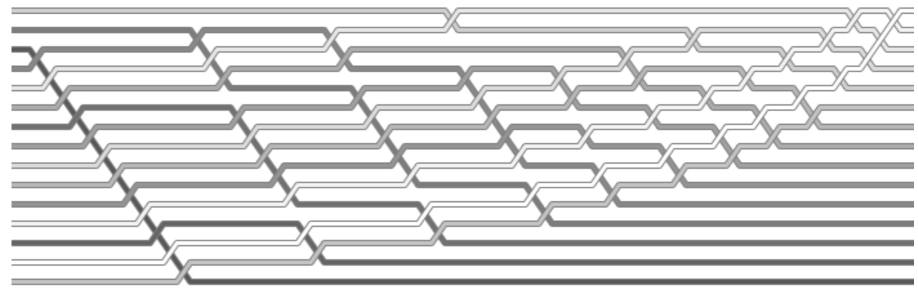
Sorting

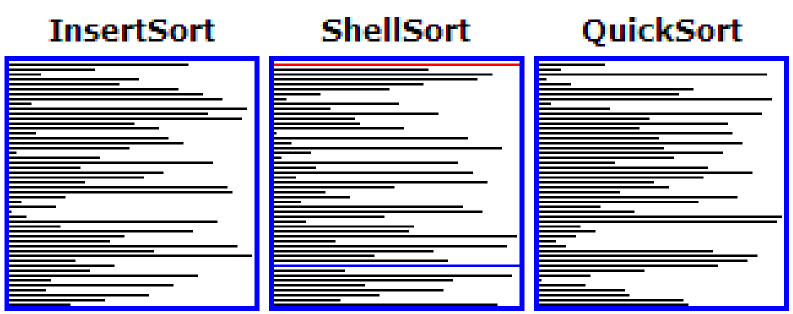
Almost half of <u>all</u> CPU cycles are spent on sorting!

- Input: array X[1..n] of integers
- Output: sorted array (permutation of input)

In: 5,2,9,1,7,3,4,8,6
Out: 1,2,3,4,5,6,7,8,9



- Assume WLOG all input numbers are unique
- Decision tree model ⇒ count comparisons "<"



Lower Bound for Sorting

Theorem: Sorting requires $\Omega(n \log n)$ time

Proof: Assume WLOG unique numbers

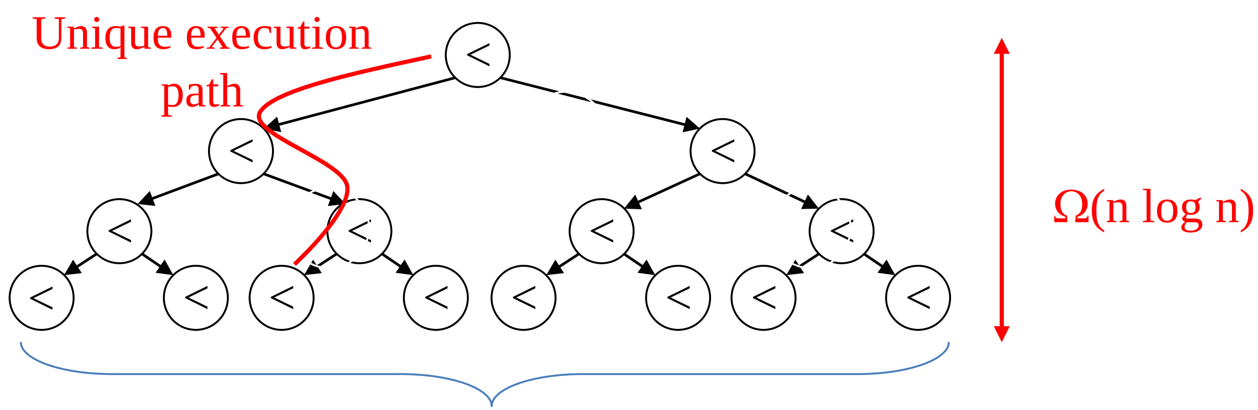
$\Rightarrow$ n! different permutations

$\Rightarrow$ comparison decision tree has n! leaves

$\Rightarrow$ tree height $\geq \log(n!) > \log\left(\left(\frac{n}{e}\right)^n\right) = n \cdot \log\left(\frac{n}{e}\right) = \Omega(n \log n)$

$\Rightarrow \Omega(n \log n)$ decisions / time necessary to sort

Non-existence proof!



n! permutations (i.e., distinct sorted outcomes)

Sorting Algorithms (Sorted!)

1. AKS sort
2. Bead sort
3. Binary tree sort
4. Bitonic sorter
5. Block sort
6. Bogosort
7. Bozo sort
8. Bubble sort
9. Bucket sort
10. Burstsort
11. Cocktail sort
12. Comb sort
13. Counting sort
14. Cubesort
15. Cycle sort
16. Flashsort
17. Franceschini's sort
18. Gnome sort
19. Heapsort
20. In-place merge sort
21. Insertion sort
22. Introspective sort
23. Library sort
24. Merge sort
25. Odd-even sort
26. Patience sorting
27. Pigeonhole sort
28. Postman sort
29. Quantum sort
30. Quicksort
31. Radix Sort
32. Sample sort
33. Selection sort
34. Shaker sort
35. Shell sort
36. Simple pancake sort
37. Sleep sort
38. Smoothsort
39. Sorting network
40. Spaghetti sort
41. Splay sort
42. Spreadsort
43. Stooge sort
44. Strand sort
45. Timsort
46. Tree sort
47. Tournament sort
48. UnShuffle Sort

Sorting Algorithms

Q: Why so many sorting algorithms?

A: There is no "best" sorting algorithm!

Some considerations:

- Worst case?
- Average case?
- In practice?
- Input distribution?
- Near-sorted data?
- Stability?
- In-situ?
- Randomized?
- Stack depth?
- Internal vs. external?
- Pipeline compatible?
- Parallelizable?
- Locality?
- Online

Problem: Given n pairs of integers (x_i, y_i), where $0 \le x_i \le n$ and $1 \le y_i \le n$ for $1 \le i \le n$, find an algorithm that sorts all n ratios x_i / y_i in linear time $O(n)$.

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations

Problem: Given n integers, find in $O(n)$ time the majority element (i.e., occurring $\geq n/2$ times, if any).

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations

Problem: Given n objects, find in $O(n)$ time the majority element (i.e., occurring $\geq n/2$ times, if any), using only equality comparisons ($=$).

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations

Problem: Given n integers, find both the maximum and the next-to-maximum using the least number of comparisons (exact comparison count, not just $O(n)$).

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations

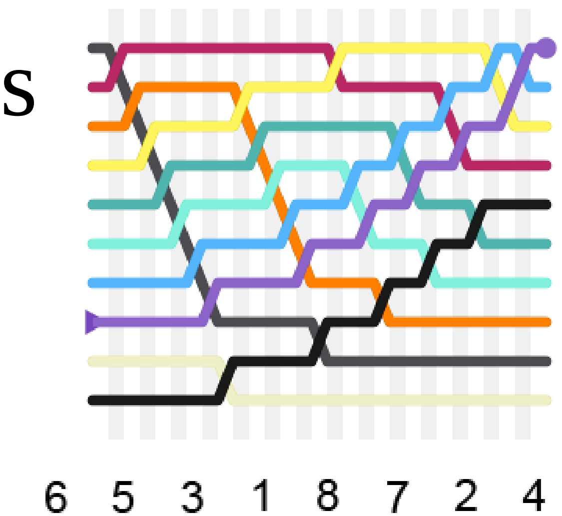
Bubble Sort

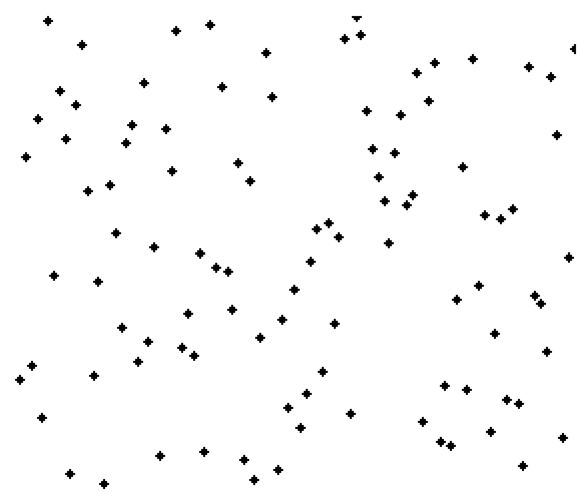
Input: array X[1..n] of integers

Output: sorted array (monotonic permutation)

Idea: keep swapping adjacent pairs

```
until array X is sorted do
    for i=1 to n-1
        if X[i+1]<X[i]
            then swap(X,i,i+1)
```





- $O(n^2)$ time worst-case,
  but sometimes faster
- Adaptive, stable, in-situ, slow

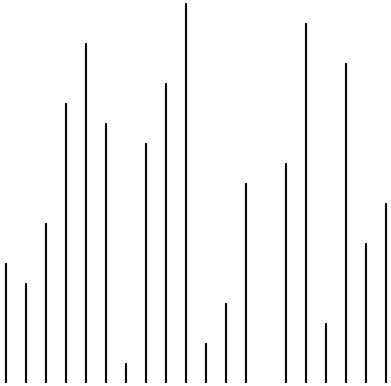
Odd-Even Sort

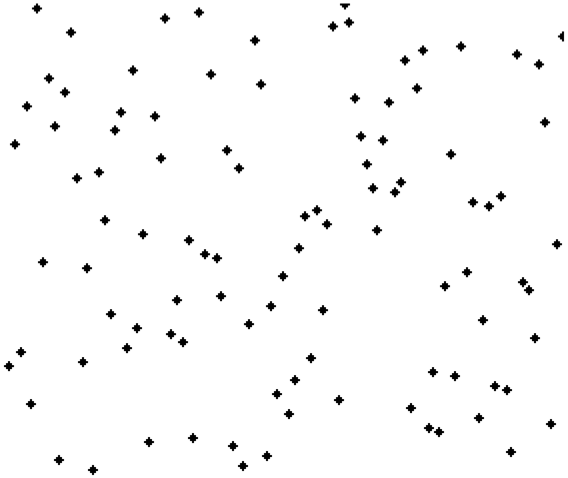
Input: array X[1..n] of integers

Output: sorted array (monotonic)

Idea: swap even and odd pairs

```
until array X is sorted do
    for even i=1 to n-1
        if X[i+1]<X[i] swap(X,i,i+1)
    for odd i=1 to n-1
        if X[i+1]<X[i] swap(X,i,i+1)
```





- $O(n^2)$ time worst-case,
  but faster on near-sorted data
- Adaptive, stable, in-situ, parallel

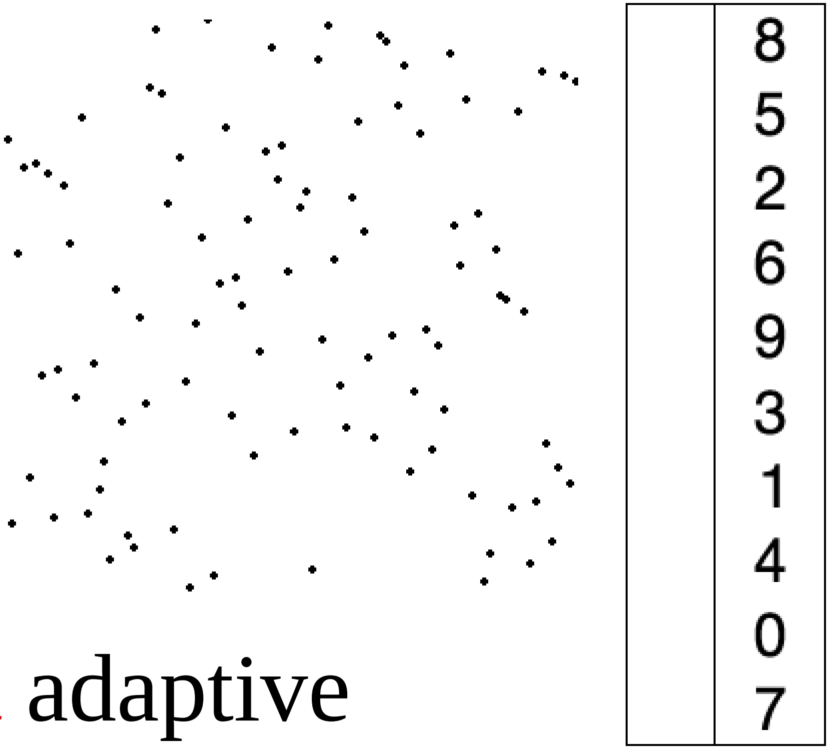
Selection Sort

Input: array X[1..n] of integers

Output: sorted array (monotonic permutation)

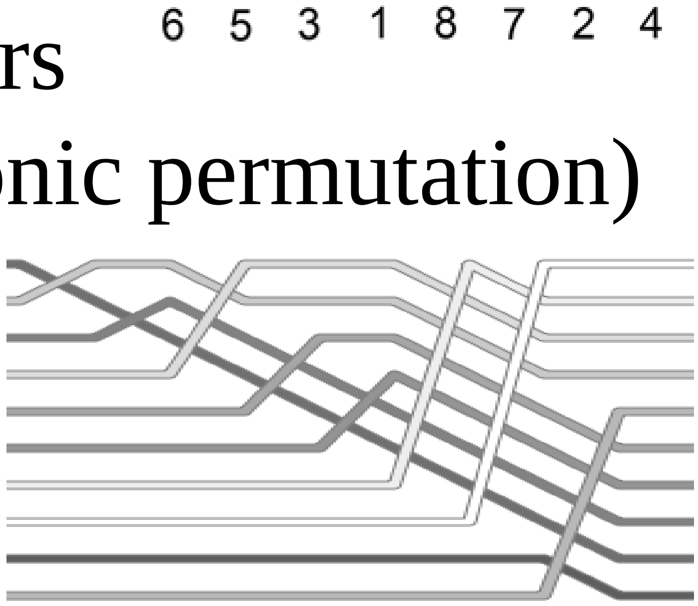
Idea: move the largest to current pos

```
for i=1 to n-1
   let X[j] be largest
         among X[i..n]
   swap(X,i,j)
```



- $\Theta(n^2)$ time worst-case
- Stable, in-situ, simple, not adaptive
- Relatively fast (among quadratic sorts)

