't remember.



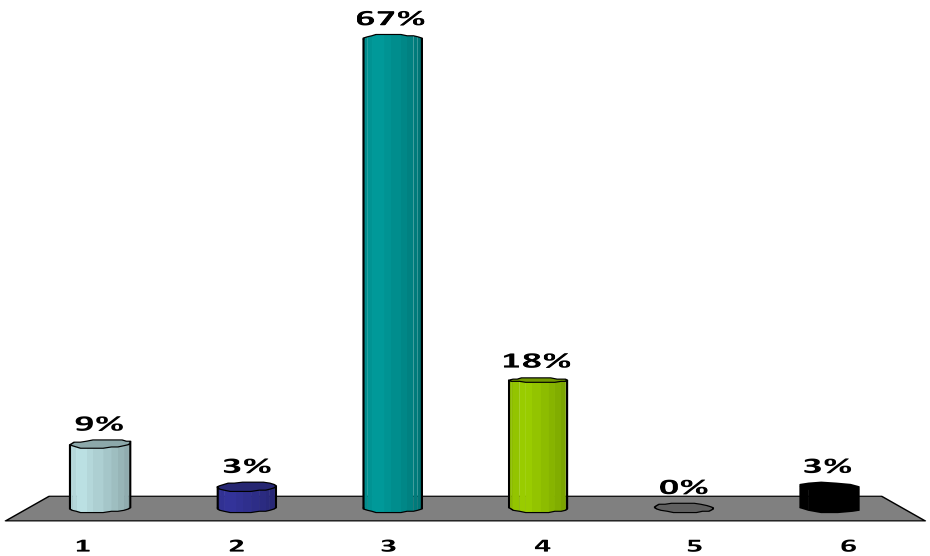
QuickSort (review)

Fact 3:

- If you choose a pivot at random, the expected number of random choices to find a *good* pivot is < 2.

The recurrence for Paranoid QuickSort is:
Expected[T(n)] = Expected[…]

1. $2T(n/2) + O(n)$
2. $2T(n/2) + O(n \log n)$
3. $T(n/10) + T(9n/10) + O(n)$
4. $T(n/10) + T(9n/10) + O(n \log n)$
5. $T(9n/10) + O(n)$
6. I don't remember.



QuickSort (review)

Fact 3:

– If you choose a pivot at random, the expected number of random choices to find a *good* pivot is < 2.

Fact 4:

– Paranoid QuickSort has the following recurrence:

$$\mathbf{E}[T(n)] = \mathbf{E}[\, T(n/10) + T(9n/10) + O(n) \,]$$
$$= O(n \log n)$$

Order Statistics

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

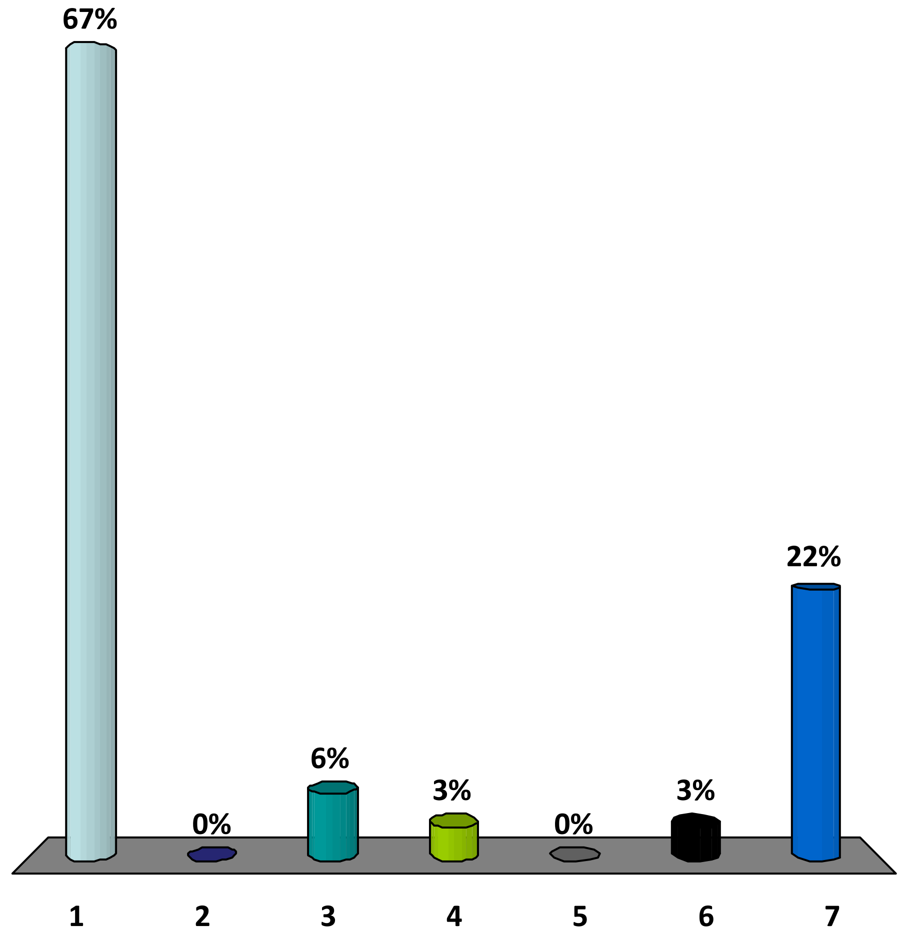
x_{10}	x_2	x_4	x_1	x_5	x_3	x_7	x_8	x_9	x_6

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

Find the 7th element ($k = 7$)

What is the 3rd smallest element in the list: [17, 5, 19, 23, 2, 101, 47]

1. 17
2. 5
3. 19
4. 23
5. 2
6. 101
7. I'm lazy.



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

Option 1:

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

Running time: O(n log n)

Order Statistics

Key Idea: partition the array

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

E.g.: Partition around x_5

Order Statistics

Key Idea: partition the array

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

E.g.: Partition around x_5

Continue searching in the correct half.

E.g.: 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

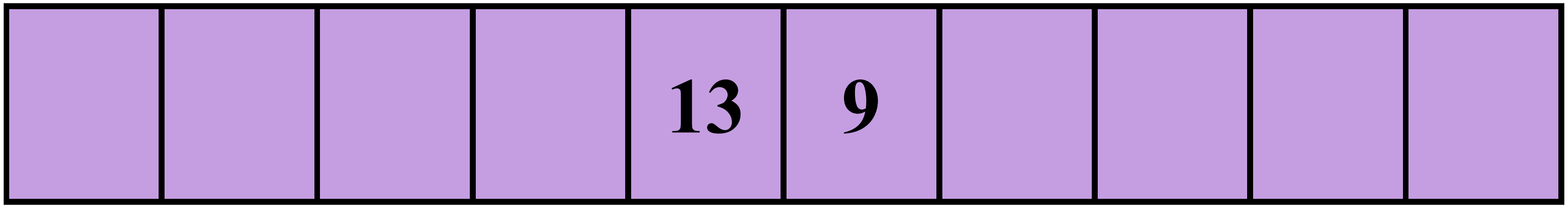
9	**8**	**13**	**5**	**3**	**6**				

Search for: 5 – 4 = 1 in right half

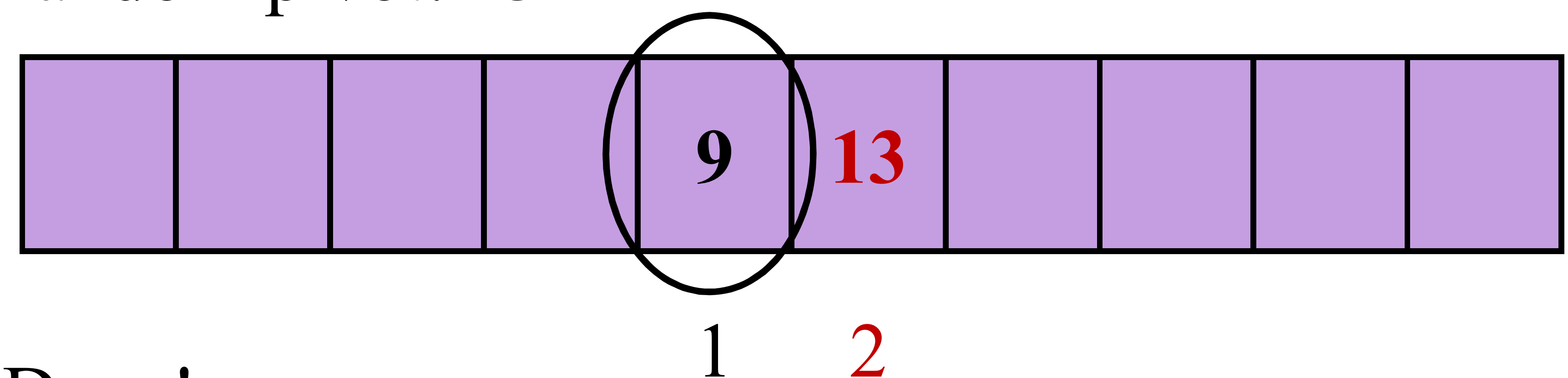
6	**3**	**5**	**8**	**13**	**9**				
1	2	3	4	5	6				

Order Statistics

Search for: 5 – 4 = 1 in right half



Random pivot: 13



Done!

Finding the k^{th} smallest element

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

if (n==1) **then return** $A[1]$;

else Choose random pivot index *pIndex*.

pIndex = partition($A[1..n]$, n, *pIndex*)

if (k == *pIndex*) **then return** A[*pIndex*];

else if (k < *pIndex*) **then**

return Select(A[1..*pIndex*–1], k)

else if ($k > p$) **then**

return Select(A[*pIndex*+1], k – *pIndex*)

Analysis

Paranoid-Select:

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

Recurrence:

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

Analysis

Paranoid-Select:

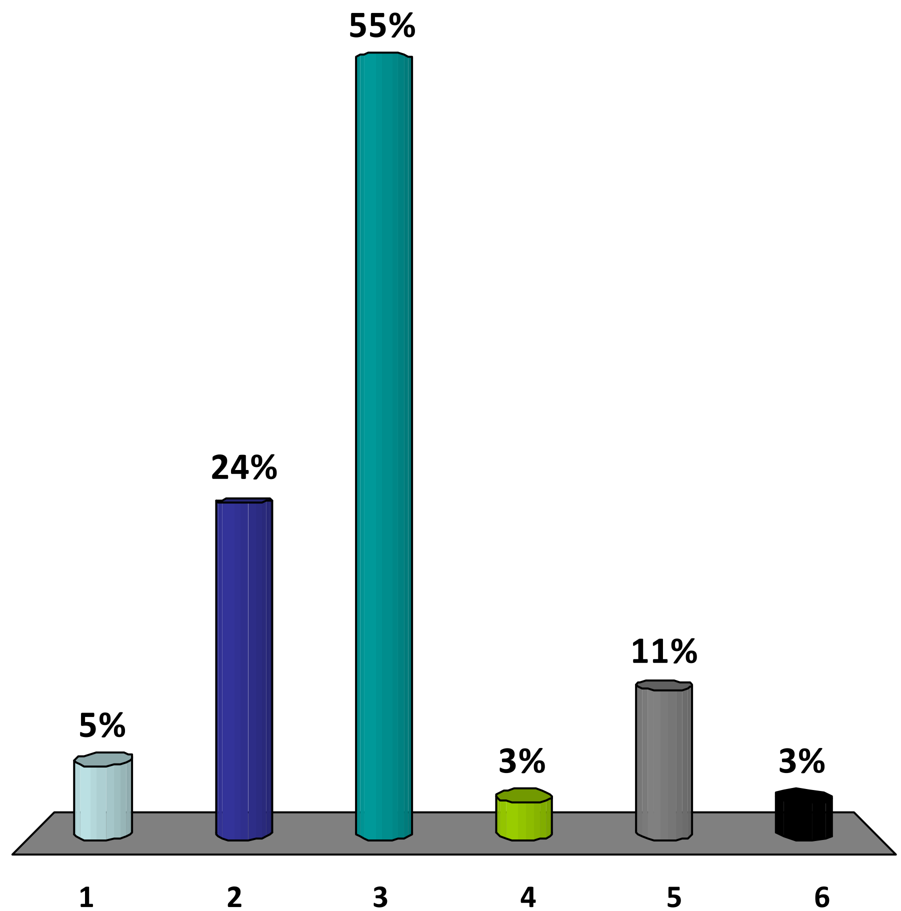
- Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$\mathbf{E}[T(n)] \le \mathbf{E}[T(9n/10)] + \mathbf{E}[\#\ \text{partitions}]*cn$$
$$\le \mathbf{E}[T(9n/10)] + 2cn$$

The expected running time of paranoid select is:

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:

$$\mathbf{E}[T(n)] \le \mathbf{E}[T(9n/10)] + \mathbf{E}[\#\text{ partitions}]*cn$$
$$\le \mathbf{E}[T(9n/10)] + 2cn$$
$$\le O(n)$$