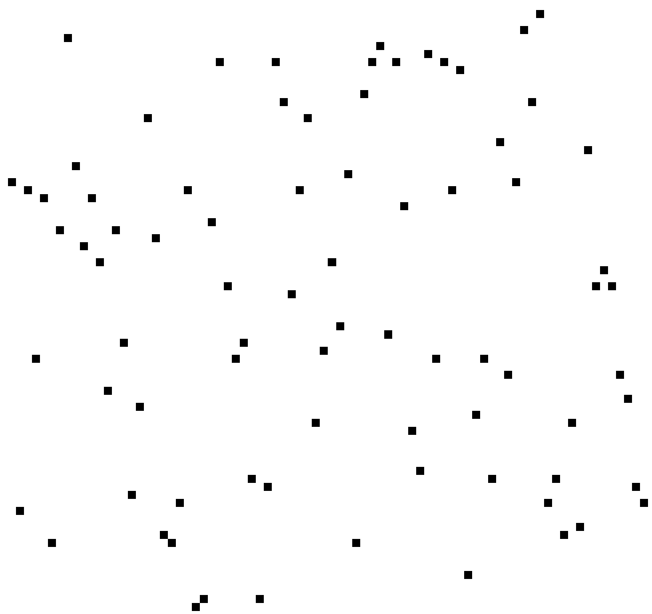
Insertion Sort



- Input: array X[1..n] of integers
- Output: sorted array (monotonic permutation)

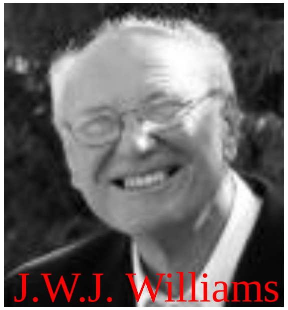
Idea: insert each item into list

```
for i=2 to n
   insert X[i] into the
         sorted list X[1..(i-1)]
```



- $O(n^2)$ time worst-case
- $O(nk)$ where k is max dist of any item from final sorted pos
- Adaptive, stable, in-situ, online

Heap Sort

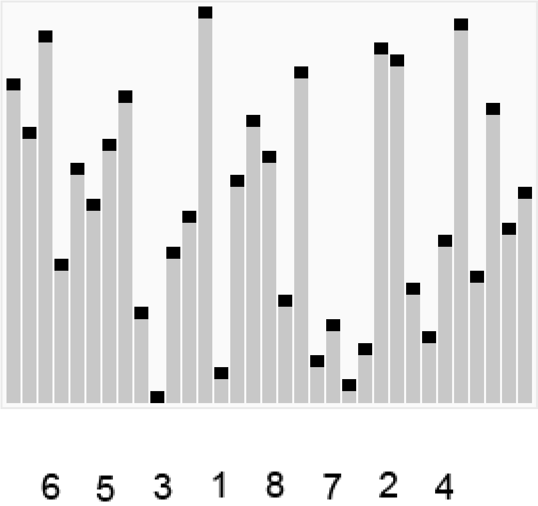


Input: array X[1..n] of integers

Output: sorted array (monotonic)

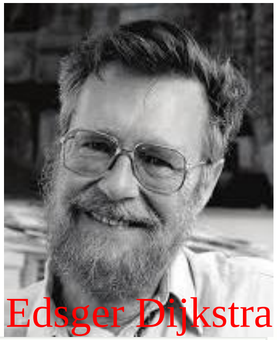
Idea: exploit a heap to sort

```
InitializeHeap
For i=1 to n HeapInsert(X[i])
For i=1 to n do
   M=HeapMax; Print(M)
   HeapDelete(M)
```



- $\Theta(n \log n)$ optimal time
- Not stable, not adaptive, in-situ

SmoothSort

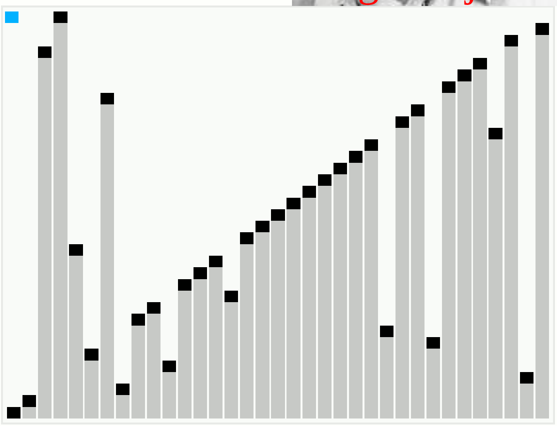


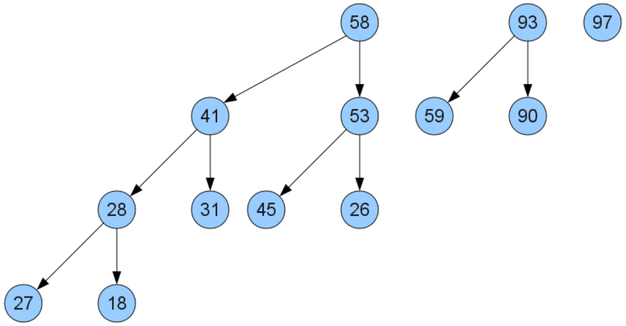
Input: array X[1..n] of integers

Output: sorted array (monotone)

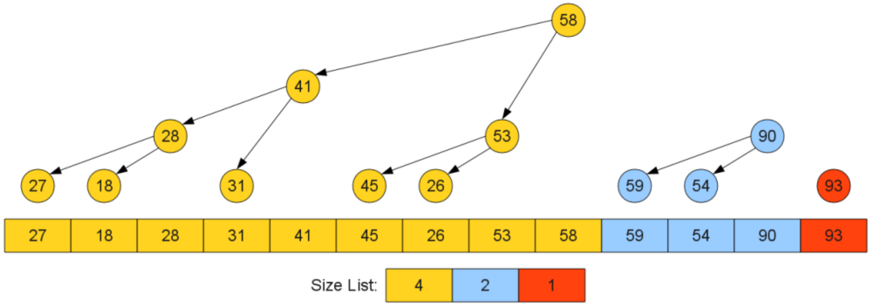
Idea: adaptive heapsort

```
InitializeHeaps
for i=1 to n HeapsInsert(X[i])
for i=1 to n do
   M=HeapsMax; Print(M)
   HeapsDelete(M)
```





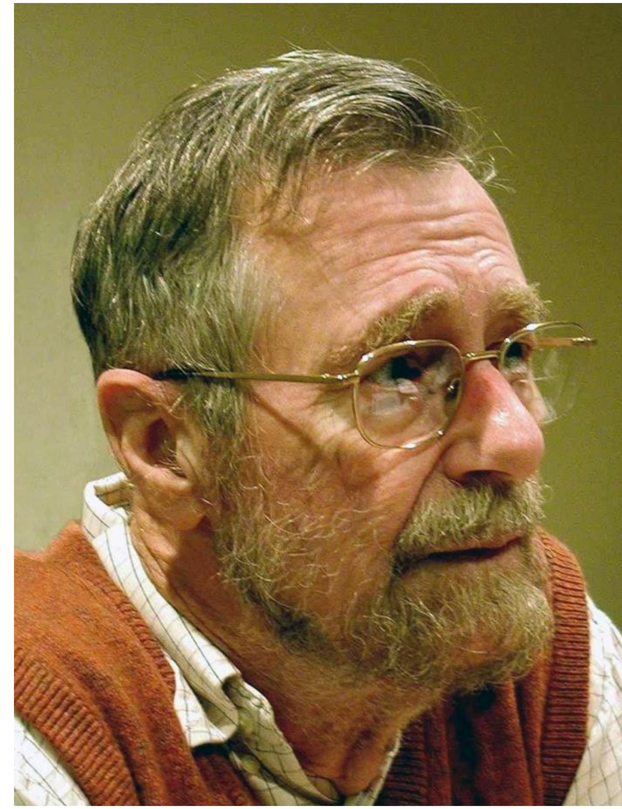
- Uses multiple (Leonardo) heaps
- O(n log n)
- O(n) if list is mostly sorted
- Not stable, adaptive, in-situ

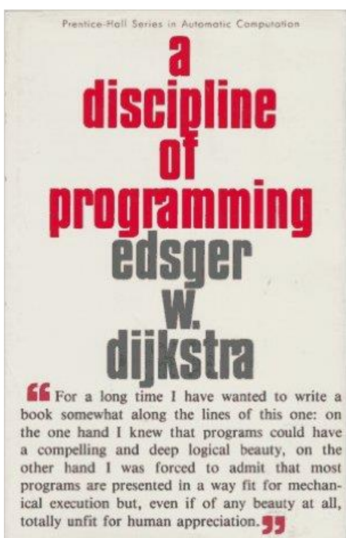


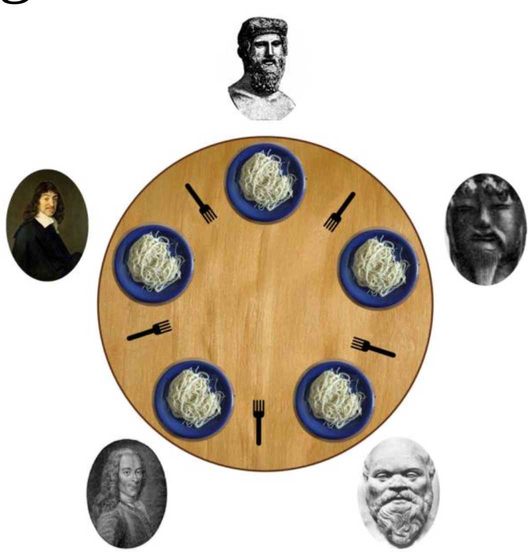
Historical Perspectives

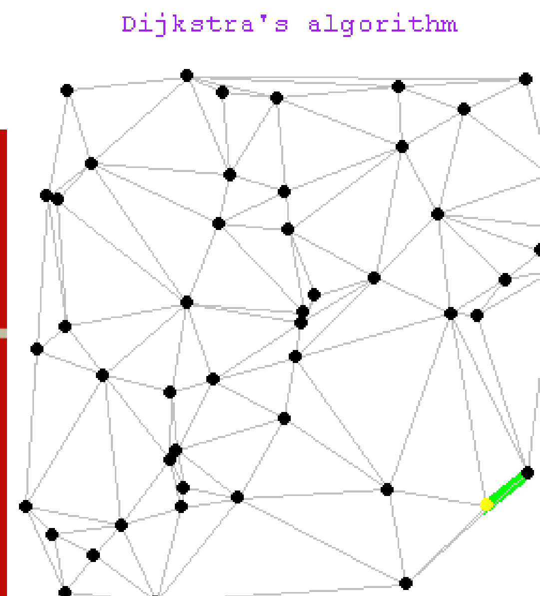
Edsger W. Dijkstra (1930-2002)

- Pioneered software engineering, OS design
- Invented concurrent programming, mutual exclusion / semaphores
- Invented shortest paths algorithm
- Advocated structured (GOTO-less) code
- Stressed elegance & simplicity in design
- Won Turing Award in 1972

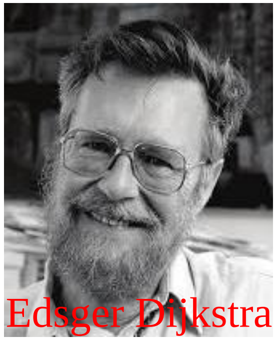






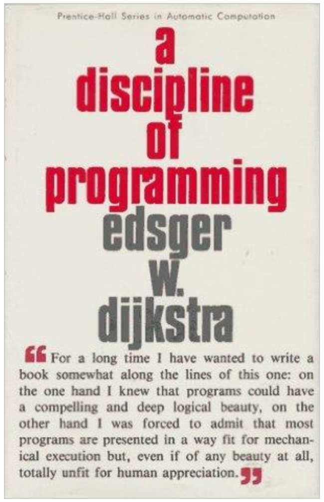


Quotes by Edsger W. Dijkstra (1930-2002)



Edsger Dijkstra

- "Computer science is no more about computers than astronomy is about telescopes."
- "If debugging is the process of removing software bugs, then programming must be the process of putting them in."
- "Testing shows the presence, not the absence of bugs."
- "Simplicity is prerequisite for reliability."
- "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense."
- "Object-oriented programming is an exceptionally bad idea which could only have originated in California."
- "Elegance has the disadvantage, if that's what it is, that hard work is needed to achieve it and a good education to appreciate it."

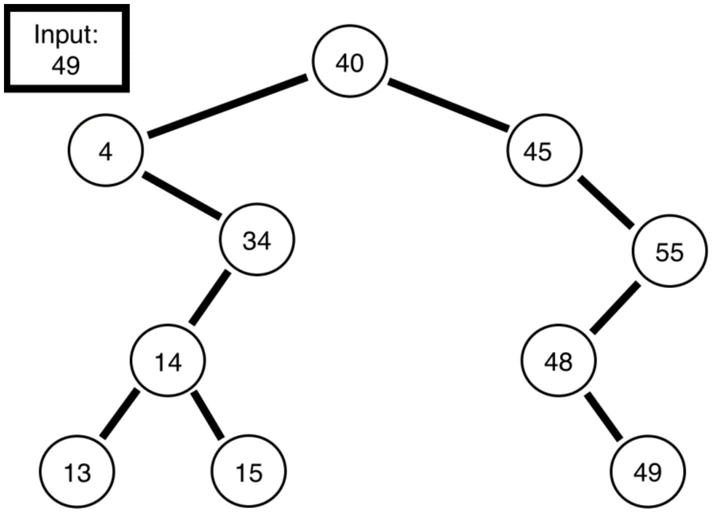


Generalizing Heap Sort

Input: array X[1..n] of integers

Output: sorted array

```
InitializeTree
For i=1 to n
    TreeInsert(X[i])
For i=1 to n do
    M=TreeMax; Print(M)
    TreeDelete(M)
```



- Observation: other data structures can work here!
- Ex: replace heap with any height-balanced tree
- Retains O(n log n) worst-case time!

Tree Sort

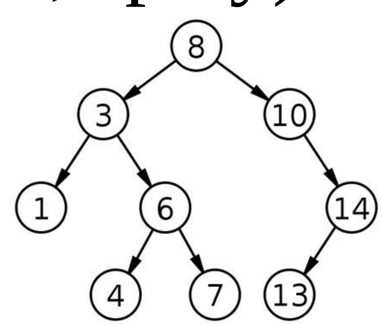
Input: array X[1..n] of integers

Output: sorted array (monotonic)

Idea: populate a tree & traverse

```
InitializeTree
for i=1 to n TreeInsert(X[i])
traverse tree in-order
   to produce sorted list
```

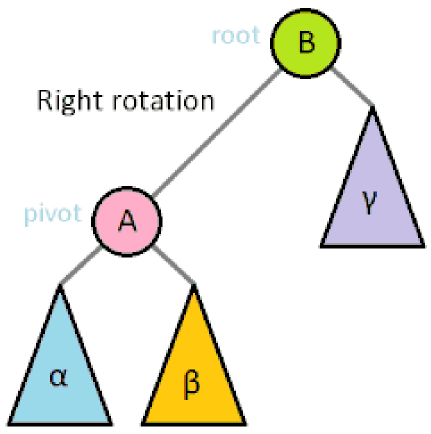
- Use balanced tree (AVL, B, 2-3, splay)
- O(n log n) time worst-case
- Faster for near-sorted inputs
- Stable, adaptive, simple



□□□□□□□□□

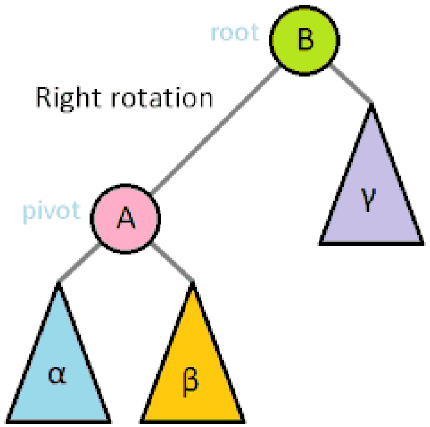
B-Tree Sort

- Multi-rotations occur infrequently
- Rotations don't propagate far
- Larger tree $\Rightarrow$ fewer rotations
- Same for other height-balanced trees
- Non-balanced search trees average $O(\log n)$ height



AVL-Tree Sort

- Multi-rotations occur infrequently
- Rotations don't propagate far
- Larger tree $\Rightarrow$ fewer rotations
- Same for other height-balanced trees
- Non-balanced trees average O(log n) height



Merge Sort

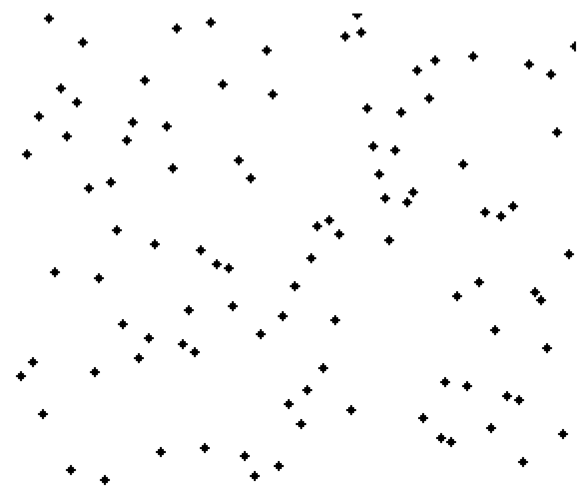
Input: array X[1..n] of integers

Output: sorted array (monotonic)

Idea: sort sublists & merge them


```
MergeSort(X,i,j)
  if i<j then m=⌊(i+j)/2⌋
      MergeSort(X,i..m)
      MergeSort(X,m+1..j)
      Merge(X,i..m,m+1..j)
```

- $T(n)=2T(n/2)+n=\Theta(n \log n)$ optimal!
- Stable, parallelizes, not in-situ
- Can be made in-situ & stable



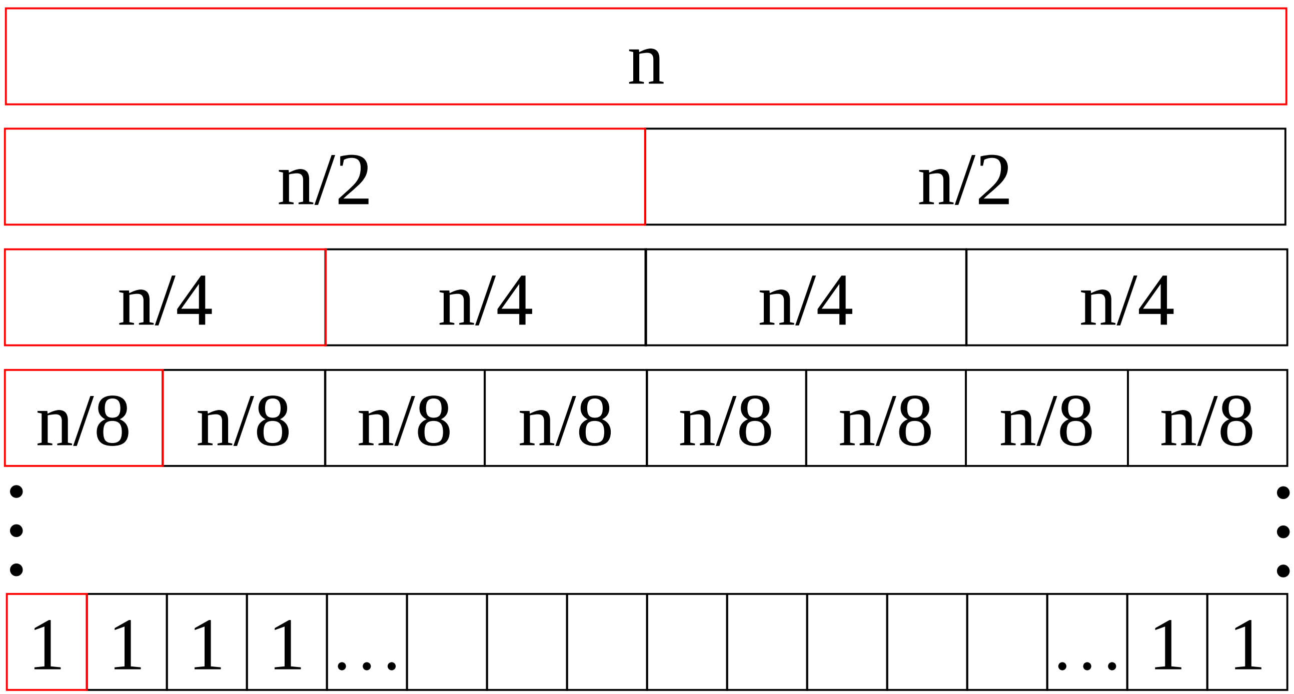
Merge Sort

Theorem: MergeSort runs within time

$\Theta(n \log n)$ which is optimal.

Proof: Even-split divide & conquer:

$$T(n) = 2 \cdot T(n/2) + n$$



$\Rightarrow$ n total / level

log n levels of recursion

Total time is $O(n \log n)$; $\Omega(n \log n) \Rightarrow \Theta(n \log n)$

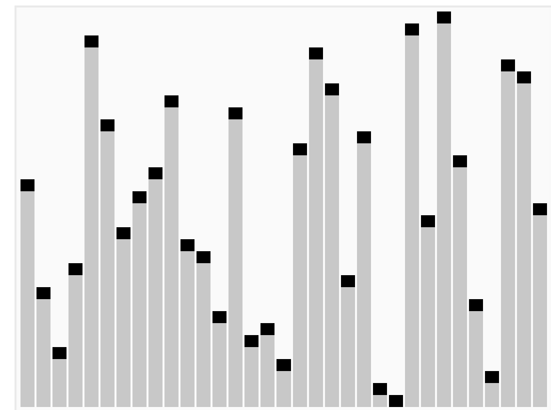
Quicksort

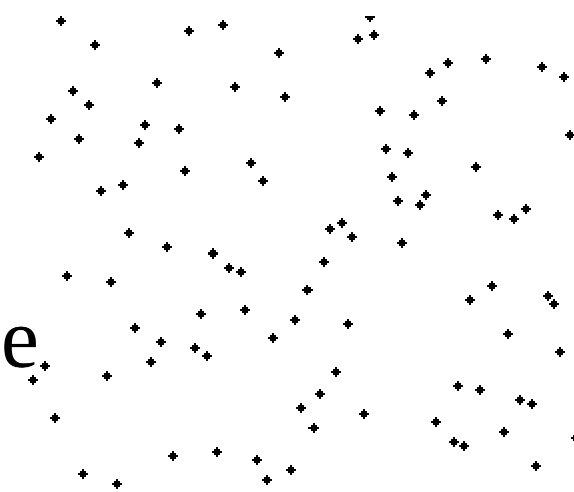
Input: array X[1..n] of integers

Output: sorted array (monotonic)

Idea: sort two sublists around pivot

```
QuickSort(X,i,j)
  If i<j Then p=Partition(X,i,j)
      QuickSort(X,i,p)
      QuickSort(X,p+1,j)
```





- $\Theta(n \log n)$ time average-case
- $\Theta(n^2)$ worst-case time (rare)
- Unstable, parallelizes, $O(\log n)$ space
- Ave: only beats $\Theta(n^2)$ sorts for n>40

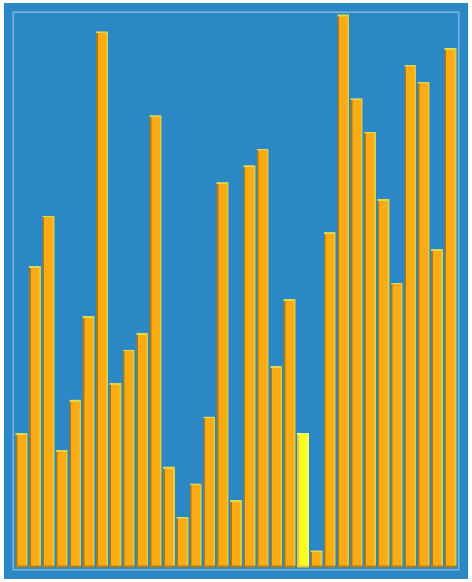
Shell Sort


Donald Shell

Input: array X[1..n] of integers

Output: sorted array (monotonic)

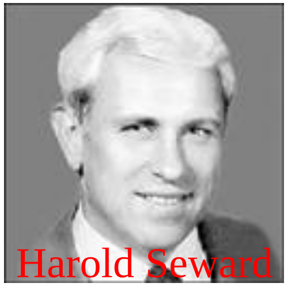
Idea: generalize insertion sort



> for each h_i in sequence $h_k,\ldots,h_1=1$
> Insertion-sort all items h_i apart

- Array is sorted after last pass ($h_i=1$)
- Long swaps quickly reduce disorder
- $O(n^2)$, $O(n^{3/2})$, $O(n^{4/3})$, … ?
- Complexity still open problem!
- LB is $\Omega(N(\log/\log\log n)^2)$
- Not stable, adaptive, in-situ

Counting Sort

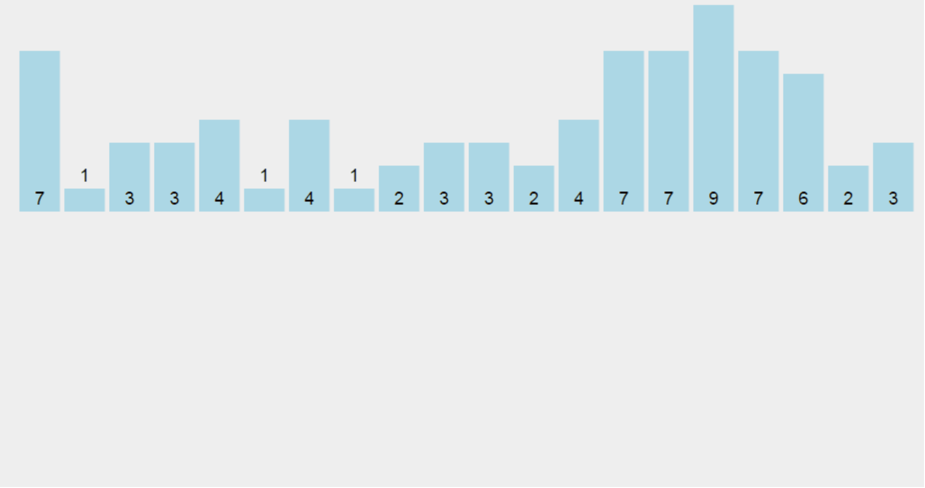


Input: array X[1..n] of integers in small range 1..k

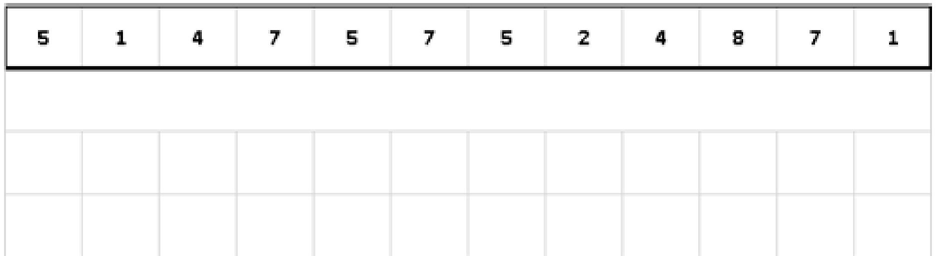
Output: sorted array (monotonic)

Idea: use values as array indices

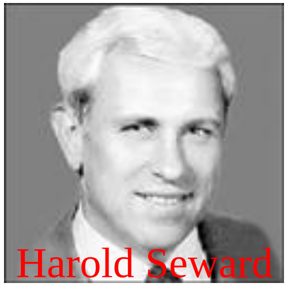
```
for i=1 to k do C[i] = 0
for i=1 to n do C[X[i]]++
for i=1 to k do if C[i] ≠ 0
   then print(i) C[i] times
```



- $\Theta(n)$ time, $\Theta(k)$ space
- Not comparison-based
- For specialized data only
- Stable, parallel, not in-situ



Counting Sort

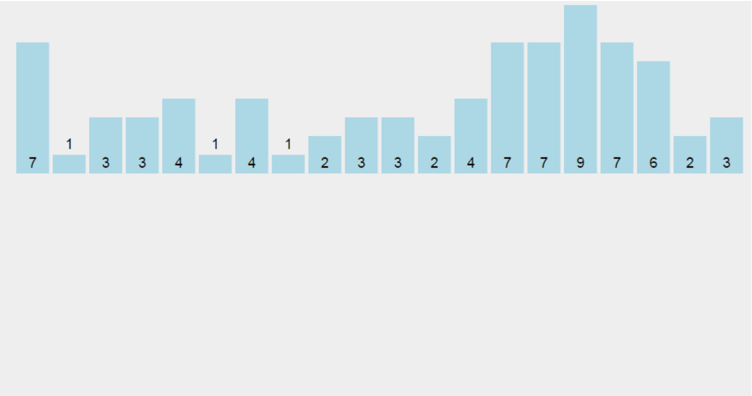

Harold Seward

Q: Why not use counting sort for arbitrary 32-bit integers? (i.e., range k is "fixed")

A: Range is fixed (2^{32}) but very large (4,294,967,296).
Space/time: the counts array will be huge (4 GB)