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

Summary

Order Statistics

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

Deterministic Select

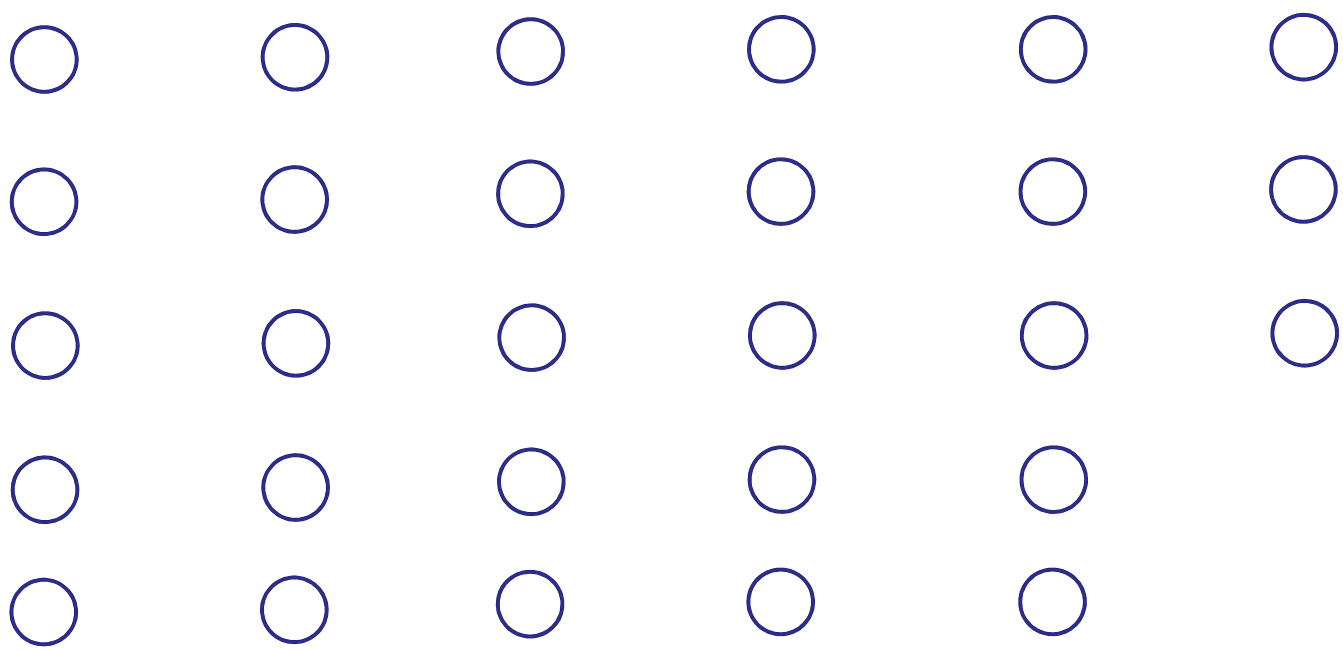
How to find the k^{th} element in $O(n)$ time:

- Deterministically!
- Tricky (a little)
- Useful (median is a good pivot)
- Theoretic interest only (random is faster)

Deterministic Select

Step 1:

- Divide the *n* elements into groups of 5.
- Find the median for each group of 5.

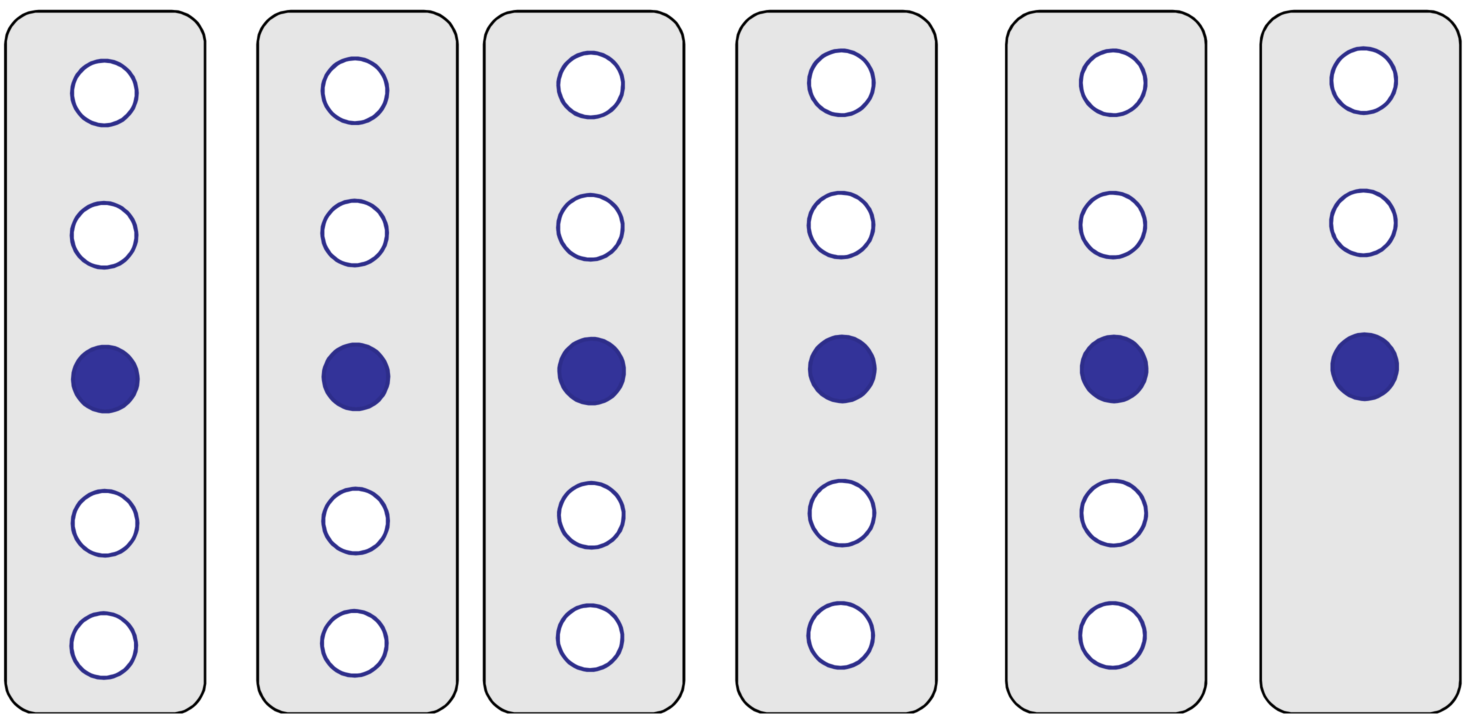


(Ex: n=28)

Deterministic Select

Step 1:

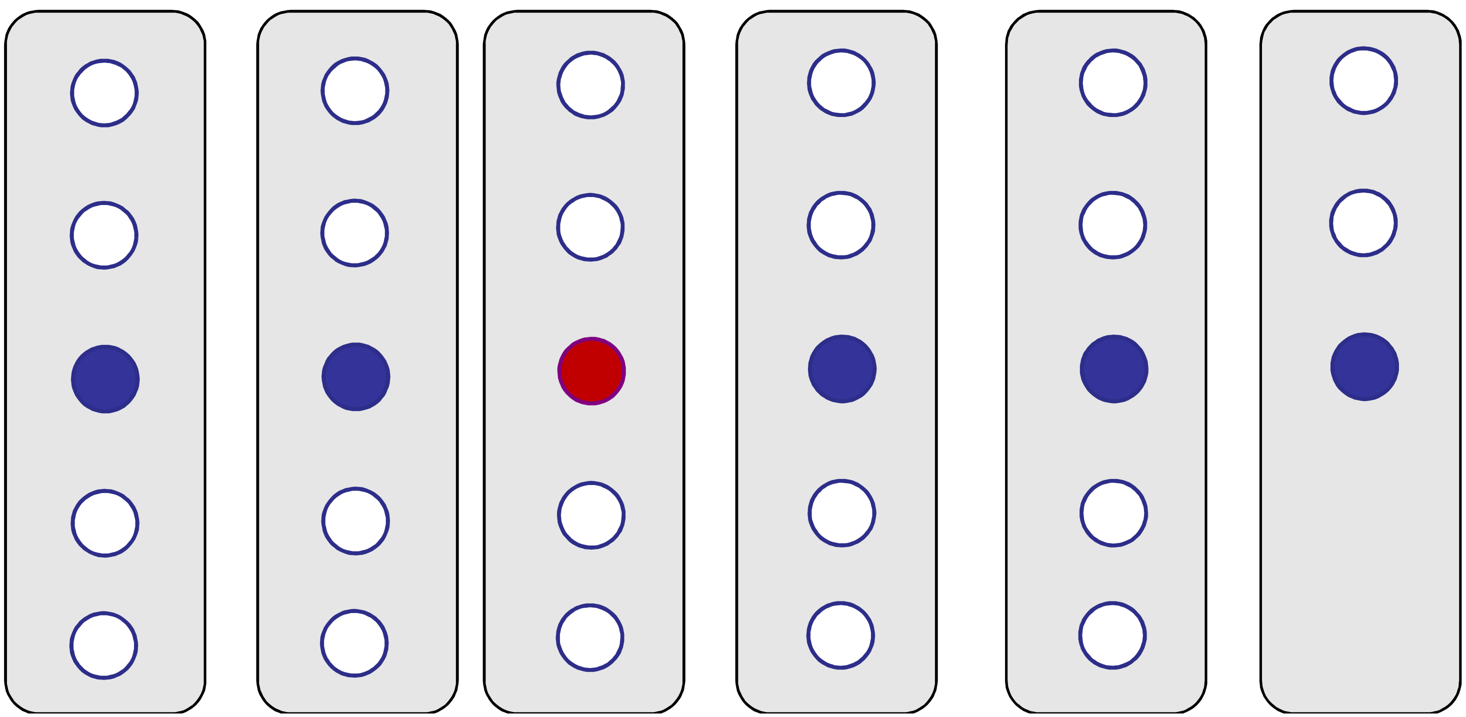
- Divide the n elements into groups of 5.
- Find the median for each group of 5.



Deterministic Median

Step 2:

- Recursively find the median of the $n/5$ middle elements.
- *Note*: this is not the real **median**!

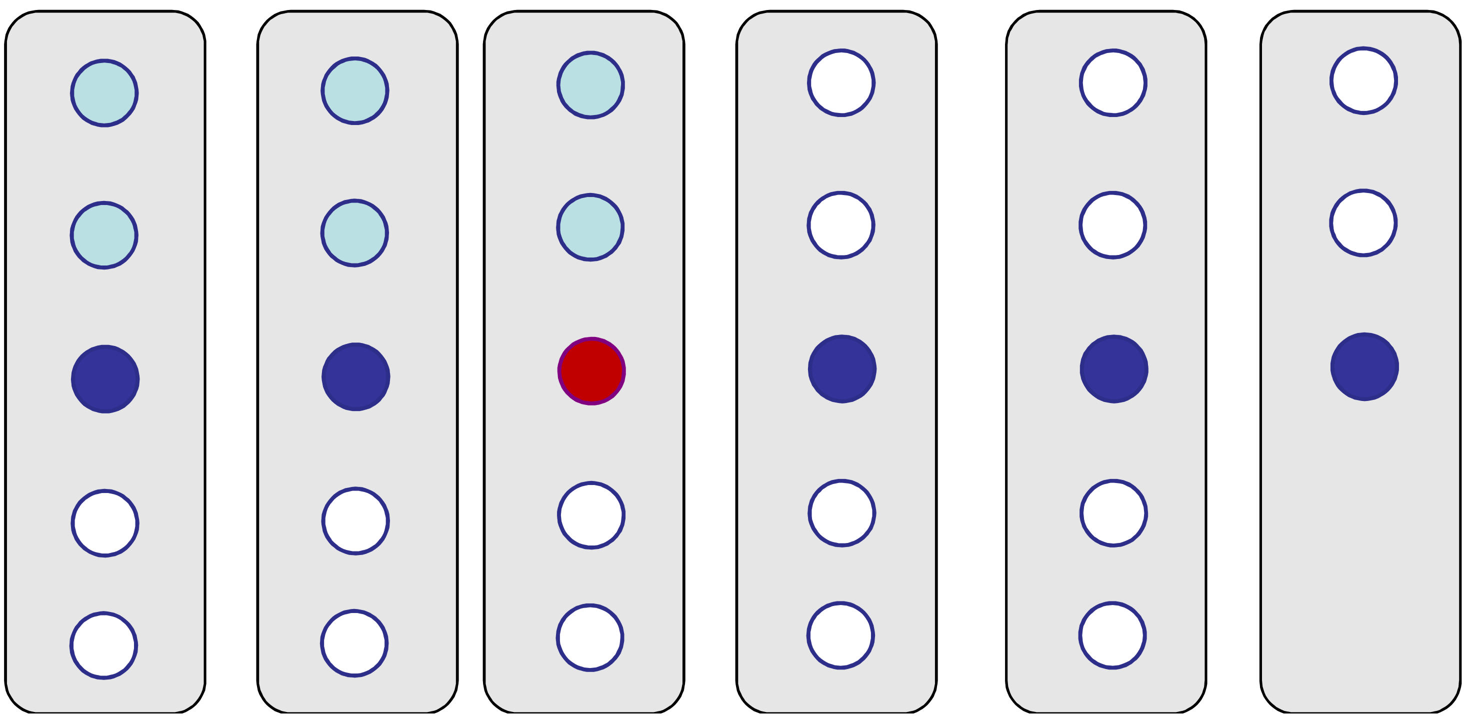


Deterministic Median

Step 2:

– Recursively find the median of the *n*/5 middle elements.

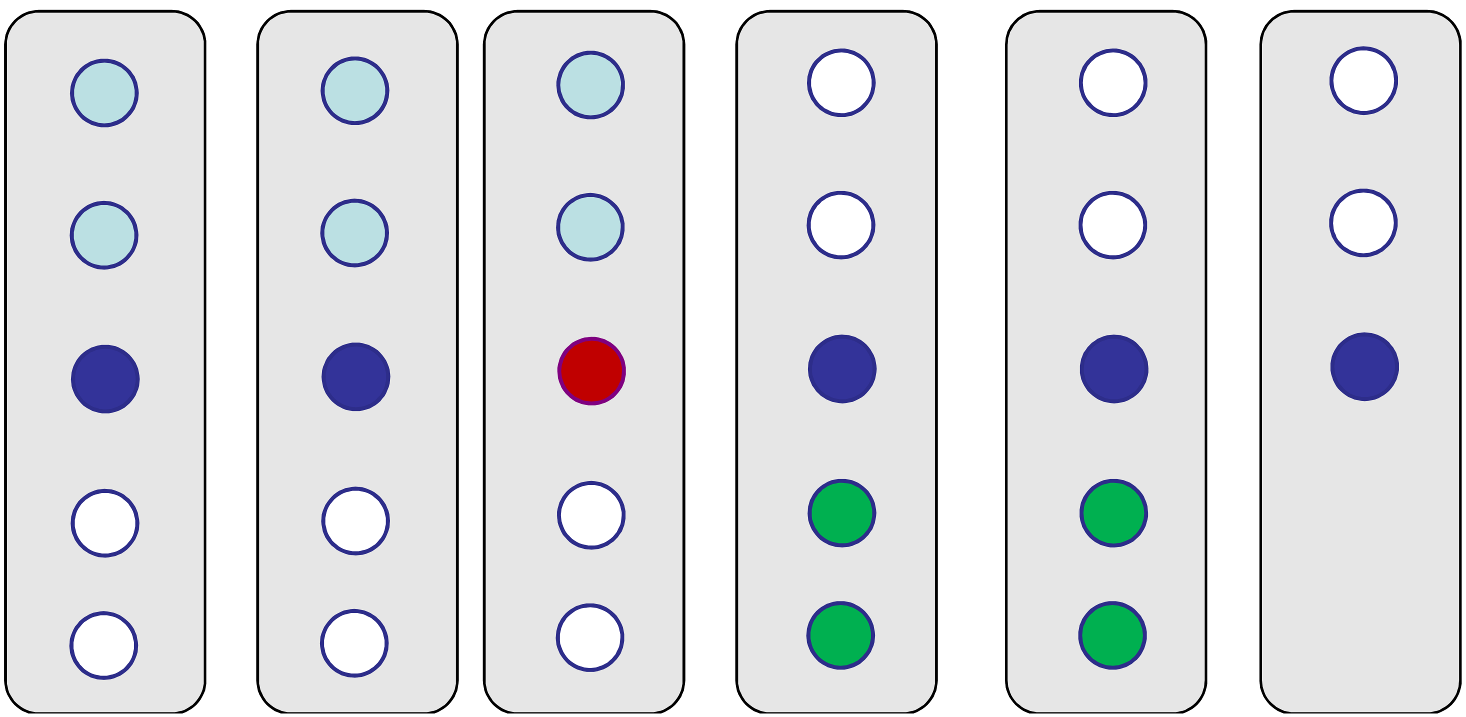
– At least (n/5)(1/2)(2) = n/5 elements are smaller.



Deterministic Median

Step 2:

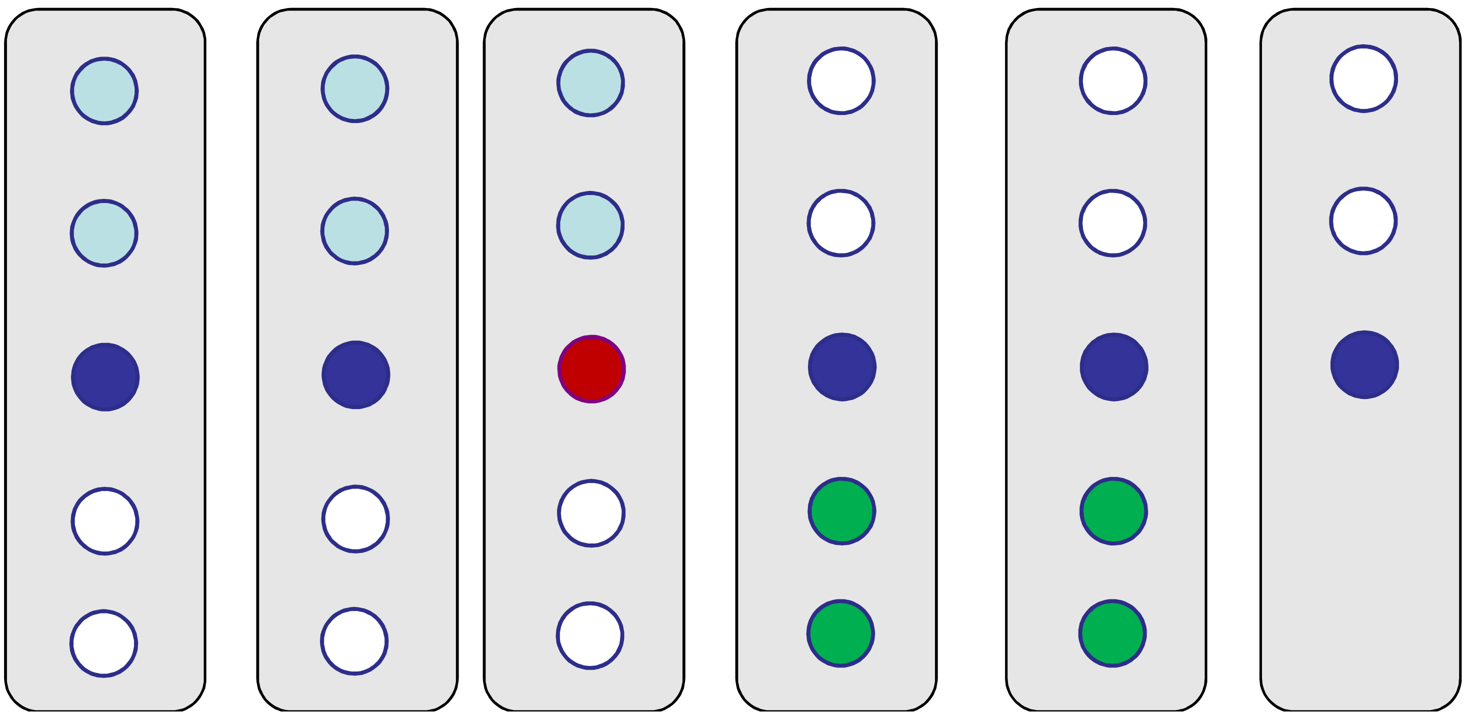
- Recursively find the median of the $n/5$ middle elements.
- At least $(n/5)(1/2)(2) = \lfloor n/5 \rfloor$ elements are smaller.
- At least $(n/5)(1/2)(2) = \lfloor n/5 \rfloor$ elements are larger.



Deterministic Median

Step 2:

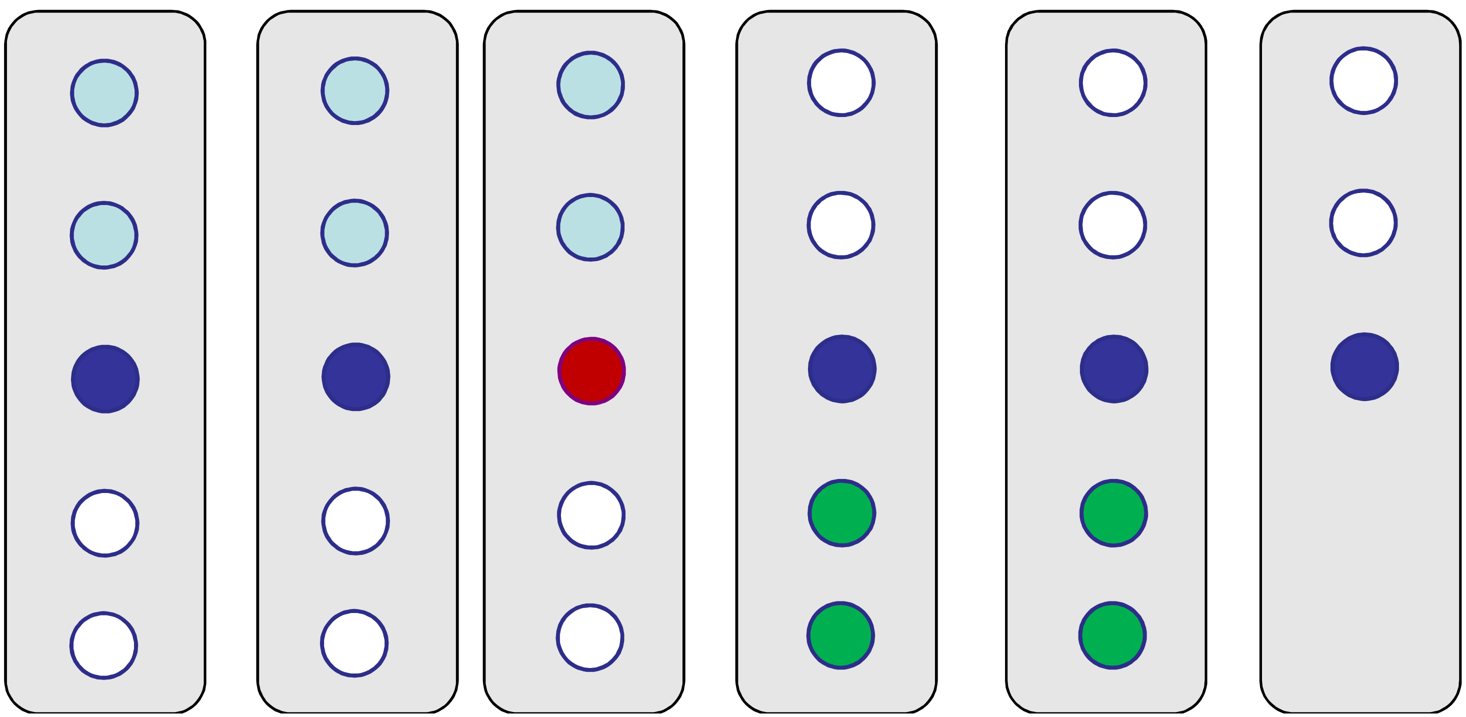
- Recursively find the median of the $n/5$ middle elements.
- At least $\lfloor 3n/10 \rfloor$ elements are smaller.
- At least $\lfloor 3n/10 \rfloor$ elements are larger.



Deterministic Median

Step 3:

- Partition around median of middle elements.
- Recurse.



Finding the kth smallest element

DSelect(A[1..n], n, k)

if (n≤5) **then return** kth element of A;

Divide A into groups of 5.

Let M be the array of medians for each group.

x = DSelect(M, n/5, n/10)

$pIndex$ = partition(A[1..n], n, x)

if (k == $pIndex$) **then return** A[$pIndex$];

else if (k < p) **then return** DSelect(A[1..p–1], k);

else if (k > p) **then return** DSelect(A[p+1,n], k–p);

Finding the kth smallest element

Recurrence:

$$T(n) = T(n/5) + T(7n/10) + O(n)$$
$$= O(n)$$

(*note*: $n/5 + 7n/10 < n$)

Exercise: *Why not divide into groups of 3?*

Deterministic QuickSort

DQuickSort(A[1..n], n)

if (n==1) **then** return;

else

$pIndex$ = DSelect(A, n, n/2)

p = partition(A[1..n], n, $pIndex$)

x = DQuickSort(A[1..p–1], p–1)

y = DQuickSort(A[p+1..n], n–p)

Summary

Fastest sorting algorithms: $O(n \log n)$

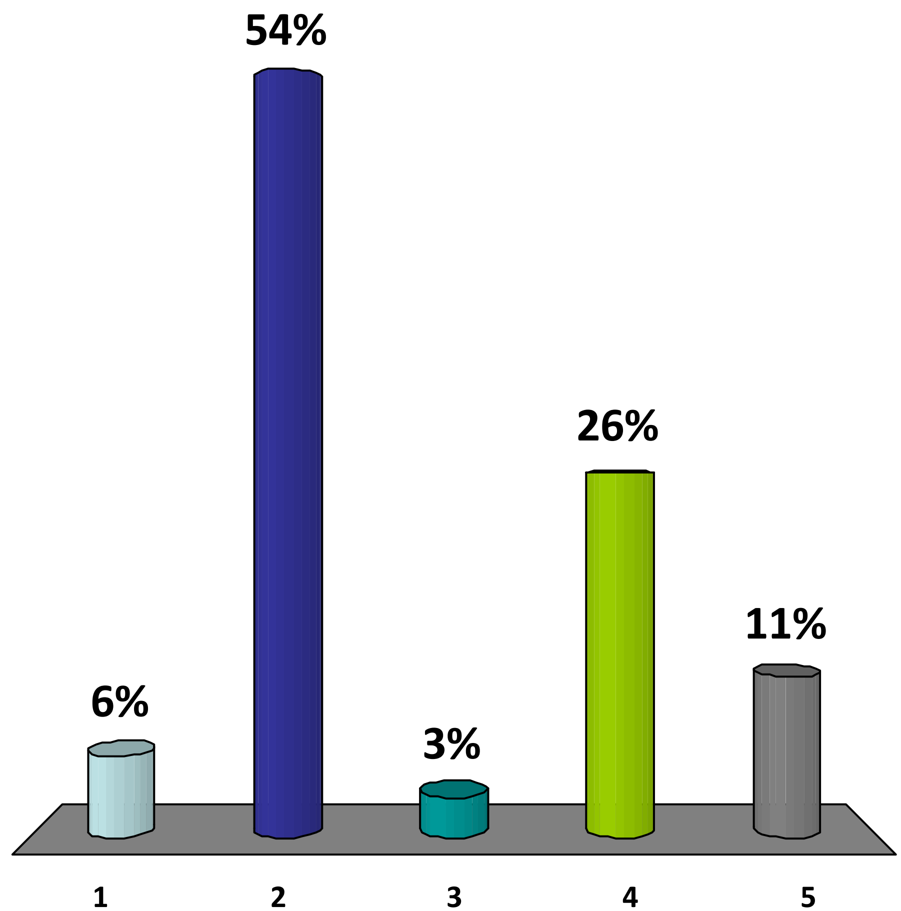
- MergeSort
- HeapSort
- Randomized QuickSort
- Deterministic QuickSort

Fastest selection algorithms: $O(n)$

- Randomized Select
- DSelect

How fast can we sort?