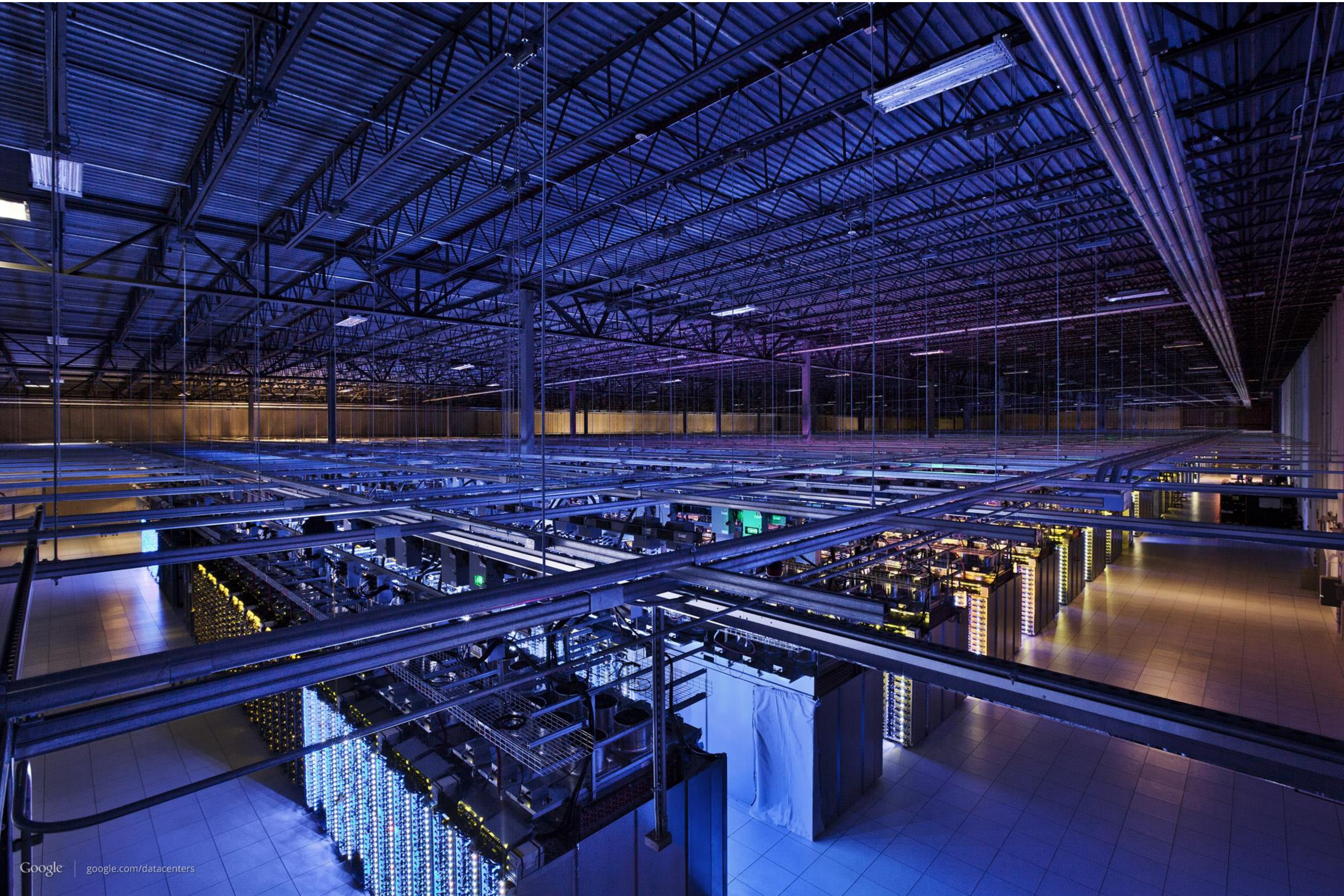
Much worse for 64-bit integers ($2^{64} > 10^{19}$):
Time: 5 GHz PC will take over $2^{64} / (5 \cdot 10^9) / (60 \cdot 60 \cdot 24 \cdot 365)$ sec >116 years to initialize array!

Memory: $2^{64} > 10^{19} >$ 18 Exabytes
> 2.3 million TB RAM chips!
> total amount of Google's data!

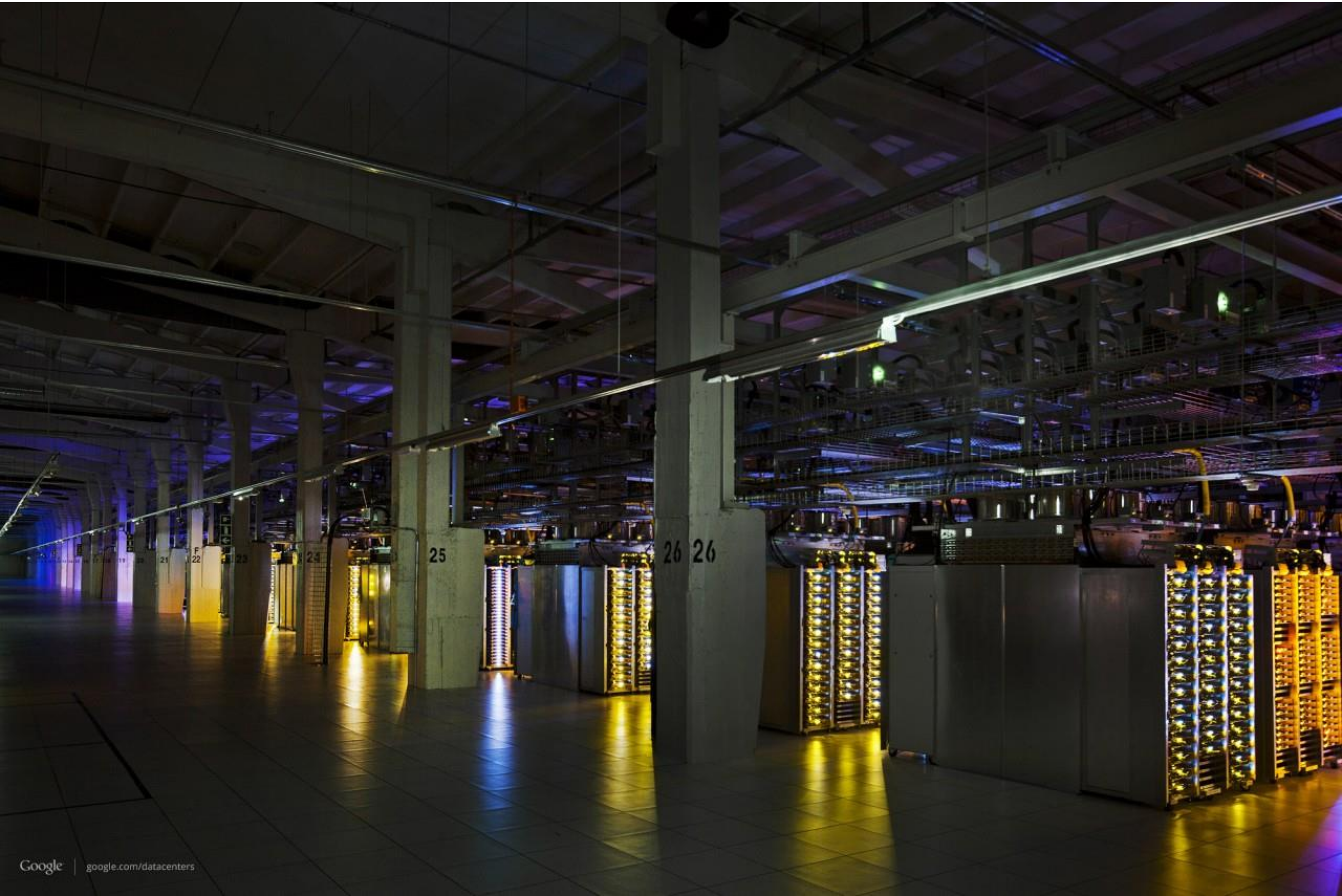


Q: What's an Exabyte? (10^{18})

What does an Exabyte look like?

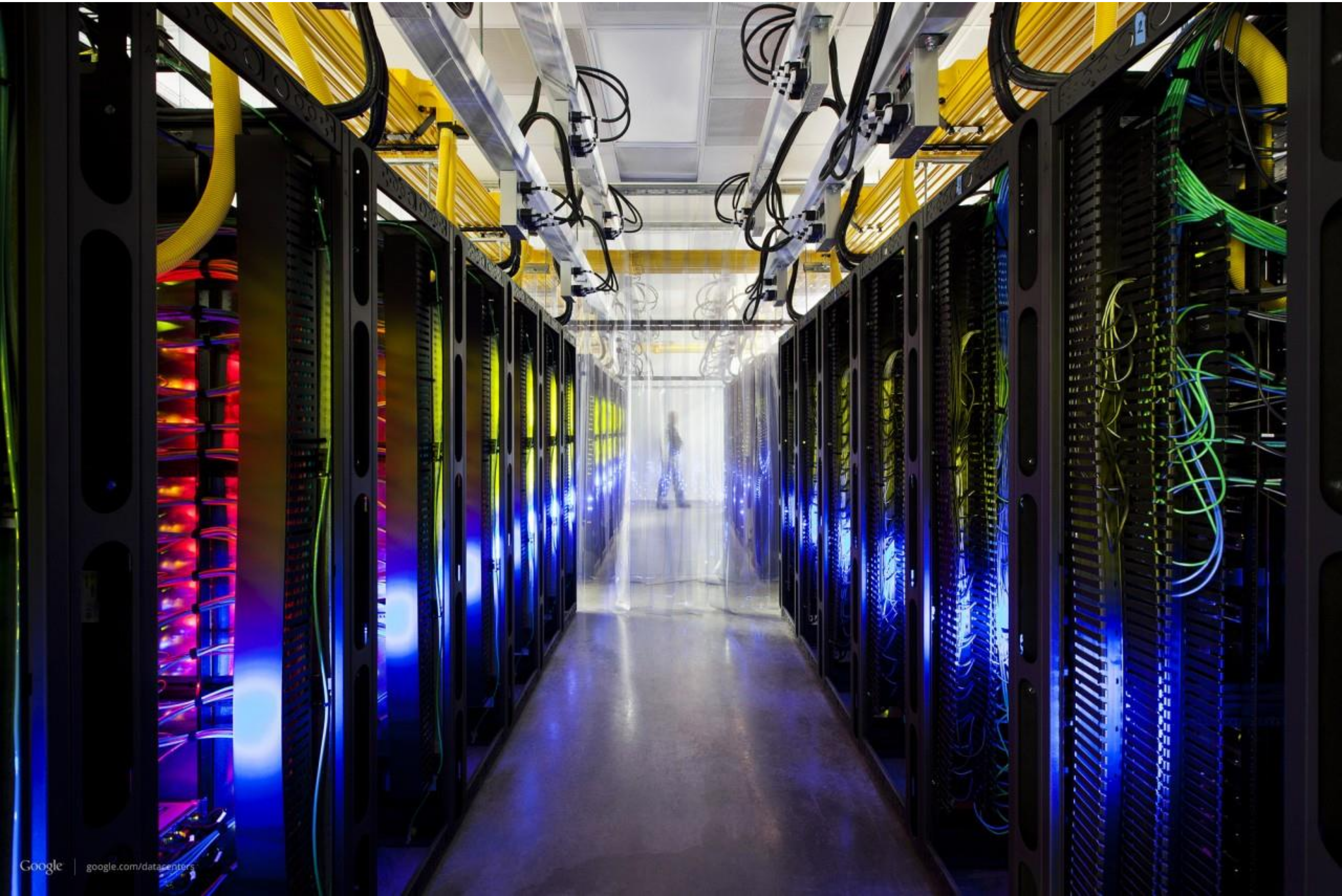


What does an Exabyte look like?

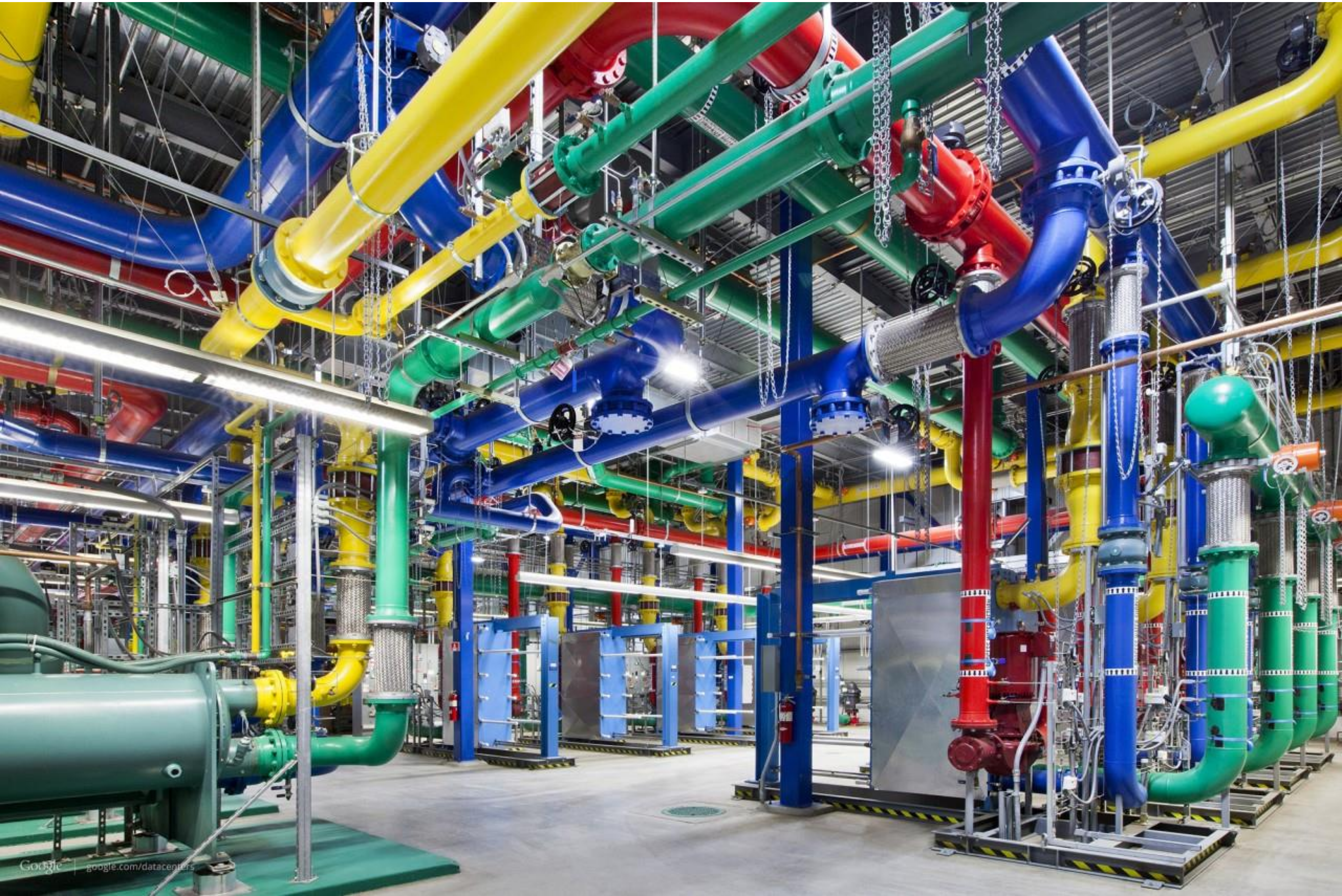


What does an Exabyte look like?