1. Faster than $O(n)$
2. $O(n)$
3. $O(n \log\log n)$
4. $O(n \log n)$
5. I'm lazy.



Comparison Sorting

What is a comparison?

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

Comparison Sorting

What is a comparison?

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

What types are *a* and *b*?

- Integer
- String
- Polynomials
- Anything that can be compared!

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

Sorting using `Comparable`:

```
void Sort(Counter[] A, int n)  // Bubblesort!
{
   for (int j=0; j<n; j++) {
          for (int i=0; i<n; i++)
          {
                 if (A[i].compareTo(A[i+1]) > 0) {
                        swap(A, i, i+1);
                 }
          }
   }
}
```

Comparison Sorting

Sorting using `Comparable`:

```
void Sort(Comparable[] A, int n)  // Bubblesort!
{
   for (int j=0; j<n; j++) {
          for (int i=0; i<n; i++)
          {
                 if (A[i].compareTo(A[i+1]) > 0) {
                        swap(A, i, i+1);
                 }
          }
   }
}
```

Comparison Sorting

Sorting using `Comparable`:

java.util.Arrays:

static void sort(Object[] a)

(Sorts an array of Comparable objects.)

static void binarySearch(Object[] a, Object key)

(Searches an array of Comparable objects.)

Comparison Sorting

Sorting using `Comparable`:

java.util.Collections:

static void sort(List[] a)

(Sorts an array of `Comparable` objects.)

static void binarySearch(L:ist[] a, Object key)

(Searches an array of `Comparable` objects.)

Comparison Sorting

We say that a sorting algorithm is a

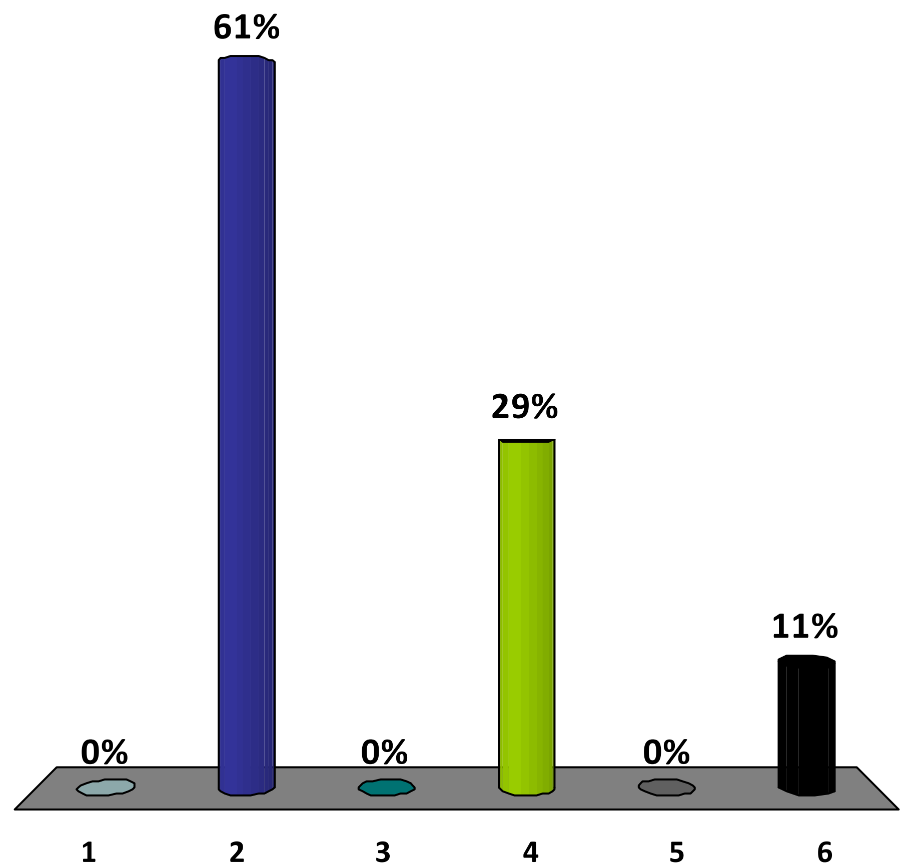
comparison sort

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

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

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

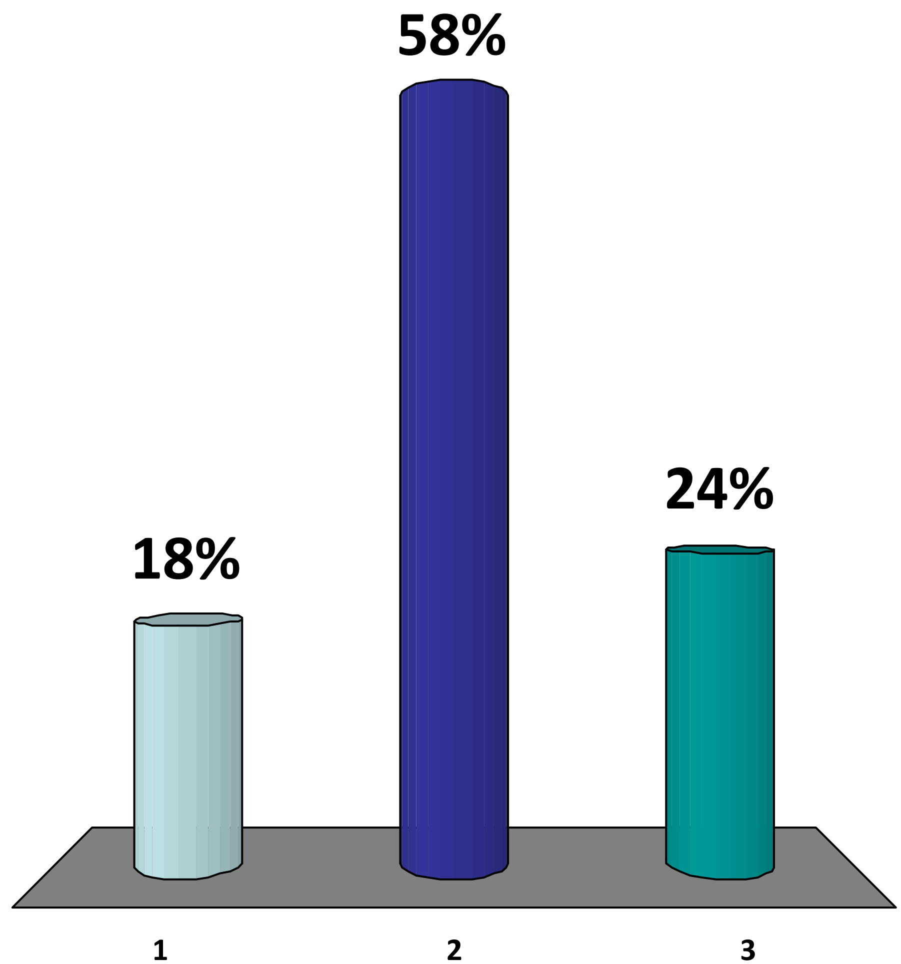
1. if (a < b) then…
2. c = a
3. if (b==c) then…
4. if (a == b/2) then…
5. if (a > b) then…
6. None of the above.



How many comparisons to sort?

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)---d and e not compared
- Ex: compare (a,b), (b,c), (c,d)---e not compared

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

Aside on asymptotic notation:

Recall:

- f(n) = O(g(n))

  There exist constants n_0 and c such that:

  for every $(n > n_0)$, $f(n) < cg(n)$

- f(n) = Ω(g(n))

  There exist constants n_0 and c such that:

  for every $(n > n_0)$, $f(n) > cg(n)$

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!

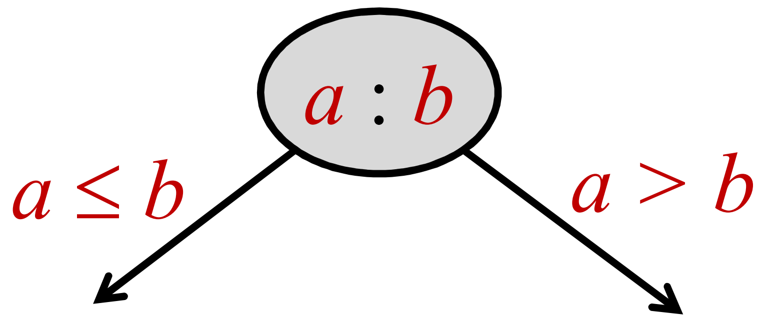
Algorithm as a Decision-Tree

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

Algorithm as a Decision-Tree

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

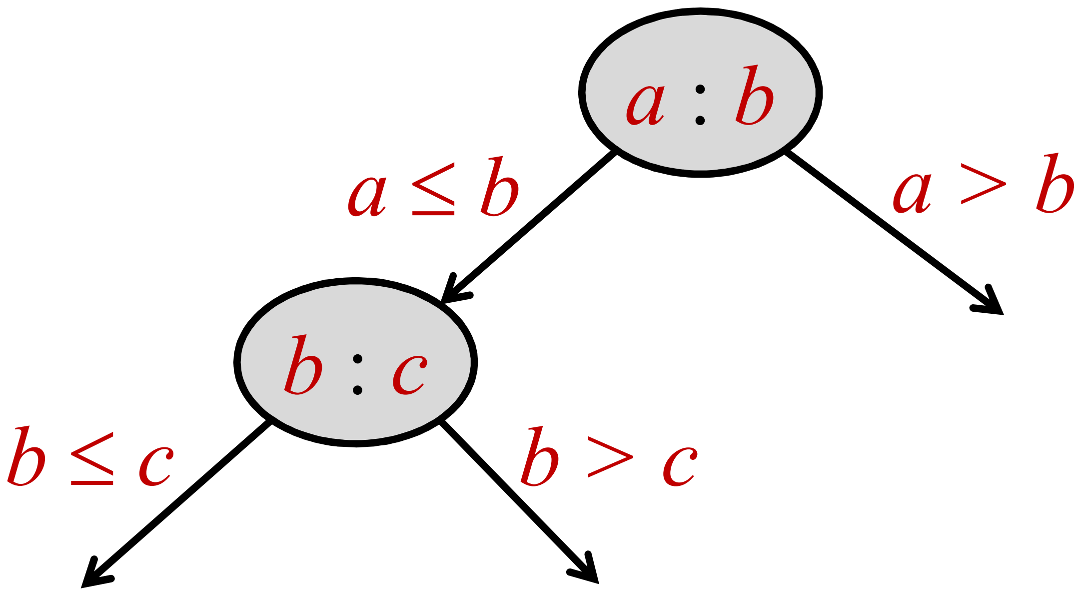
– Step 1: compare a and b



Algorithm as a Decision-Tree

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

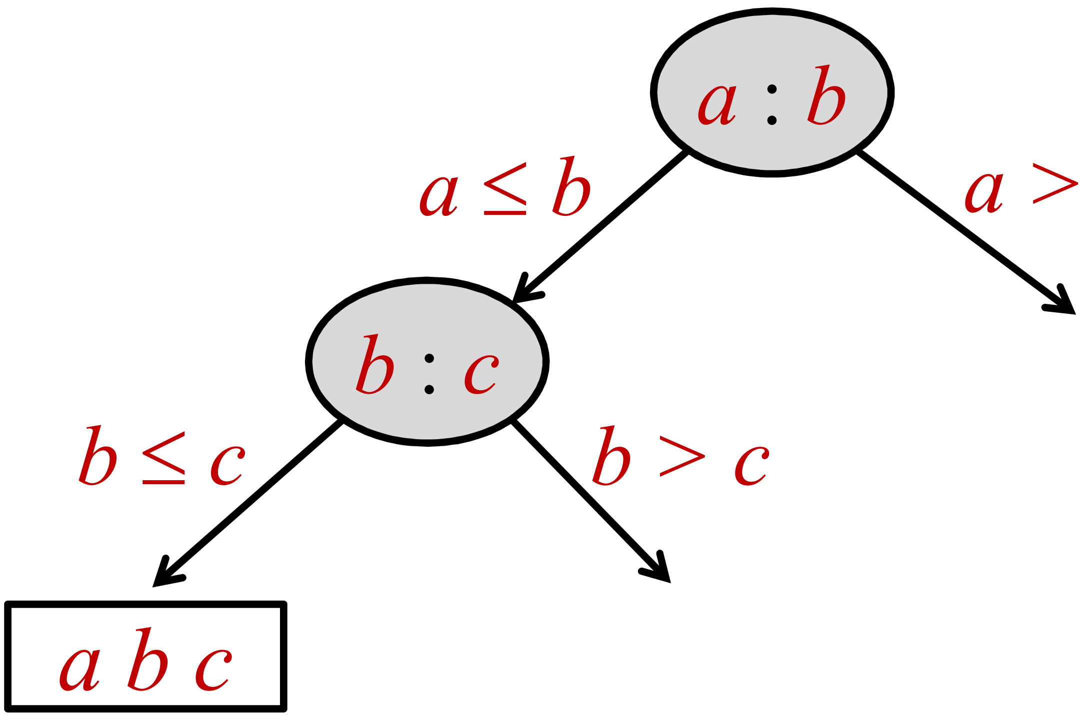
- Step 2: if ($a < b$) then compare b and c



Algorithm as a Decision-Tree

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

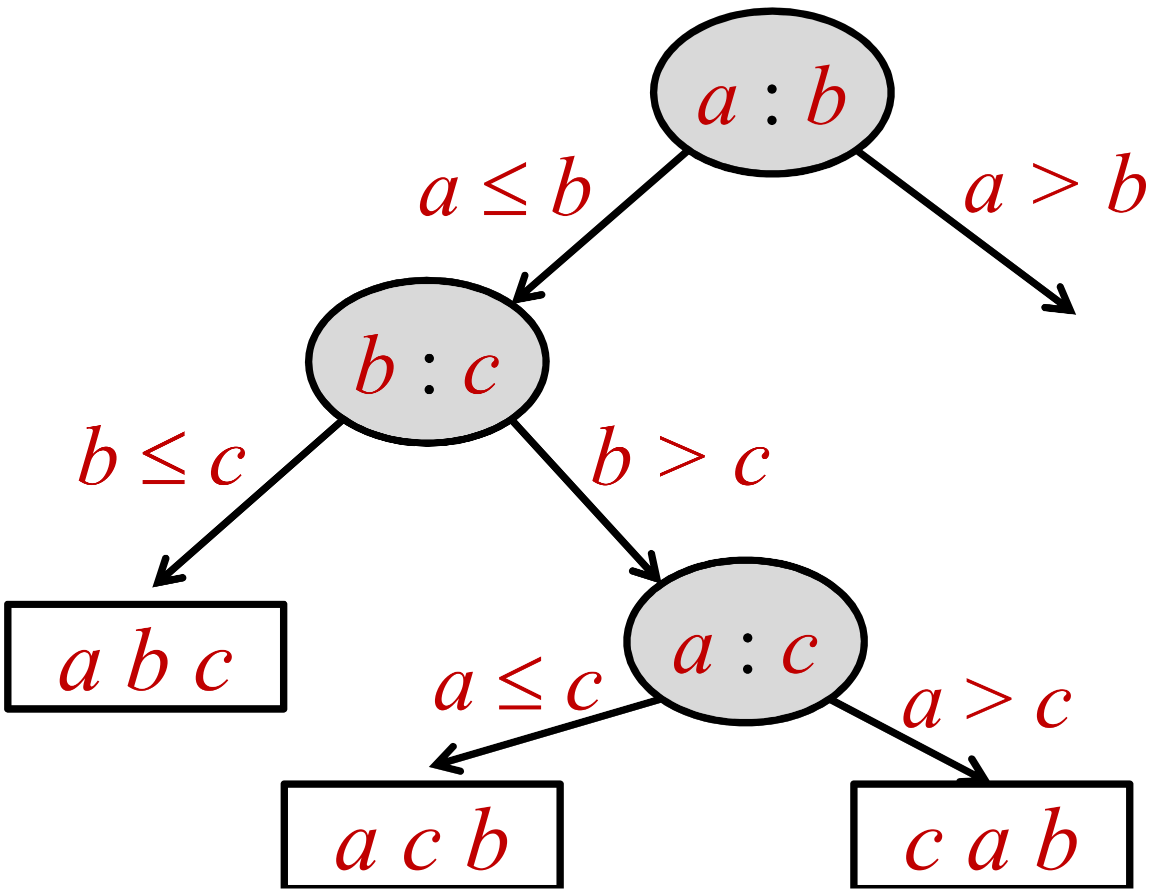
– Step 3: if ($a < b$) and ($b<c$) then output $<a, b, c>$



Algorithm as a Decision-Tree

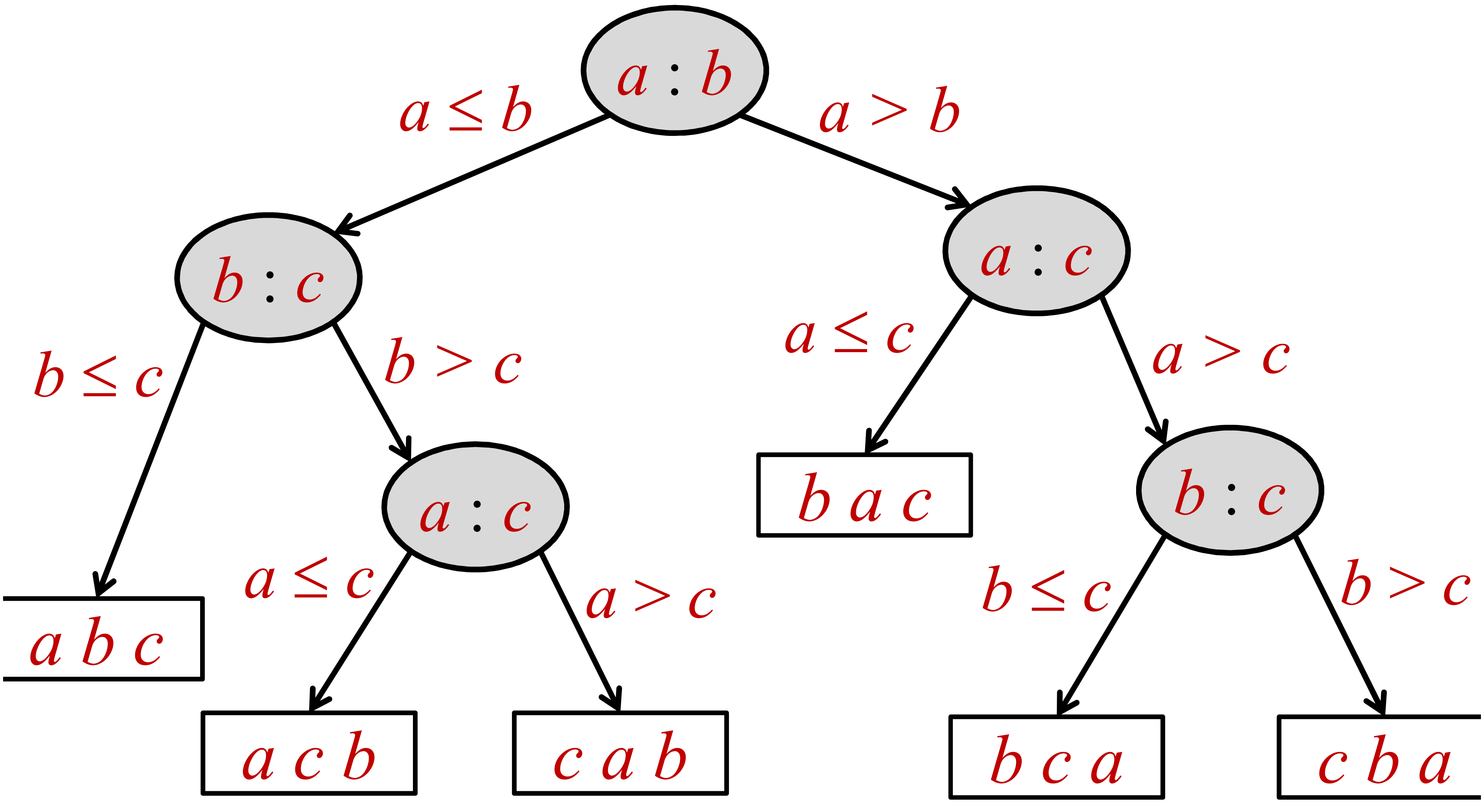
Consider sorting: $\{a, b, c\}$

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



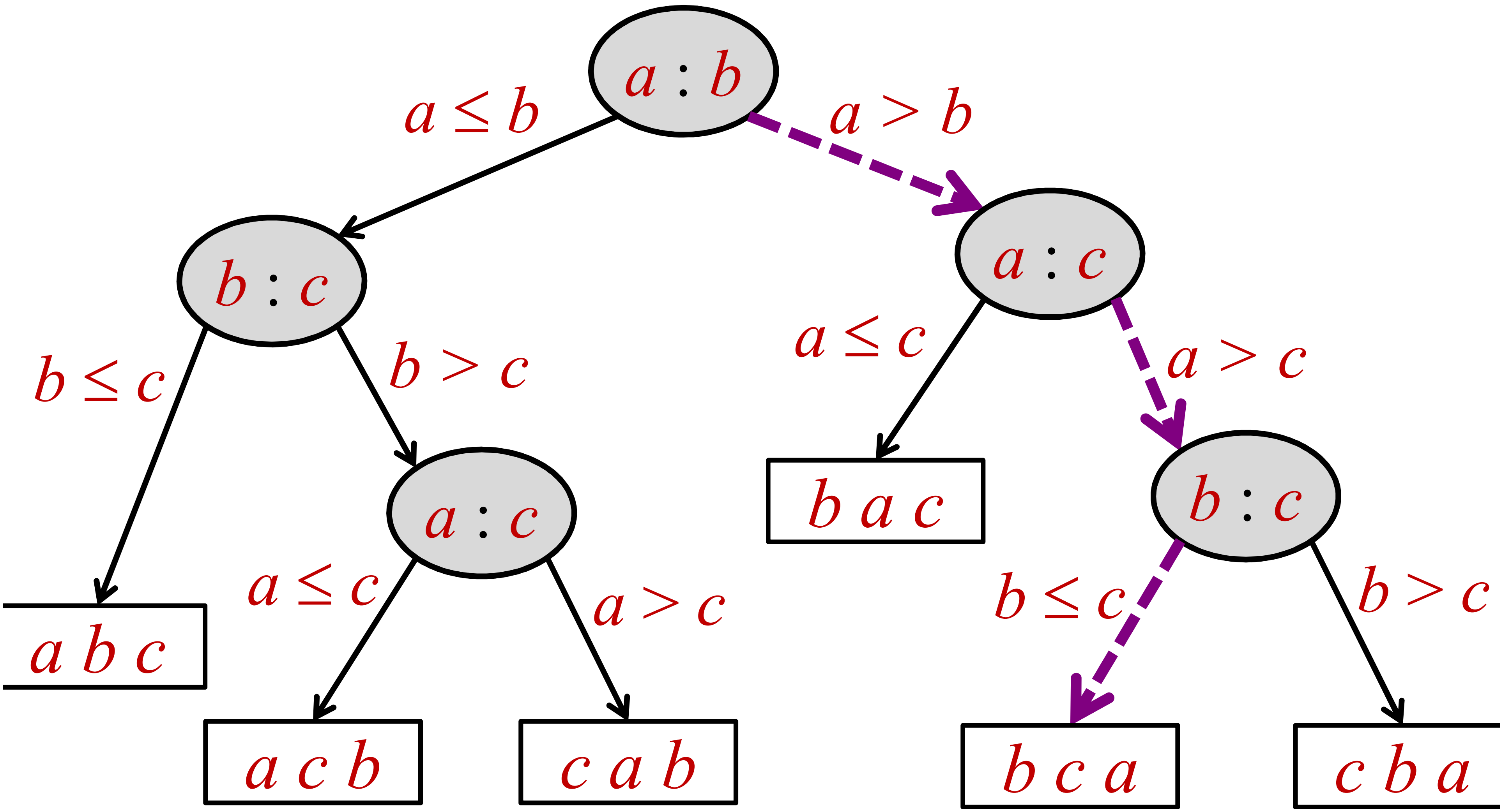
Algorithm as a Decision-Tree

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



Algorithm as a Decision-Tree

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



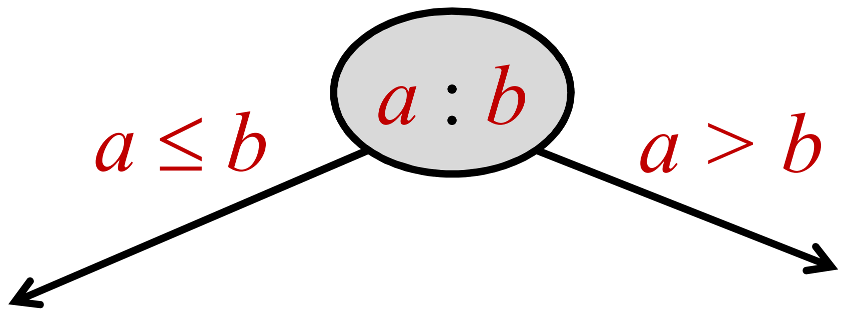
Algorithm as a Decision-Tree

A comparison-sort consists of:

A tree where:

Each node specifies two elements to compare.