What does an Exabyte look like?



What does an Exabyte look like?

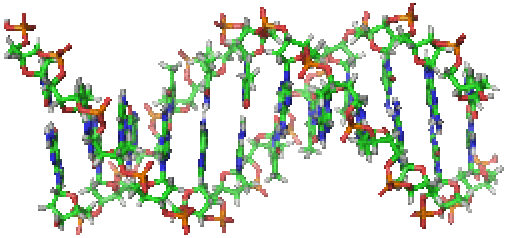


What does an Exabyte look like?

What does an Exabyte look like?

- All content of Library of Congress: ~ 0.001 Exabytes
- Total words ever spoken by humans: ~ 5 Exabytes
- Total data stored by Google: ~ 15 Exabytes
- Total monthly world internet traffic: ~ 110 Exabytes

- Storage capacity of 1 gram of DNA: ~ 455 Exabytes



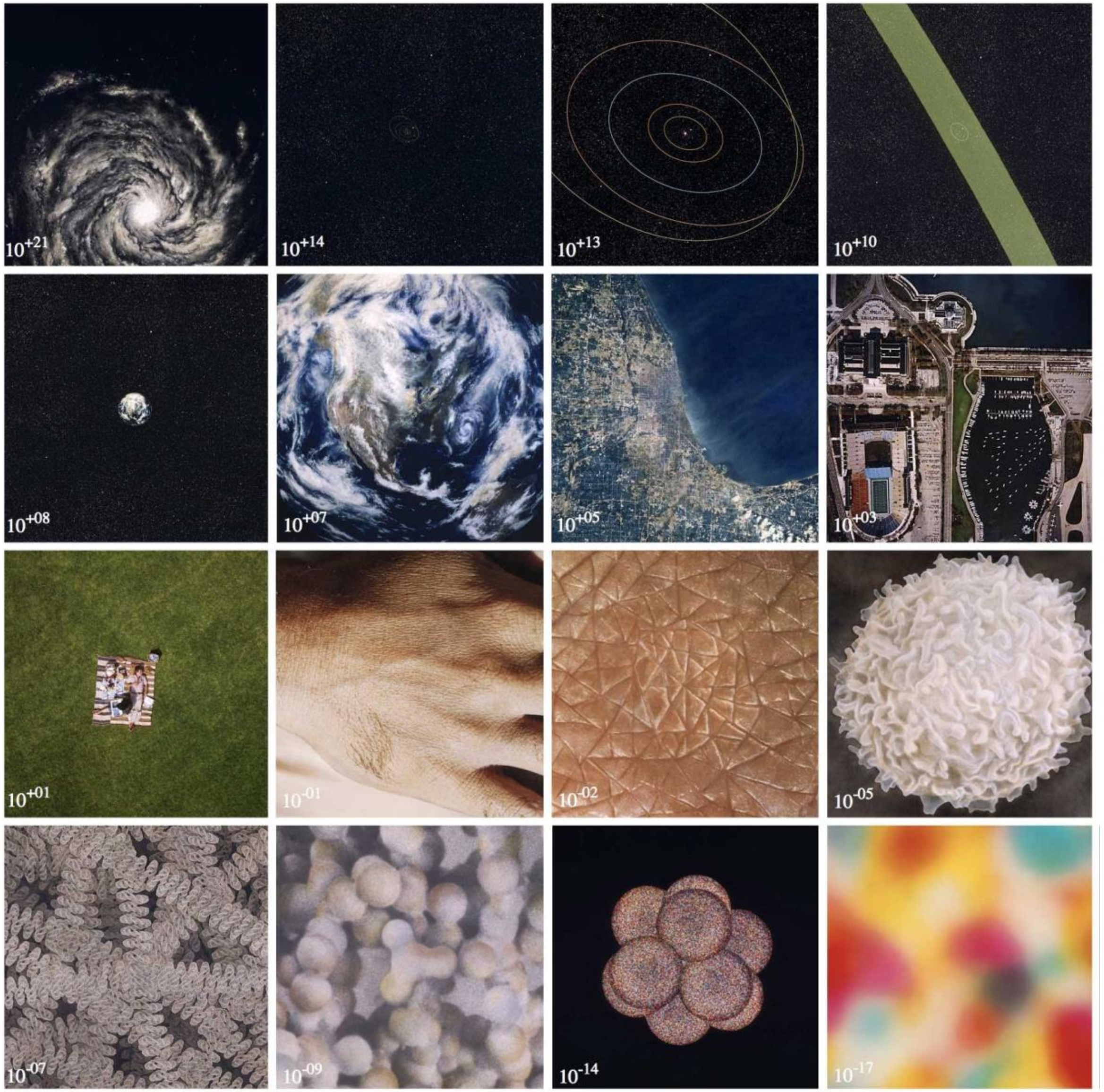
Orders-of-Magnitude

Standard International (SI) quantities:

Deca	10^{1}	Deci	10^{-1}
Hecto	10^{2}	Centi	10^{-2}
Kilo	10^{3}	Milli	10^{-3}
Mega	10^{6}	Micro	10^{-6}
Giga	10^{9}	Nano	10^{-9}
Tera	10^{12}	Pico	10^{-12}
Peta	10^{15}	Femto	10^{-15}
Exa	10^{18}	Atto	10^{-18}
Zetta	10^{21}	Zepto	10^{-21}
Yotta	10^{24}	Yocto	10^{-24}

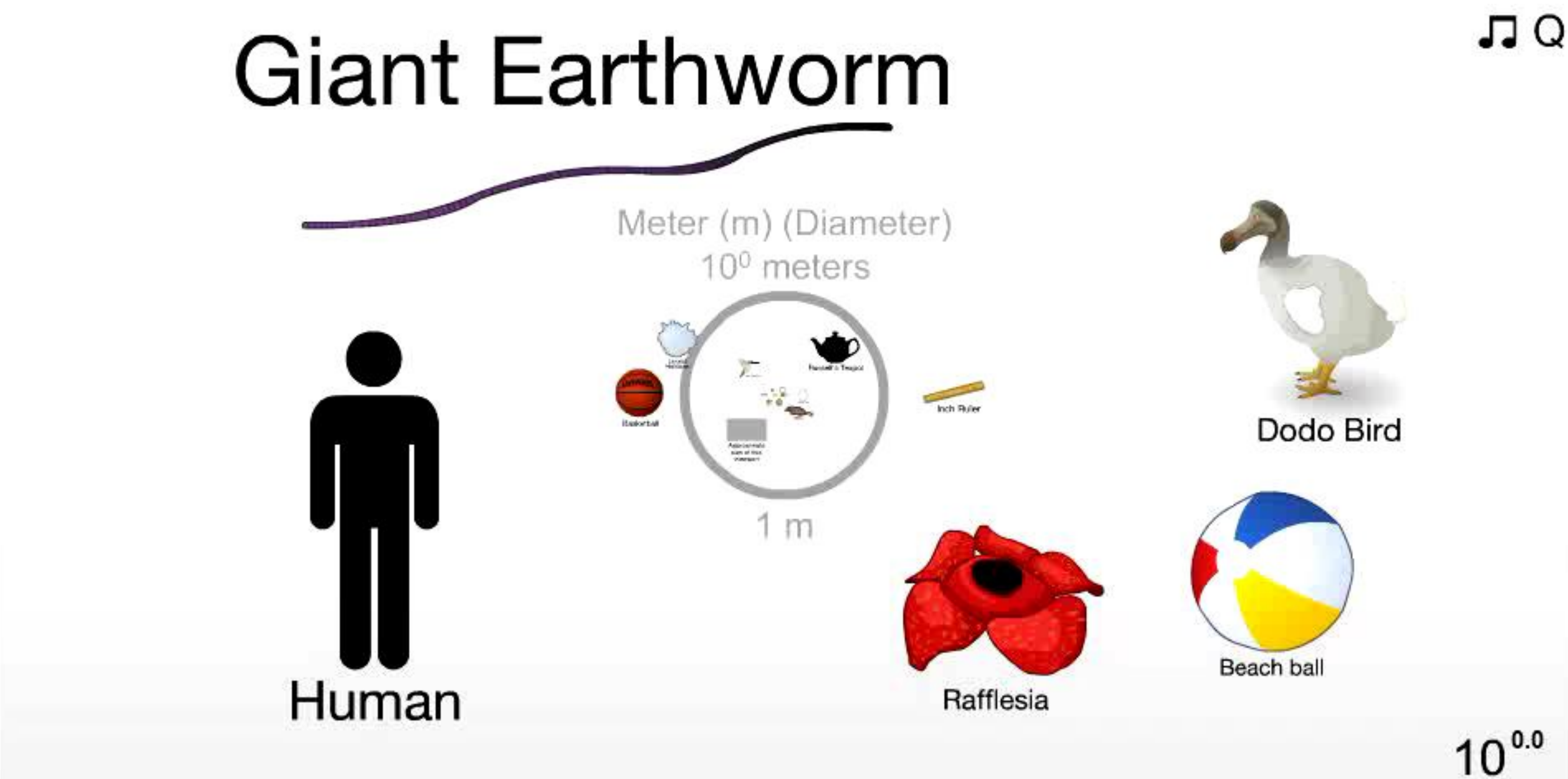
Orders-of-Magnitude

- "Powers of Ten", Charles and Ray Eames, 1977



Orders-of-Magnitude

- "Scale of the Universe", Cary and Michael Huang, 2012



- 10^{-24} to 10^{26} meters $\Rightarrow$ 50 orders of magnitude!

Bucket Sort

Input: array X[1..n] of real numbers in [0,1]

Output: sorted array (monotonic)

Idea: spread data among buckets

> for i=1 to n do
>   insert X[i] into bucket $\lfloor n \cdot X[i] \rfloor$
>   for i=1 to n do Sort bucket i
>   concatenate all the buckets

- O(n+k) time expected, O(n) space
- O(Sort) time worst-case
- Assumes subtantial data uniformity
- Stable, parallel, not in-situ
- Generalizes counting sort / quicksort

Bucket Sort

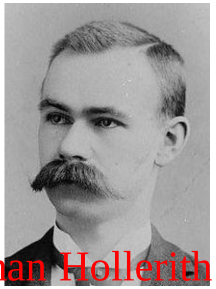
946	847	123	837	176	588	467	689	763	337	347	130	529	878	868	92	882	305	906	749	871	5	552	596	86	216	561	994	388	219
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

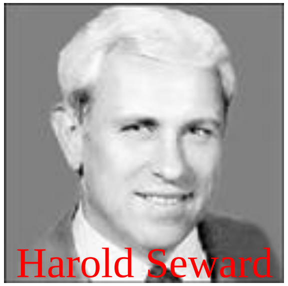
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

Animation Completed

Q: How does bucket sort generalize counting sort? Quicksort?

Radix Sort


Herman Hollerith


Harold Seward

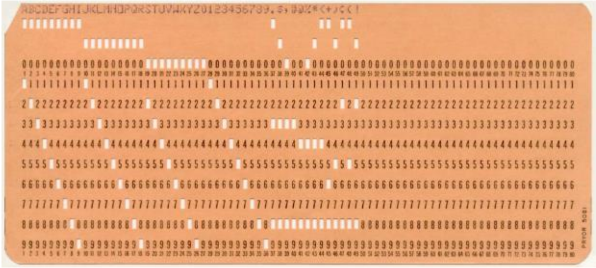
Input: array X[1..n] of integers
each with d digits in range 1..k

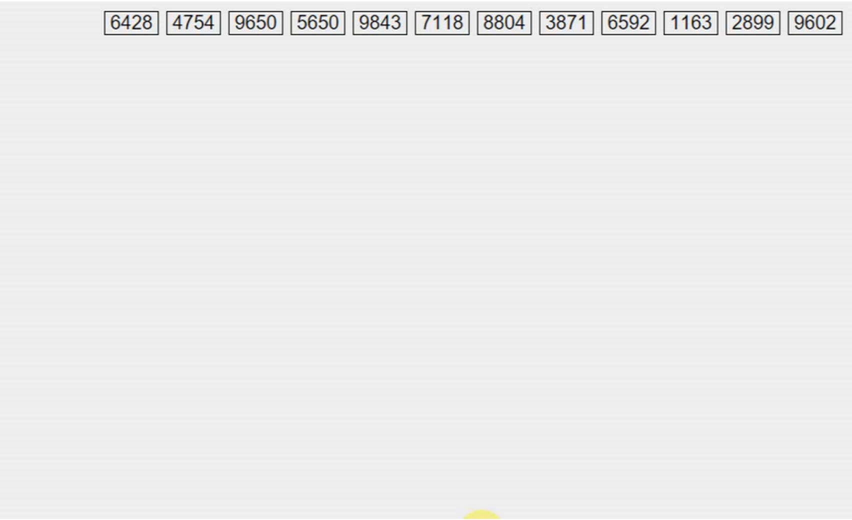
Output: sorted array (monotonic)