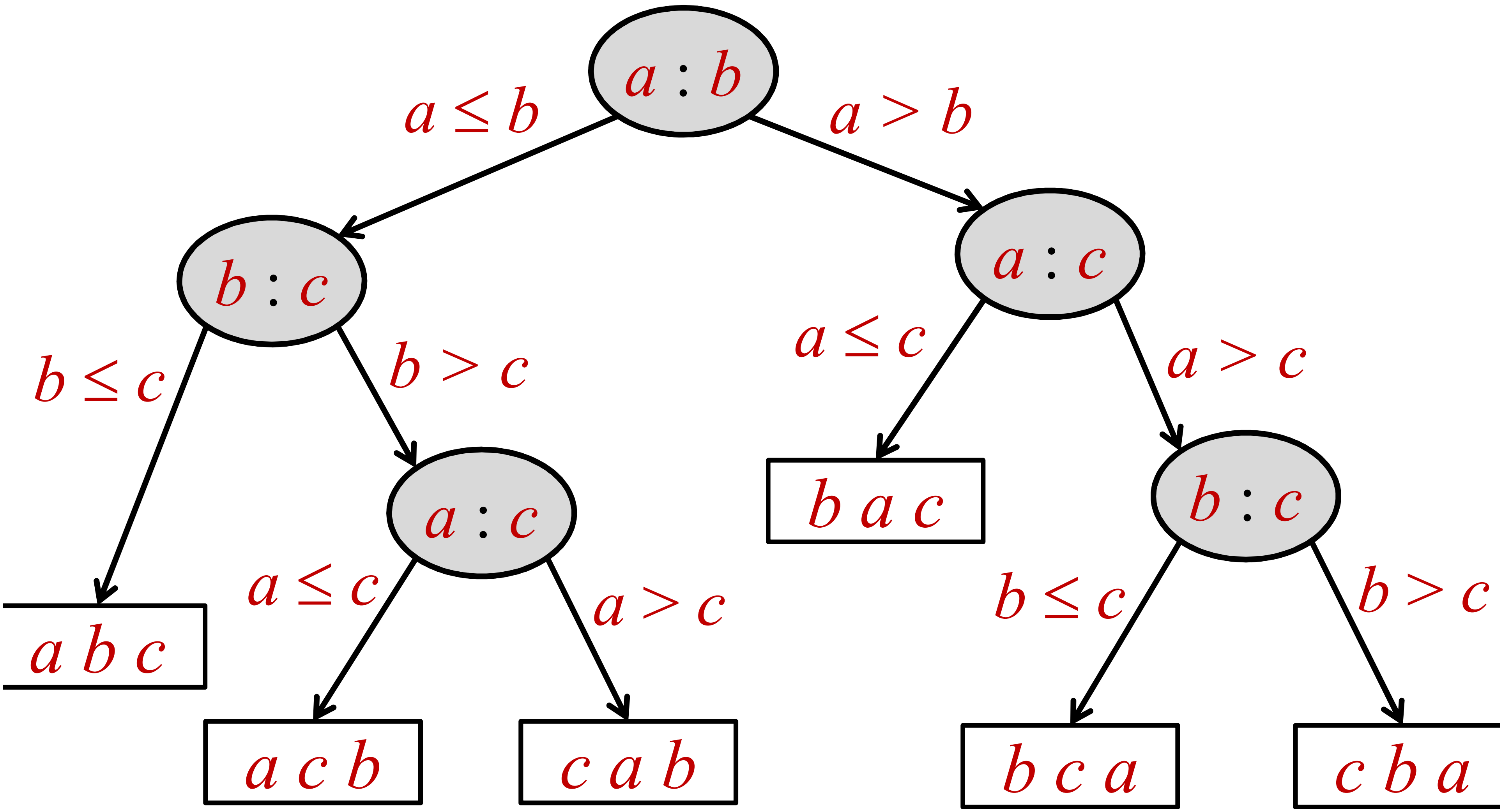
Each edge indicates which element is larger.



Every comparison-sort can be expressed this way.

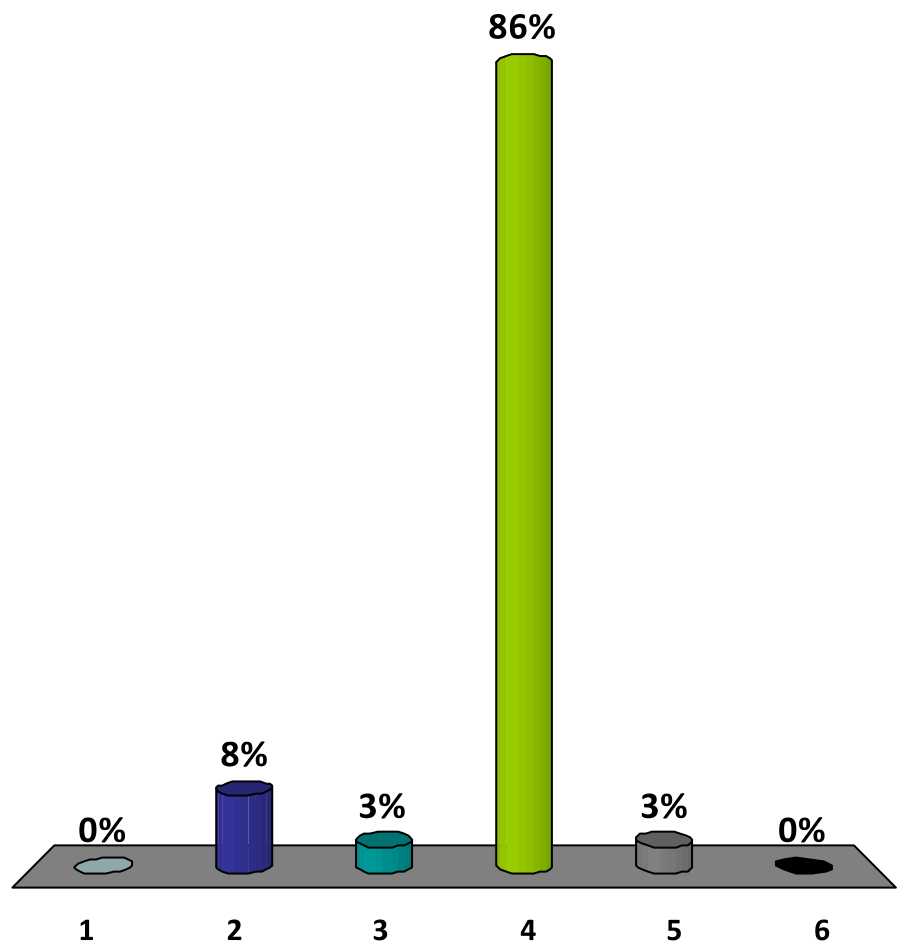
Algorithm as a Decision-Tree

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



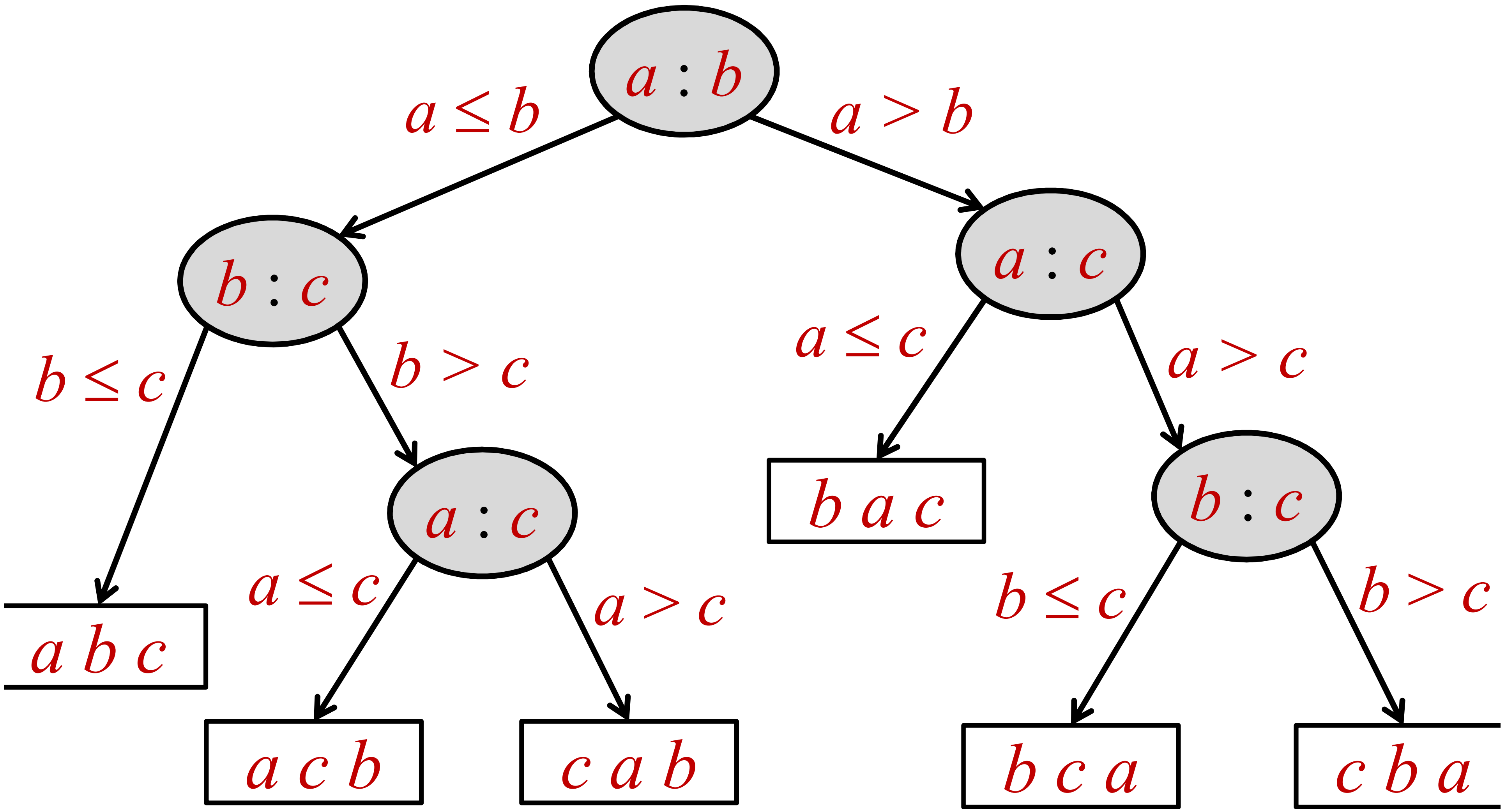
How many leaves are there in the *comparison-tree* for sorting $\{a, b, c, d, e\}$?

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!

  $a \quad b \quad c \quad d \quad e$

  $a \quad b \quad c \quad e \quad d$

  $a \quad b \quad d \quad c \quad e$

  $a \quad b \quad d \quad e \quad c$

  $a \quad b \quad e \quad c \quad d$

  $a \quad b \quad e \quad d \quad c$

  …

- 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!
- 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.

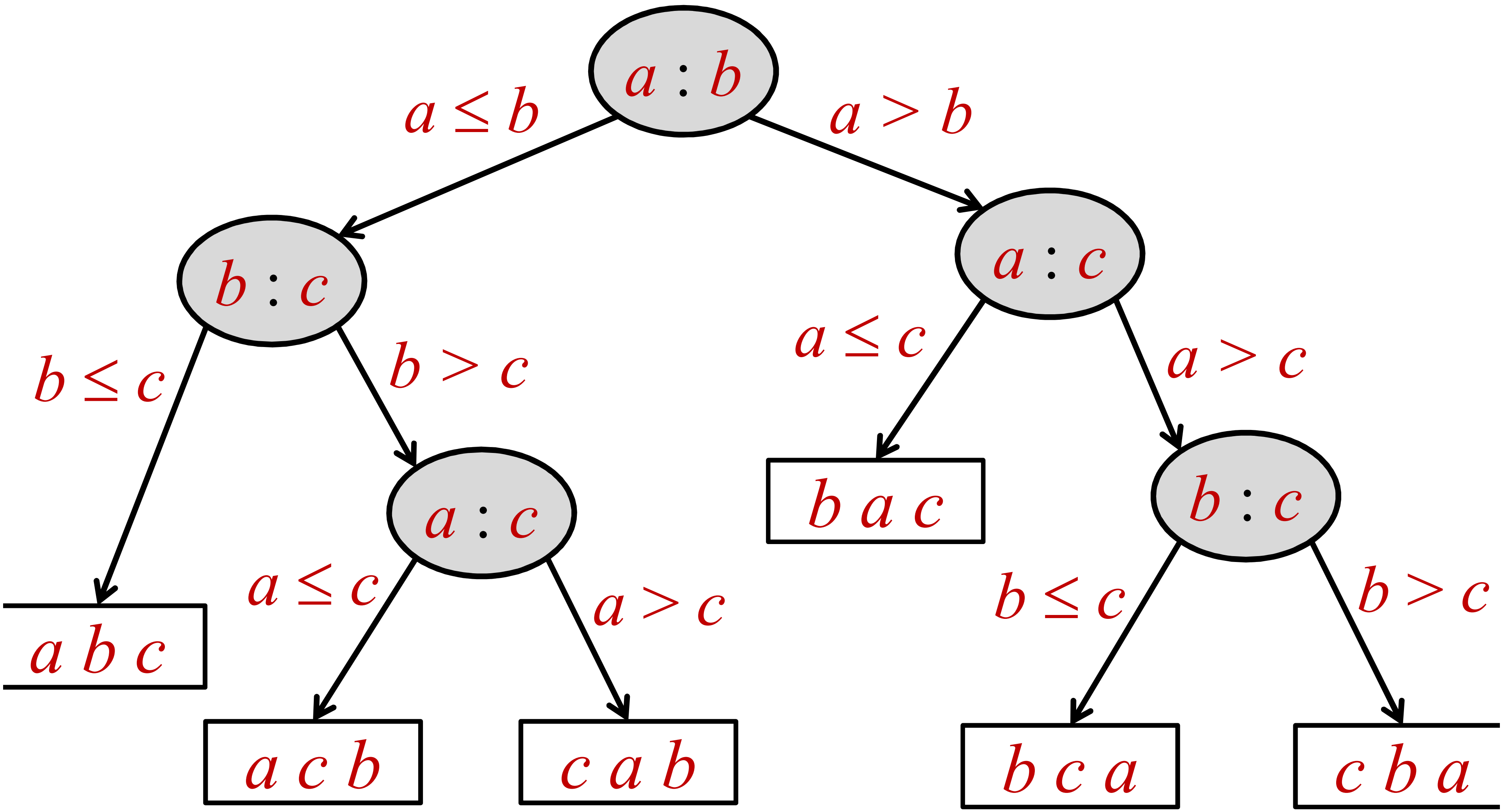
Running time of an algorithm:

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

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

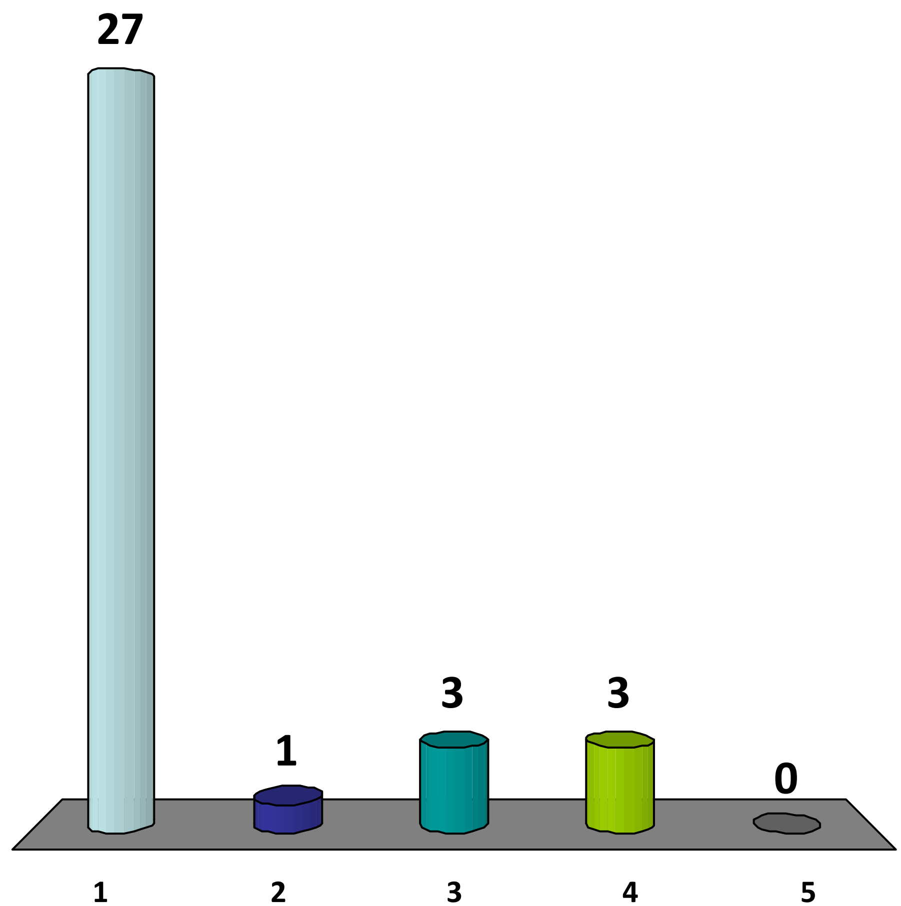
Algorithm as a Decision-Tree

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 $\Omega(n \log n)$.

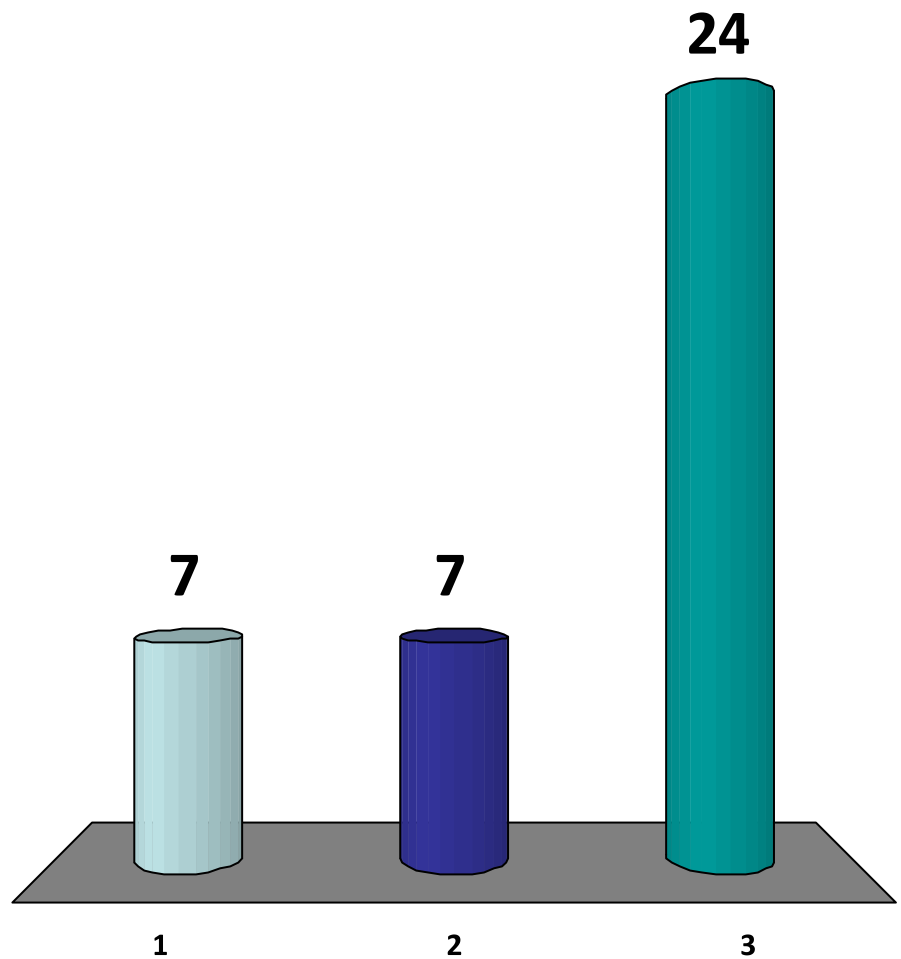
How many comparisons to sort?

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

Corollary: HeapSort and MergeSort are asymptotically optimal comparison sorts.

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, Heapsort, InsertionSort, DQuickSort
- For objects that implement `Comparable` interface.
- Every comparison sort requires time $\Omega(n \log n)$.
- MergeSort, HeapSort, and DQuickSort are asymptotically optimal.

Can we do faster non-comparison sorts?

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

Auxiliary data

Typically, databases contain pairs:

[key, data]

For example:

Age	Name
18	John
32	Sam
18	Mary
19	Bob

Sort by age!

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

Key properties:

- Only for sorting integers.
- Assume that all the integers in the input are in the range: [1, k]
- No comparisons!

Counting Sort

Input: $A[1..n]$ where $A[j] \in \{1, 2, \ldots, k\}$

Output: $B[1..n]$, sorted

Extra space: $C[1..k]$, initially $C[j] = 0$

Counting Sort

Input: $A[1..n]$ where $A[j] \in \{1, 2, \ldots, k\}$

Output: $B[1..n]$, sorted

Extra space: $C[1..\mathrm{k}]$, initially $C[j] = 0$

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

C =

1	2	3	4
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: Accumulating

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

Goal: $C[j] = \#(\text{keys} \leq j)$

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 =	0	0	0	0	0

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 =	0	0	3	0	0

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 =	0	0	3	0	0

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 =	0	0	3	0	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 =	0	0	3	0	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 =	0	3	3	0	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 =	0	3	3	0	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	0	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	0	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