Idea: sort each digit in turn

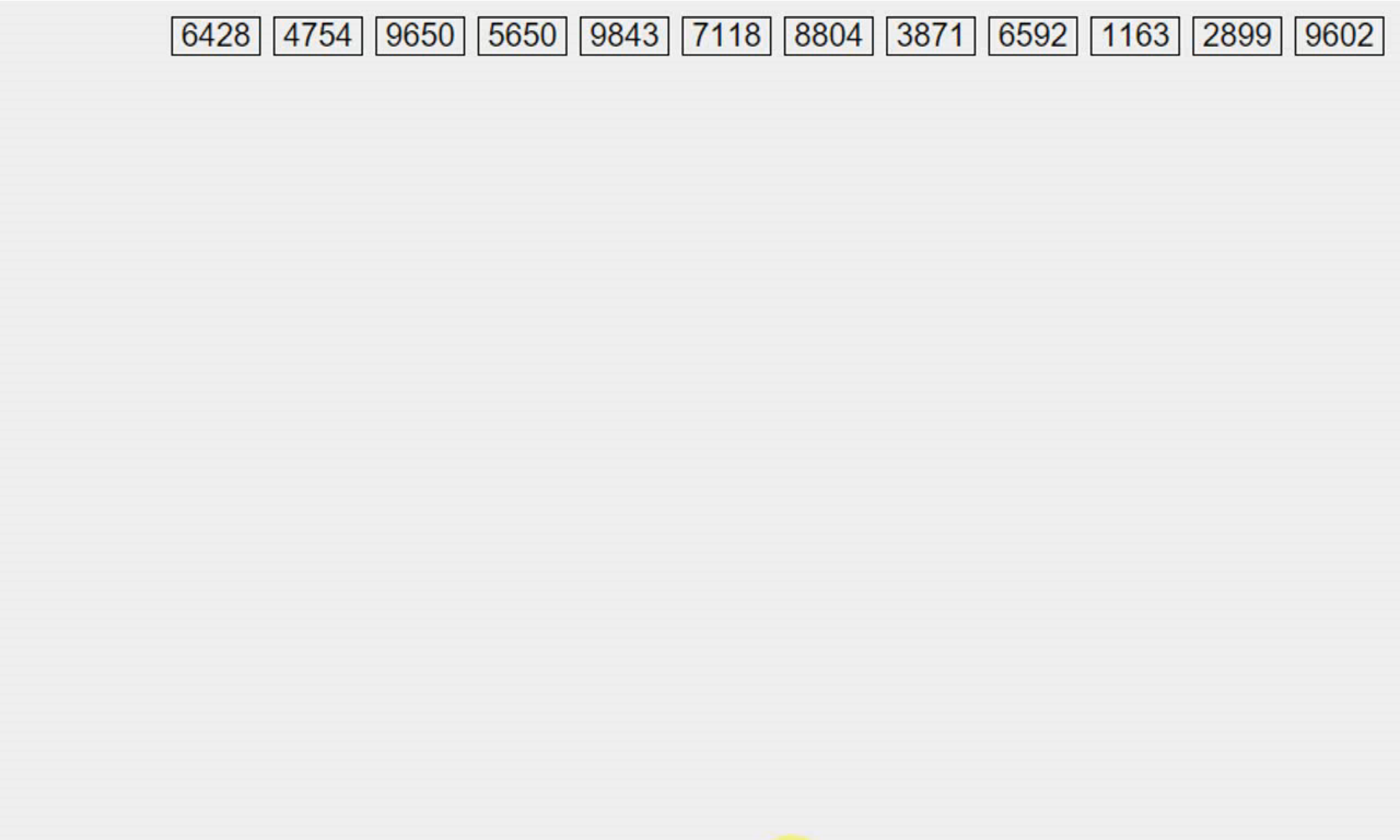
```
For i=1 to d do
  StableSort(X on digit i)
```

- Makes d calls to bucket sort
- Θ(d·n) time, Θ(k+n) space
- Not comparison-based
- Stable
- Parallel
- Not in-situ





Radix Sort



Q: is Radix Sort faster than Merge Sort? $\Theta(d \cdot n)$ vs. $\Theta(n \log n)$

Sorting Comparison

- O(n log n) sorts tend to beat the O(n^2) sorts (n>50)
- Some sorts work faster on random data vs. near-sorted data
- For more details see http://www.sorting-algorithms.com

Meta Sort

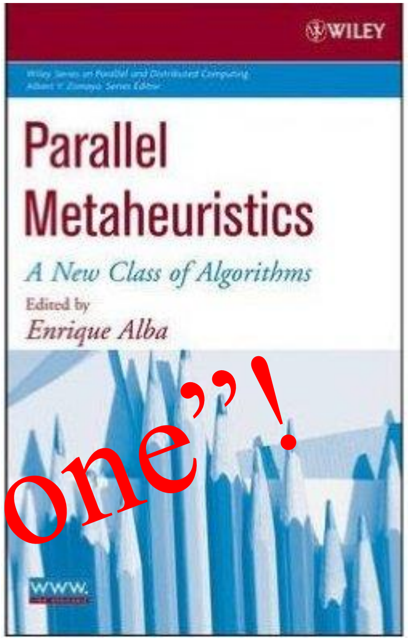
Q: how can we easily modify quicksort to have O(n log n) worst-case time?

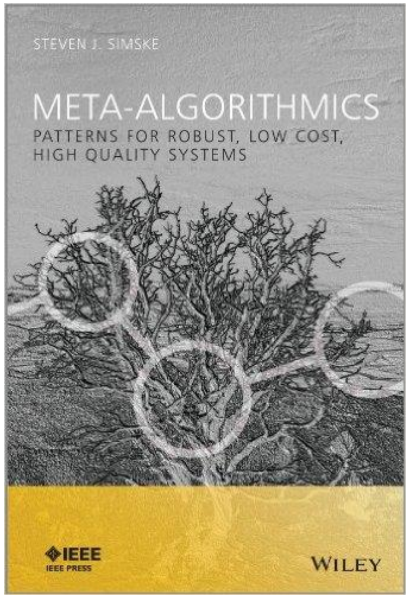
Idea: combine two algorithms to leverage the best behaviors of each one.

```
MetaSort(X,i,j):
    parallel-run:
        • QuickSort(X,i,j)
        • MergeSort(X,i,j)
    when either stops, abort the other
```

“Two heads are better than one”!

- Ave-case time is Min of both: O(n log n)
- Worst-case time is Min of both: O(n log n)
- Meta-algorithms / meta-heuristics generalize!

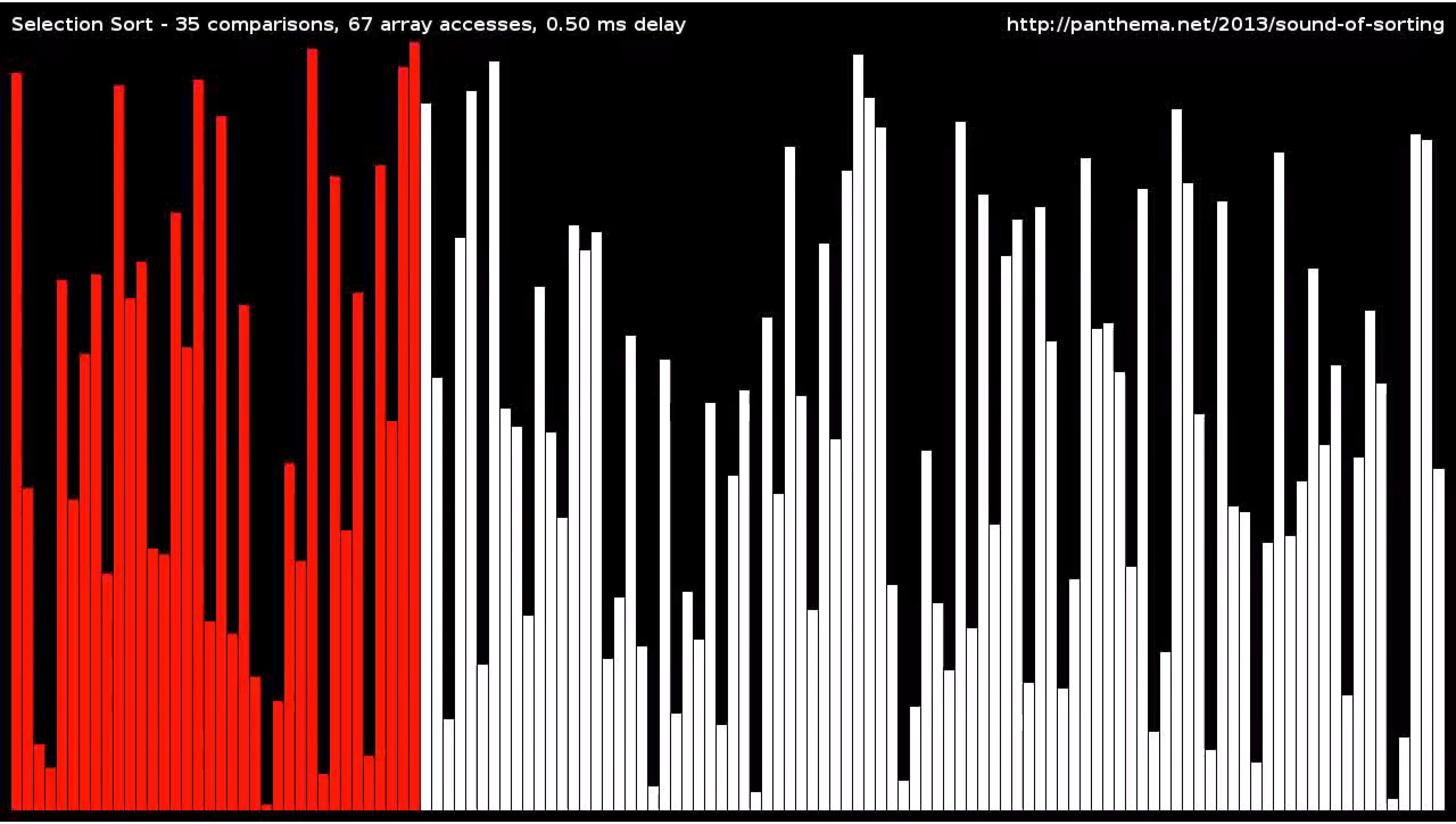





Metaheuristics
Progress in Complex Systems Optimization
Karl F. Doerner · Michel Gendreau · Peter Greistorfer
Walter J. Gutjahr · Richard F. Hartl · Marc Reimann
Editors
Springer
Nature-Inspired Metaheuristic Algorithms
Xin-She Yang
Luniver Press
Metaheuristics to solve some variants of vehicle routing problems
Metaheuristics algorithms for the optimization of some variants of logistics and transport problems
LAMBERT Academic Publishing
ESSAYS AND SURVEYS IN METAHEURISTICS
Edited by
CELSO C. RIBEIRO
PIERRE HANSEN
KLUWER ACADEMIC PUBLISHERS
APPLIED OPTIMIZATION
Edited by
Mauricio G. C. Resende
Jorge Pinho de Sousa
With assistance from Ana Viana
METAHEURISTICS
COMPUTER DECISION-MAKING
Springer Science+Business Media, LLC
AUTOMATION – CONTROL AND INDUSTRIAL ENGINEERING SERIES
Metaheuristics for Production Scheduling
Edited by Bassem Jarboui
Patrick Siarry and Jacques Teghem
ISTE
WILEY
Current Approaches in Metaheuristics
Amelia Epps
DISKI 331
Heuristics and Metaheuristics for heavily constrained hybrid Flowshop Problems
Thijs Urlings
IOS Press
AKA
Günther Zäpfel
Roland Braune
Michael Bögl
Metaheuristic Search Concepts
A Tutorial with Applications to Production and Logistics
Springer
Studies in Computational Intelligence 482
El-Ghazali Talbi (Ed.)
Metaheuristics for Bi-level Optimization
Springer
PREMIER REFERENCE SOURCE
Trends in Developing Metaheuristics, Algorithms, and Optimization Approaches
Peng-Yeng Yin
Richard Malek
A Framework for Alternation of Meta-heuristic Algorithms
Alternation of Meta-heuristic Algorithms with Runtime Analysis and Parameter Adaptation
LAMBERT Academic Publishing
META-HEURISTICS:
Theory & Applications
Edited by
Ibrahim H. Osman
James P. Kelly
KLUWER ACADEMIC PUBLISHERS
CHAPMAN & HALL/CRC COMPUTER and INFORMATION SCIENCE SERIES
Handbook of Approximation Algorithms and Metaheuristics
Edited by
Teofilo F. Gonzalez
Chapman & Hall/CRC
Marcus Randall
General Meta-heuristic Search Algorithms
Generalised Modelling of Combinatorial Problems
VDM
METAHEURISTICS
From Design to Implementation
EL-GHAZALI TALBI
WILEY
Natural Computing Series
Yossi Borenstein
Alberto Moraglio Editors
Theory and Principled Methods for the Design of Metaheuristics
Springer
Daniel Wagner
Generic Metaheuristics
A Presentation of a Generic Library for Metaheuristics and its Application to the Quadratic Assignment Problem
VDM


A Tabu Search Metaheuristic for the Air Refueling Tanker Assignment Problem
BIBLIOSCHOLAR DISSERTATIONS
Shay R. Capehart
META-HEURISTICS
Advances and Trends in Local Search Paradigms for Optimization
Edited by
Stefan Voss
Silvano Martello
Ibrahim H. Osman
Catherine Roucairol
INRIA
Springer Science+Business Media, LLC
Nature-Inspired Metaheuristic Algorithms
Second Edition
Xin-She Yang
Luniver Press
INTERNATIONAL SERIES IN OPERATIONS RESEARCH AND MANAGEMENT SCIENCE
Handbook of Metaheuristics
Second Edition
Edited by
Michel Gendreau
Jean-Yves Potvin
Springer
Operations Research/Computer Science Interfaces Series
El-Ghazali Talbi
Farouk Yalaoui
Lionel Amodeo Editors
Metaheuristics for Production Systems
Springer
ELSEVIER INSIGHTS
METAHEURISTICS IN WATER, GEOTECHNICAL AND TRANSPORT ENGINEERING
Edited by
XIN-SHE YANG • AMIR HOSSEIN GANDOMI
SIAMAK TALATAHARI • AMIR HOSSEIN ALAVI
Metaheuristics for Vehicle Routing Problems
Nacima Labadie, Christian Prins and Caroline Prodhon
ISTE
WILEY
Thomas Bucsics
Metaheuristic Approaches for Designing Survivable Fiber-Optic Networks
A Presentation of Various Heuristic, Meta-Heuristic and Hybrid Approaches and Experimental Results
Studies in Computational Intelligence 114
Blum · Aguilera
Roli · Sampels
Hybrid Metaheuristics
An Emerging Approach to Optimization
Springer
Matheuristics
Hybridizing Metaheuristics and Mathematical Programming
Volume 10
Vittorio Maniezzo
Thomas Stützle
Stefan Voß
Annals of Information Systems
SPRINGER
PREMIER REFERENCE SOURCE
Meta-Heuristics Optimization Algorithms in Engineering, Business, Economics, and Finance
Pandian Vasant
A. Kaveh
Advances in Metaheuristic Algorithms for Optimal Design of Structures
Springer
Yaohua HE and Chi-Wai HUI
Meta-Heuristics for Large-Scale Process Scheduling
Francisco Almeida María J. Blesa Aguilera
Christian Blum José Marcos Moreno Vega
Melquíades Pérez Pérez Andrea Roli
Michael Sampels (Eds.)
LNCS 4030
Hybrid Metaheuristics
Third International Workshop, HM 2006
Gran Canaria, Spain, October 2006
Proceedings
Springer
Sean Luke
Essentials of Metaheuristics
Second Edition
Jalel Euchi
Metaheuristics to solve some variants of vehicle routing problems
Metaheuristics algorithms for the optimization of some variants of logistics and transport problems
LAMBERT Academic Publishing
SURYA PRAKASH SINGH
Facility Layout Planning and Design
Heuristic and Meta-heuristic Approaches For Static and Dynamic Layout
LAMBERT Academic Publishing
Manar Hosny
Vehicle Routing with Pickup and Delivery
Heuristic and Meta-heuristic Solution Algorithms
LAMBERT Academic Publishing

“The Sound of Sorting” (15 algorithms)



- Sound pitch is proportional to value of current sort element sorted!

https://www.youtube.com/watch?v=kPRA0W1kECg

Problem: Given n pairs of integers (x_i, y_i), where $0 \le x_i \le n$ and $1 \le y_i \le n$ for $1 \le i \le n$, find an algorithm that sorts all n ratios x_i / y_i in linear time $O(n)$.

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations

Problem: Given n integers, find in $O(n)$ time the majority element (i.e., occurring $\geq n/2$ times, if any).

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations

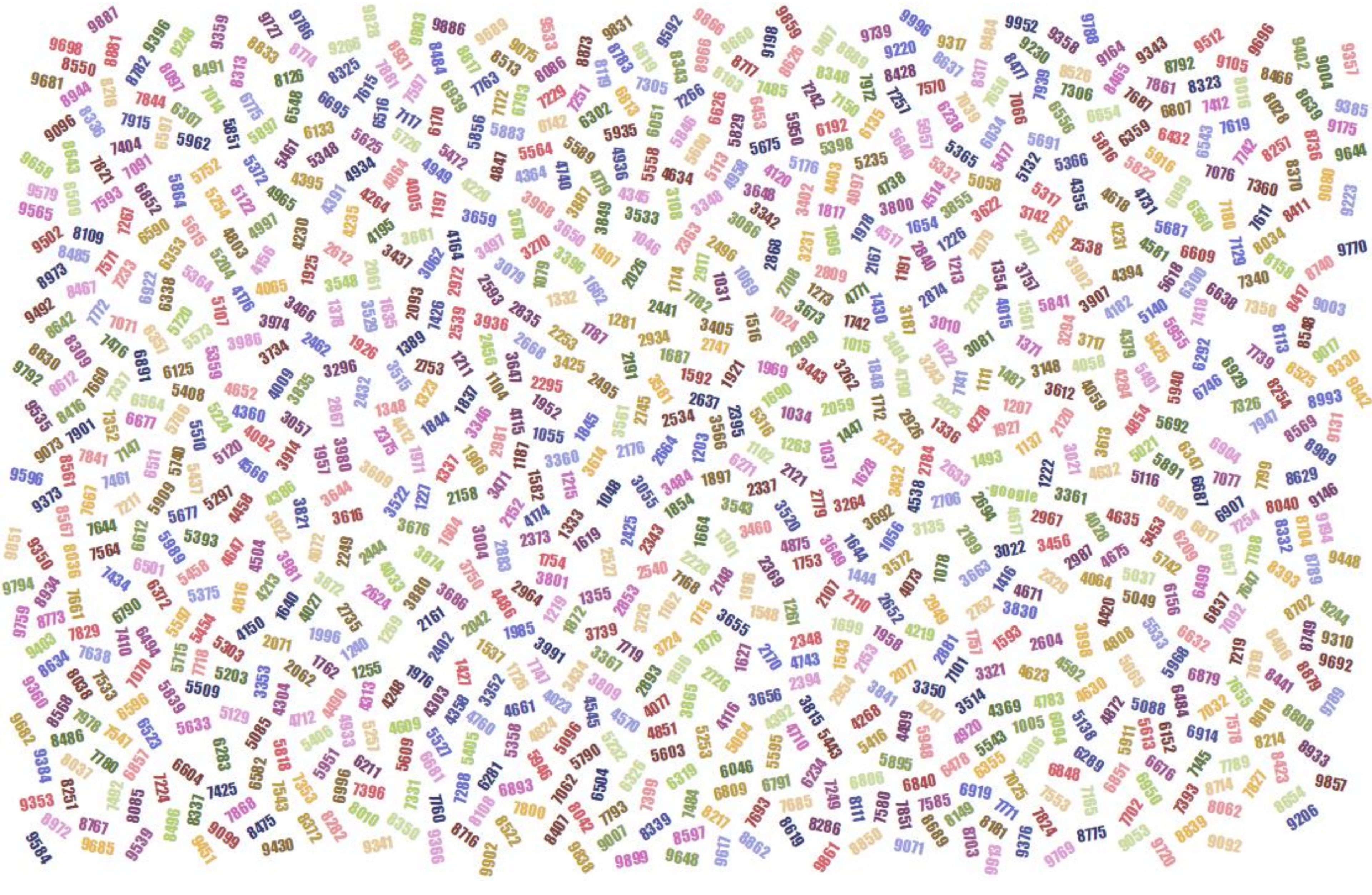
Problem: Given n objects, find in $O(n)$ time the majority element (i.e., occurring $\geq n/2$ times, if any), using only equality comparisons ($=$).

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations

Problem: Given n integers, find both the maximum and the next-to-maximum using the least number of comparisons (exact comparison count, not just O(n)).

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations

Finding the Minimum



Finding the Minimum

Input: array X[1..n] of integers

Output: minimum element

Theorem: $\Omega(n)$ time is necessary to find Min.

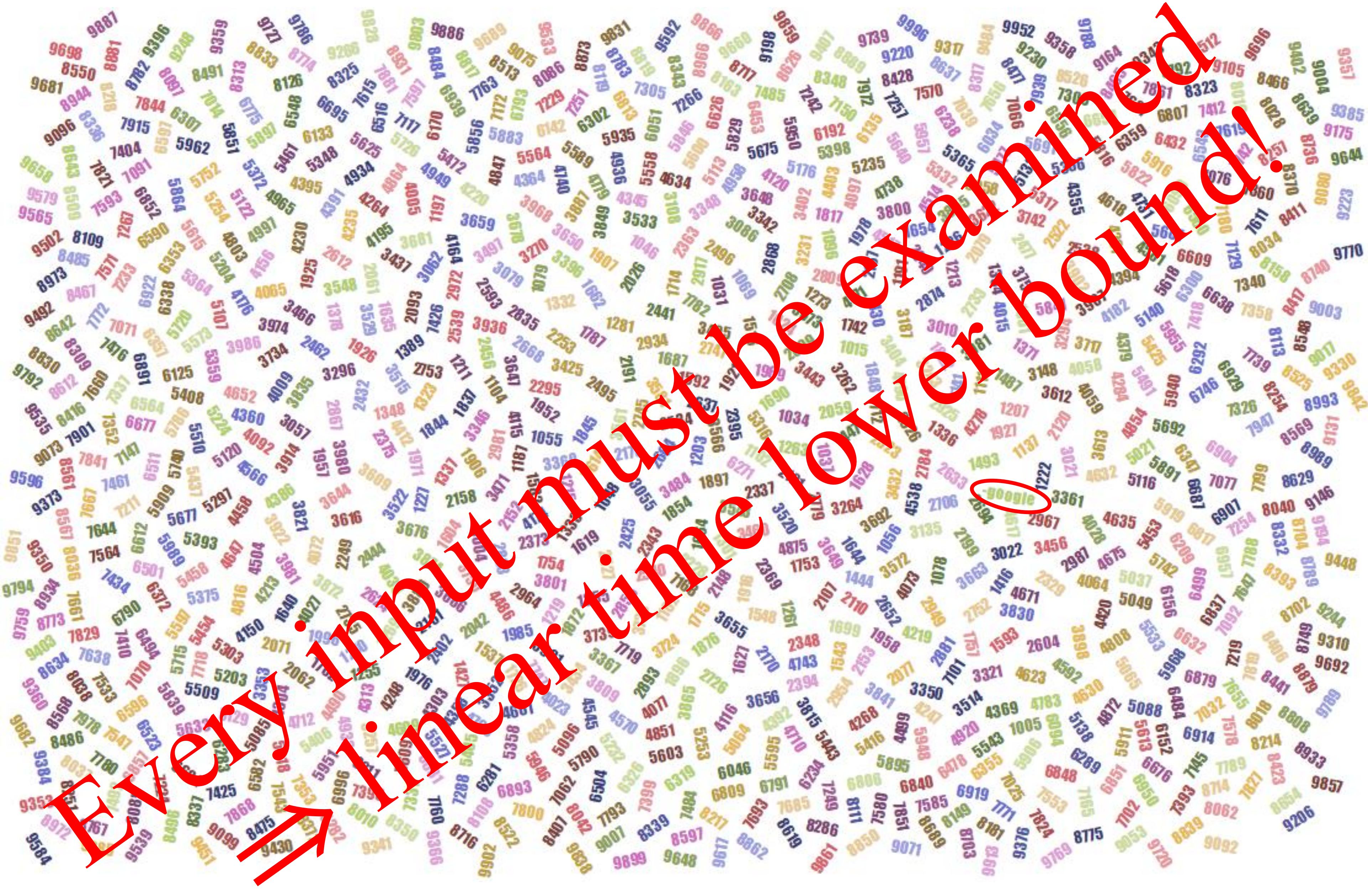
Proof 1: each element must be examined at least once, otherwise we may miss the true minimum. Therefore $\Omega(n)$ work is required.

Proof 2: Assume a correct min-finding algorithm didn't examine element X_i for some array X. Then the same algorithm will be wrong on X with X_i replaced with say -10^{100}.

Non-existence argument!

Proof by contradiction

Finding the Minimum



Every input must be examined
⇒ linear time lower bound!

Finding the Minimum

Input: array X[1..n] of integers

Output: minimum element

Idea: keep track of the best-so-far

```
Min = X[1]
   for i = 2 to n
        if X[i] < min then min = X[i]
```

- Exact comparison count: $n-1$

Theorem: $n-1$ comparisons are sufficient for finding the minimum.

Corollary: This $\Theta(n)$-time algorithm is optimal.

Q: What about finding the maximum?

Finding the Minimum

Q: Can we do better than n-1 comparisons?

Theorem: n/2 comparisons are necessary
for finding the minimum.

Idea: must examine all n inputs!

Proof: each element must participate in at least 1 comparison (otherwise we may miss e.g. -10^{100}).

- Each comparison involves 2 elements
- At least n/2 comparisons are necessary

Non-existence proof!

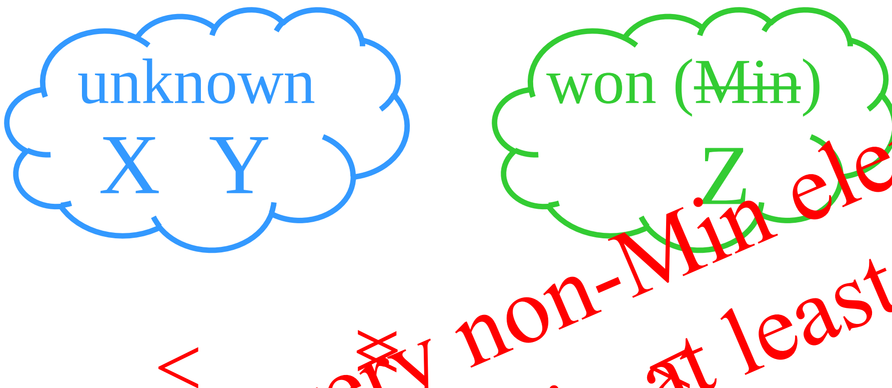
Q: Can we improve lower bound up to n-1?

Finding the Minimum

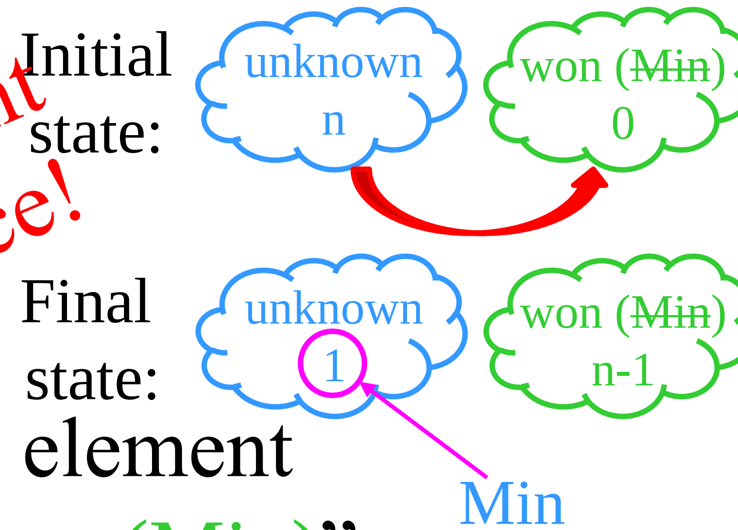
Theorem: n-1 comparisons are necessary
for finding the minimum (or maximum).

Idea: keep track of "knowledge" gained!

Proof: consider two classes of elements:



Every non-Min element must win at least once!



- At each comparison, at most 1 element moves from "unknown" to "won (~~Min~~)".
- At least n-1 moves / comparisons are necessary to convert the initial state into the final state

Corollary: The (n-1)-comparison algorithm is optimal.

Finding the Min and Max

Input: array X[1..n] of integers

Output: minimum and maximum elements

Idea: find Min independently from Max

FindMin(X)

FindMax(X) ≡ FindMin(-X)

- n-1 comparisons to find Min
- n-1 comparisons to find Max
- Total 2n-2 comparisons needed

Observation: much information is discarded!

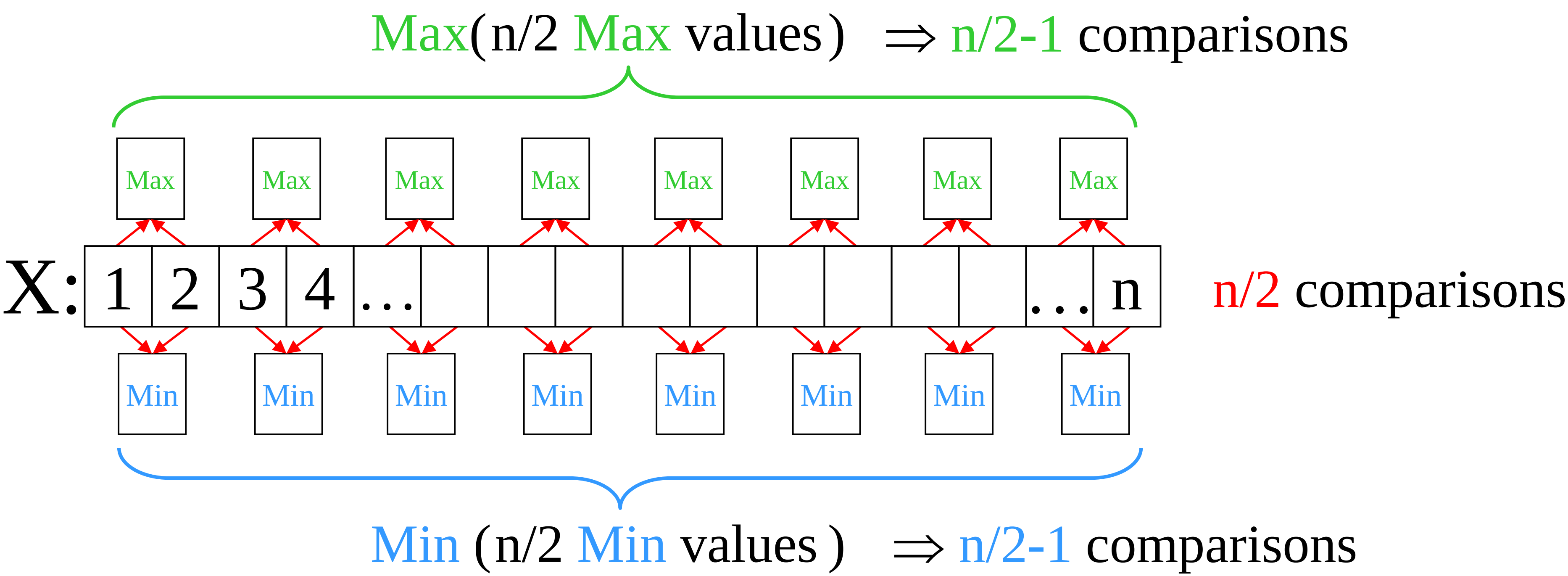
Q: Can we do better than 2n-2 comparisons?