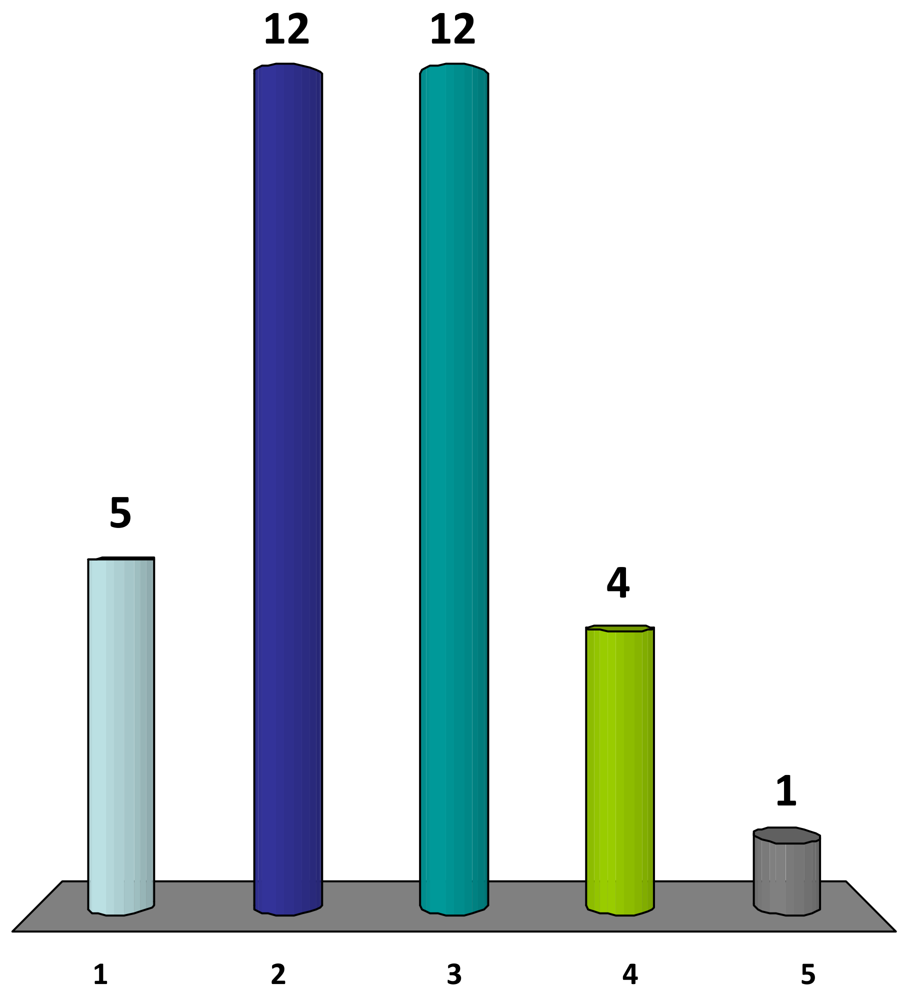
	1	2	3	4	5
A =	4	1	3	4	3
B =	1	3	3	4	4

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;
    }
```

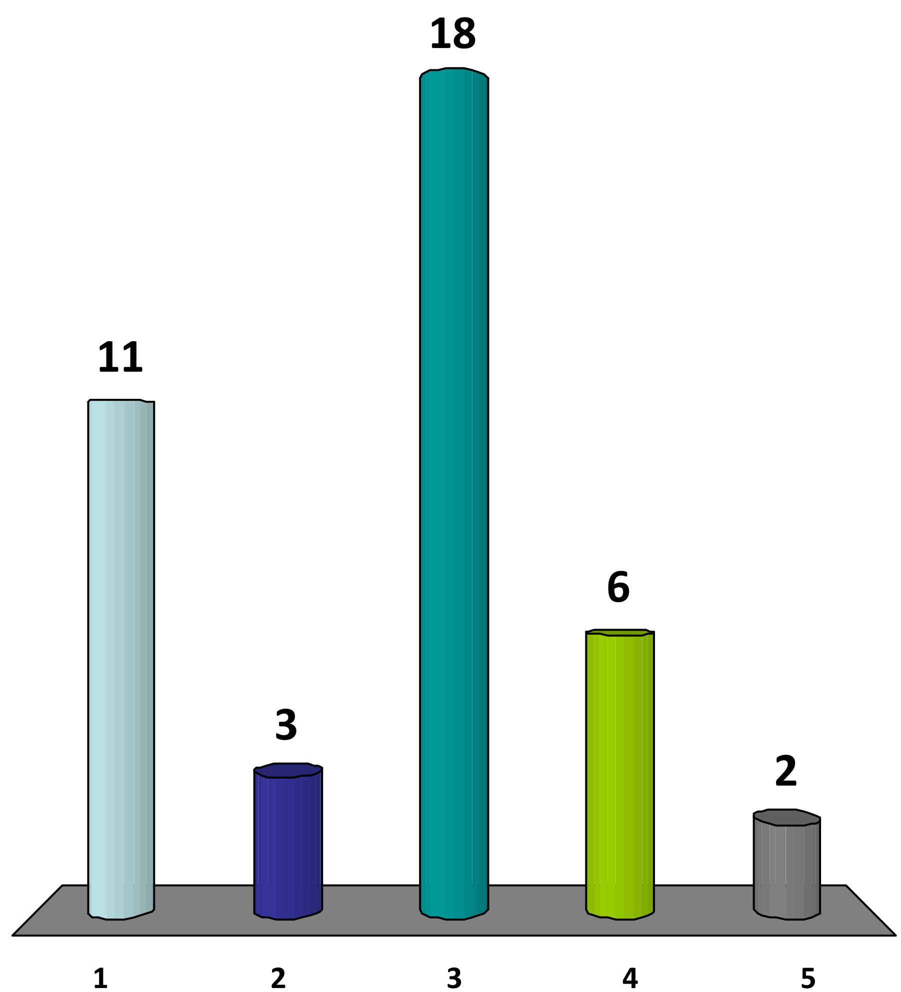
What is the running time of Counting Sort?

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



What is the space usage of Counting Sort?

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



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

Typically, databases contain pairs:

[key, data]

For example:

Age	Name
18	John
18	Mary
19	Bob
32	Sam

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

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.

Stability

Exercise:

- Is InsertionSort stable?
- Is QuickSort stable?

Radix Sort

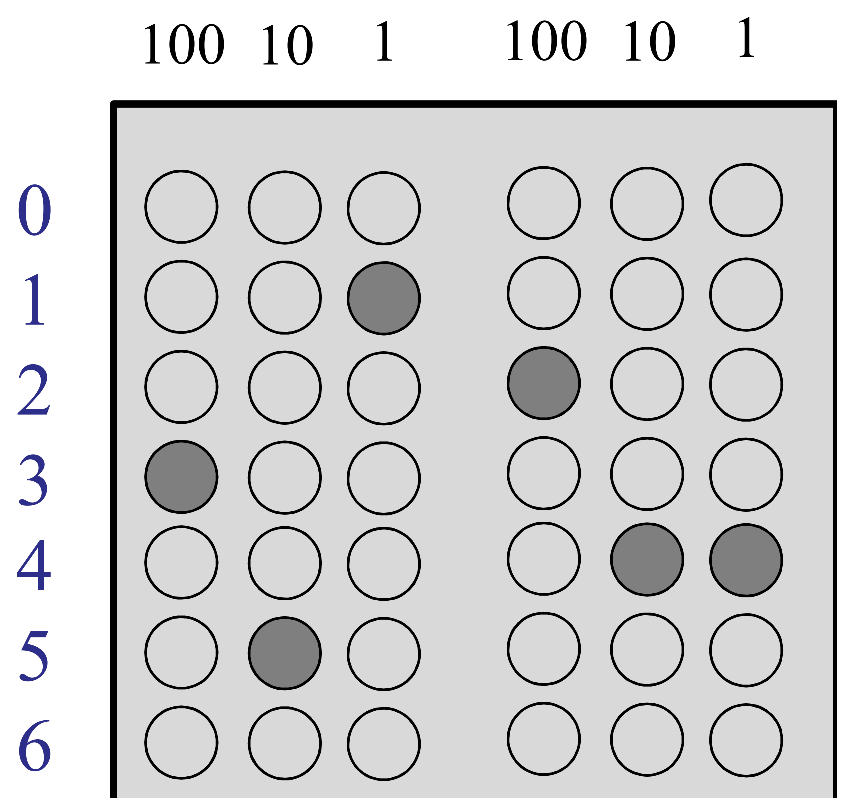
Digit-by-digit sorting:

– Originated at IBM

– Large numbers of punch-cards to sort

– Each pass through the machine can sort by only one digit.

351
244 =



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

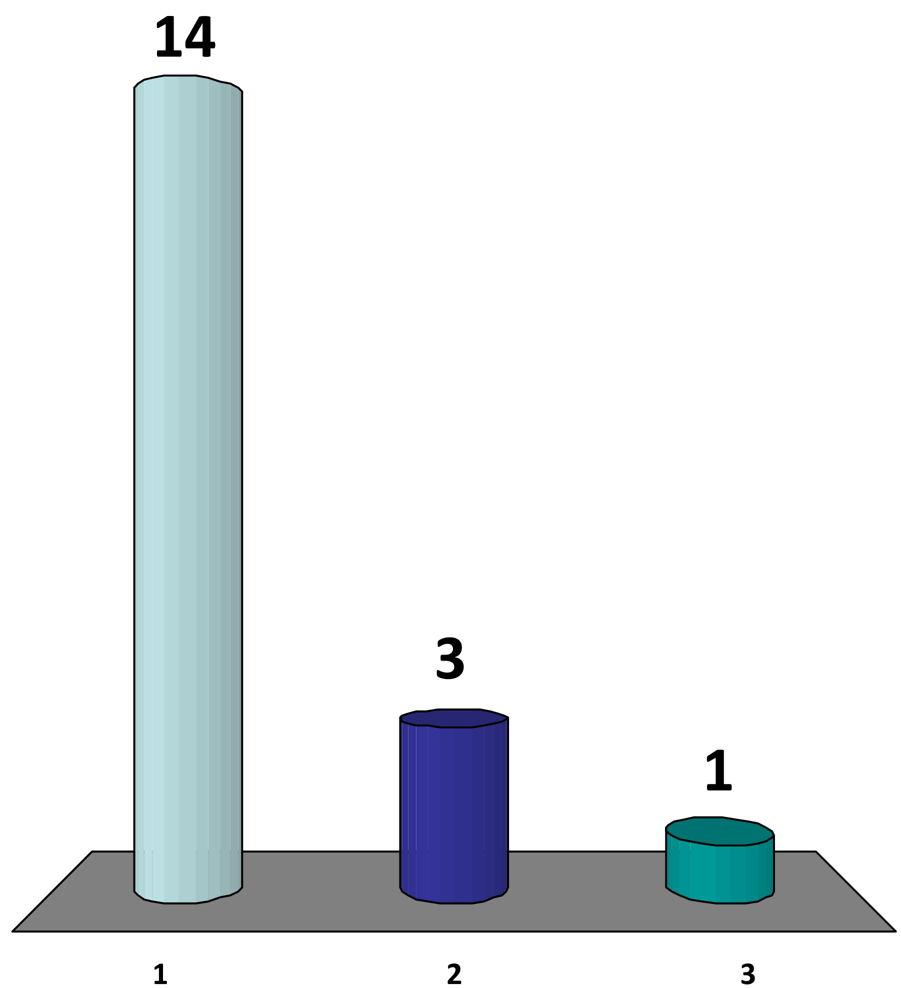
Which to sort first: least significant bit?

most significant bit?

doesn't matter?

Sort by:

1. Most significant bit
2. Least significant bit
3. Does not matter.



Radix Sort: 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**

Problem: have to sort each pile separately for next column.

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 Counting Sort for each column.
- Sort n words of b bits each.
- Each digit has r bits.
- Each word 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: $\mathrm{O}(n + 2^r)$

Space usage: $\mathrm{O}(2^r)$

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	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:

- $O(n \log\log n)$ time
- Efficient space
- Complicated and mostly theoretical

Quiz Basics

Quiz Basics

Basics:

- In class Friday (100 minutes).
- Open book (bring notes, textbook, etc.)
- Covers material through today.

Quiz Basics

Five questions:

1. Recurrences and Asymptotic Analysis
2. Multiple Choice
3. Java
4. Algorithms
5. Algorithms

Quiz Basics

Recurrences:

- Know the basic recurrences that we see over and over again.
- Know how to determine asymptotic running time from a recurrence.
- See Discussion Group 1 for examples

Asymptotics: big-O notation

- How to use it.
- What it means.

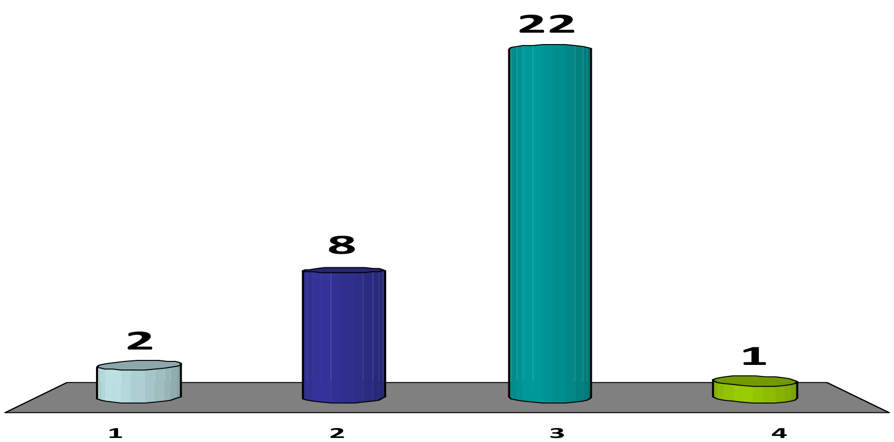
Quiz Basics

Multiple Choice Questions

- Understand how algorithms work
- Properties of algorithms / data structures
- Invariants

Which of the following is a defining property of a (2,3,4)-tree?

1. Every red node has only black nodes for children.
2. The height of a node’s children differ by at most 1.
3. Every leaf is at the same height.
4. Every node contains at least 4 keys.

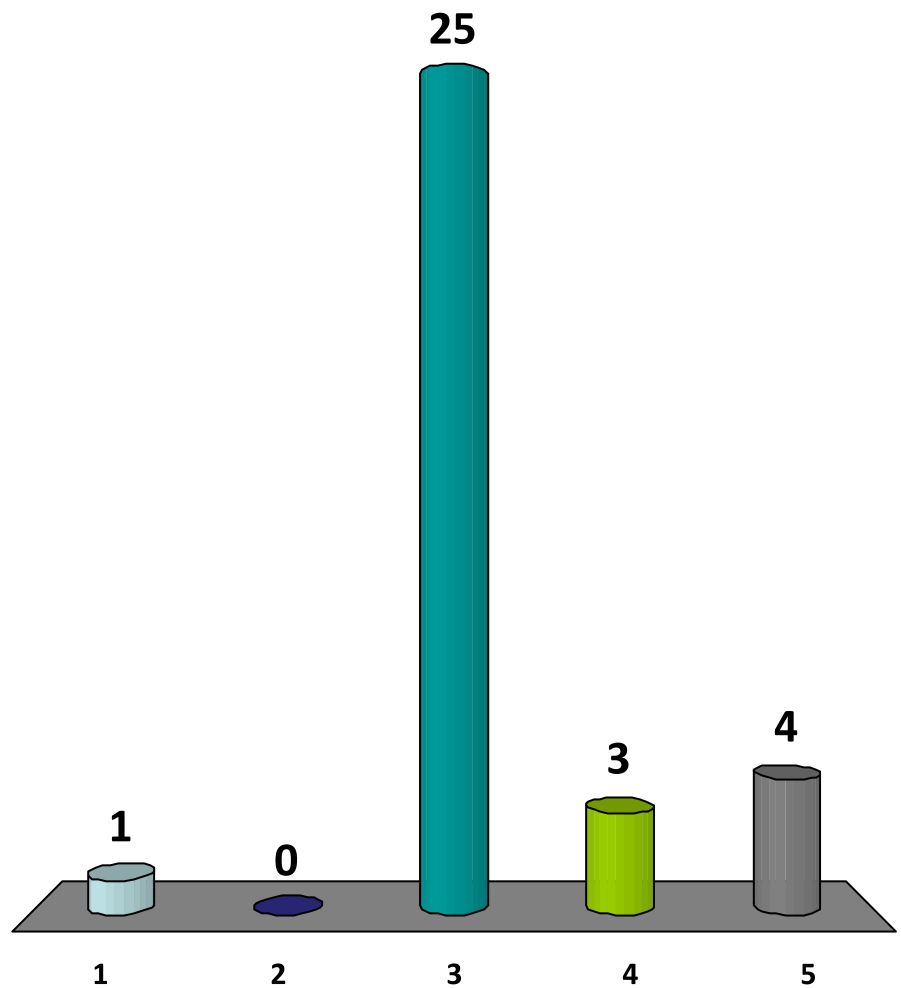


Which of the following is a good invariant for binary search for key k at node x?

1. $\min(x.\text{left}) < k < \max(x.\text{right})$ 4
2. $\min(x.\text{left}) \leq k \leq \max(x.\text{right})$ 7
3. $\text{pred}(x) < k < \text{succ}(x)$ 13
4. ✓ $\text{pred}(\min(x.\text{left})) < k < \text{succ}(\max(x.\text{right}))$ 3
5. $\text{pred}(\min(x.\text{left})) \leq k \leq \text{succ}(\max(x.\text{right}))$ 2

Which of the following best characterizes the depth of a balanced binary search tree?

1. < log(n) / 2
2. < log(n/2)
3. < log(n)
4. < log(2n)
5. < 2log(n)



Quiz Basics

Java Question

- Understand Java syntax / semantics
- Be able to read Java code and determine what it does.
- Basic object-oriented design (as in DG 1)
  - Abstract Data Types and encapsulation
- Classes and interfaces
- Inheritance
  - extend class
  - implement interfaces
- Exception-handling (as in Recitation 1)

Quiz Basics

Algorithms Questions (2)

Sorting algorithms:

1. InsertionSort
2. MergeSort
3. HeapSort
4. QuickSort (with simple pivot and randomized pivot)
5. Counting Sort
6. Radix Sort (basic idea, but not details)

Quiz Basics

Algorithms Questions (2)

Divide-and-Conquer techniques:

1. Binary search
2. Peak finding
3. MergeSort
4. QuickSort
5. Randomized Select

Quiz Basics

Algorithms Questions (2)

Search Trees:

1. Basic binary search trees
   - In-order-traversal
   - Search / Insert / Delete
2. AVL trees
   - Height-balance
   - Rotations
3. B-trees
   - Search / Insert / node-split   (but not delete)

Quiz Basics

Algorithms Questions (2)

Other topics:

1. Heaps
   - Constructing
   - Inserting / Deleting
   - ExtractMax
2. Priority Queues
3. Stacks & Queues (as examples of abstract data types)
3. Randomized Select

Quiz Summary

Remember:

- Don't stress too much!
- The quiz is not designed to be a <u>killer, brutal, miserable</u> experience.   It should be <u>fun</u>.
- It <u>is</u> designed to test whether you have been paying attention.
- If you do well on the in-class clicker-questions, you should do well on the quiz!
- Goal: everyone gets a perfect score!

But: learn the material!