Finding the Min and Max

Input: array X[1..n] of integers

Output: minimum and maximum elements

Idea: pairwise compare to reduce work



Theorem: 3n/2-2 comparisons are sufficient for finding the minimum and maximum.

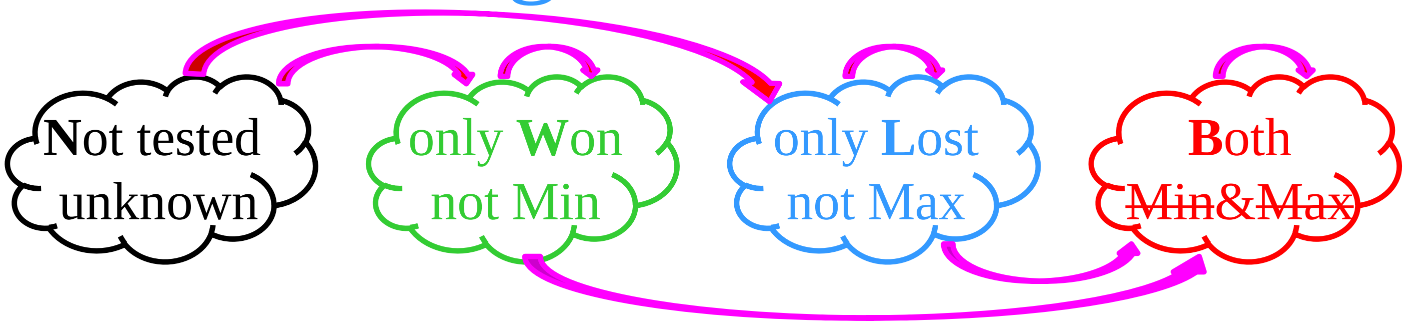
Finding the Min and Max

Theorem: $3n/2-2$ comparisons are necessary for finding the minimum and maximum.

Idea: keep track of "knowledge" gained!

Proof: consider four classes of elements:

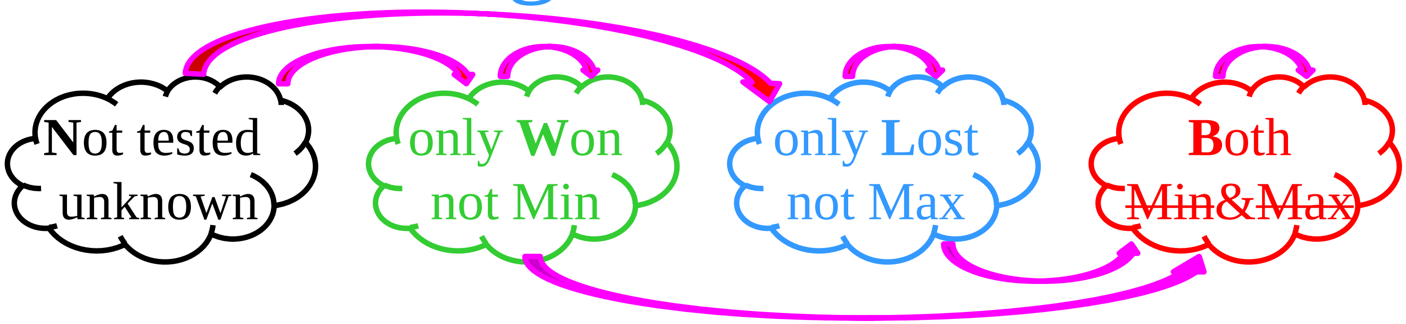
Finding the Min and Max



N < N ⇒ L & W	N > N ⇒ W & L	2
N < W ⇒ L & W	N > W ⇒ W & B	1
N < L ⇒ L & B	N > L ⇒ W & L	1
N < B ⇒ L & B	N > B ⇒ W & B	1
W < W ⇒ B & W	W > W ⇒ W & B	1
W < L ⇒ B & B	W > L ⇒ W & L	0
W < B ⇒ B & B	W > B ⇒ W & B	0
L < L ⇒ L & B	L > L ⇒ B & L	1
L < B ⇒ L & B	L > B ⇒ B & B	0
B < B ⇒ B & B	B > B ⇒ B & B	0

Minimum guaranteed knowledge gained i.e. “moves” towards final state

Finding the Min and Max



- Moving from N to B forces passing through W or L
- Emptying N into W & L takes n/2 comparisons
- Emptying most of W takes n/2-1 comparisons
- Emptying most of L takes n/2-1 comparisons
- Other moves will not reach the "final state" any faster
- Total comparisons required: 3n/2-2

$\Rightarrow$ 3n/2-2 comparisons are necessary for finding the minimum and maximum.

Theorem: Our Min&Max algorithm is optimal.

Non-existence proof !

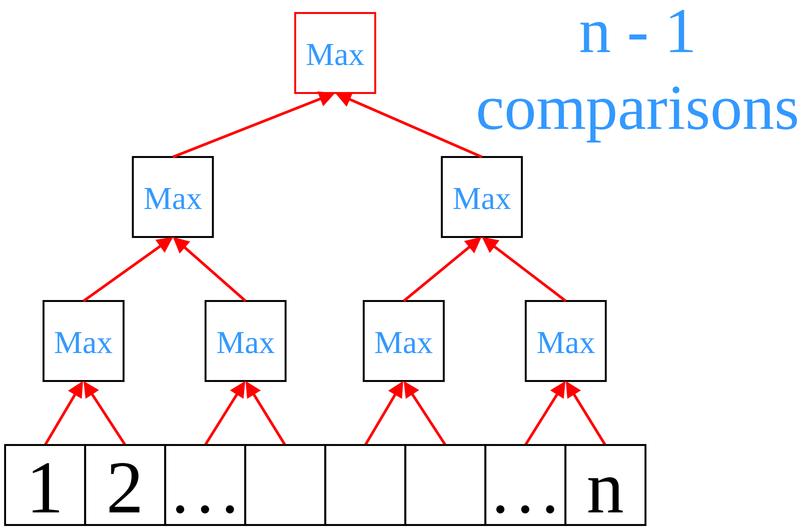
Problem: Given n integers, find both the maximum and the next-to-maximum using the least number of comparisons (exact comparison count, not just $O(n)$).

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations

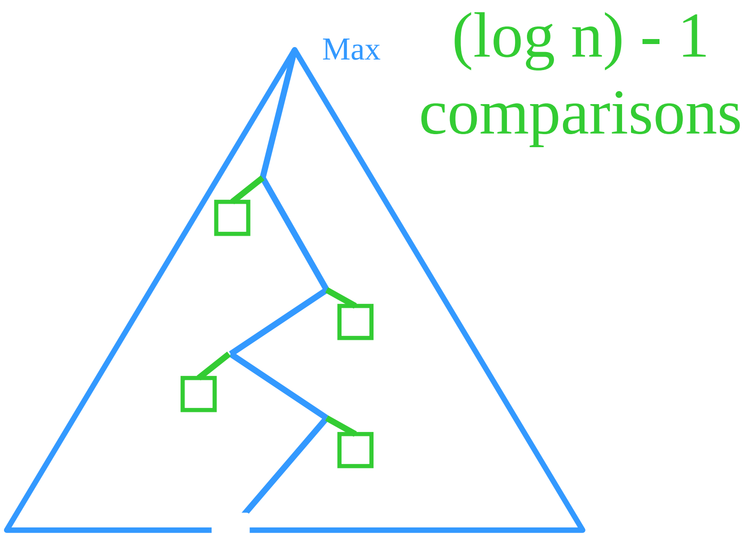
Finding the Max and Next-to-Max

Theorem: (n-2) + log n comparisons are sufficient for finding the maximum and next-to-maximum.

Proof: consider elimination tournament:



maximum



next-to-maximum

Theorem: (n-2) + log n comparisons are necessary for finding the maximum and next-to-maximum.

Selection (Order Statistics)

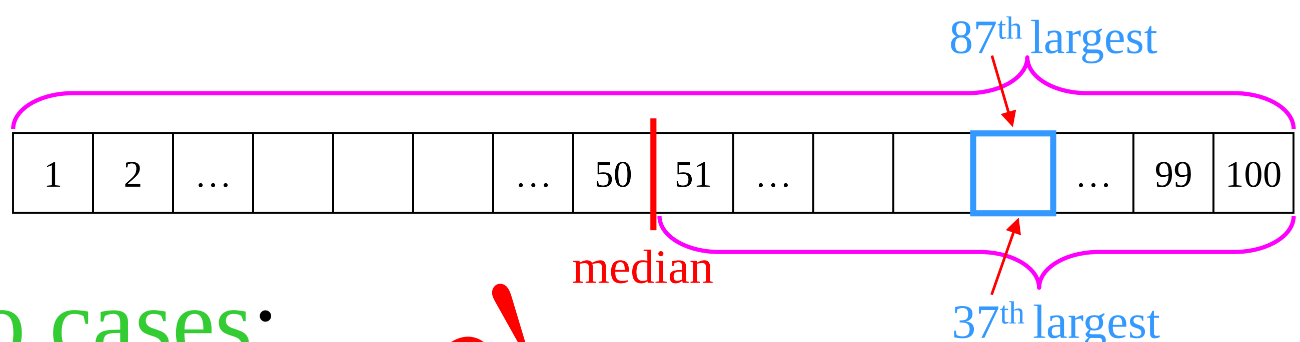
Input: array X[1..n] of integers and i

Output: i^{th} largest integer

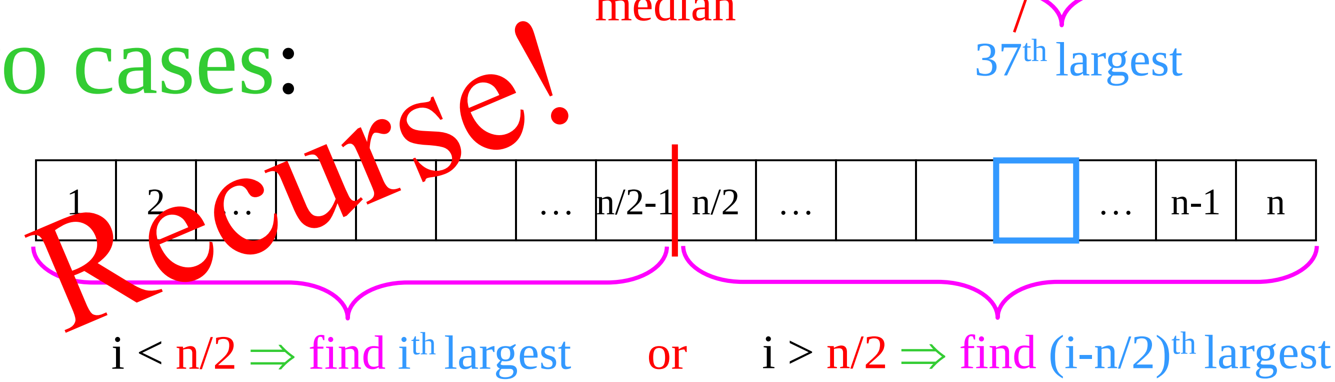
Obvious: i^{th}-largest subroutine can find median

since median is the special case $(n/2)^{th}$-largest

Not obvious: repeat medians can find i^{th} largest:



Two cases:



$i < n/2 \Rightarrow$ find i^{th} largest or $i > n/2 \Rightarrow$ find $(i-n/2)^{th}$ largest

Selection (Order Statistics)

Run time for i^{th} largest: $T(n) = T(n/2) + M(n)$

where $M(n)$ is time to find median

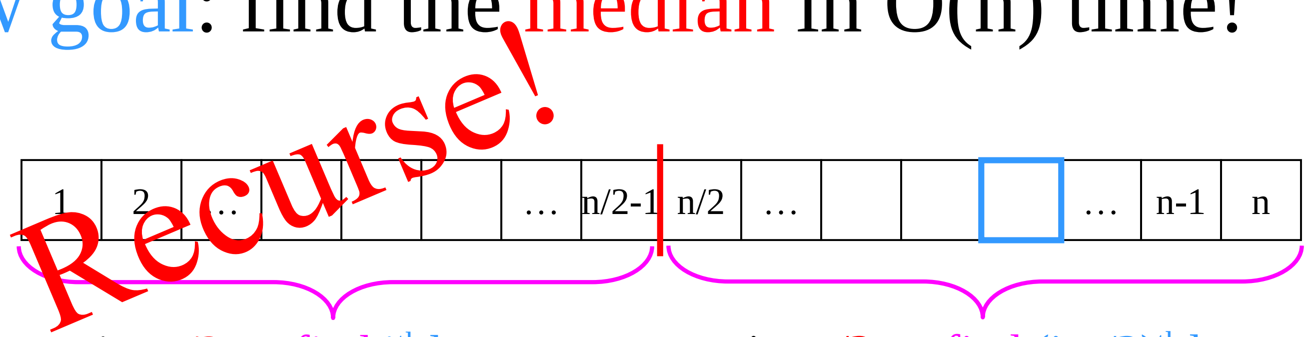
- Finding median in $O(n \log n)$ time is easy (why?)
- Assume $M(n) = c \cdot n = O(n)$

$$\Rightarrow T(n) < c \cdot (n + n/2 + n/4 + n/8 + \ldots)$$

$$< c \cdot (2n) = O(n)$$

Conclusion: linear-time median algorithm automatically yields linear-time i^{th} selection!

New goal: find the median in $O(n)$ time!



$i < n/2 \Rightarrow$ find i^{th} largest or $i > n/2 \Rightarrow$ find $(i-n/2)^{th}$ largest

QuickSelect (i^{th}-Largest)

Idea: partition around pivot and recurse

X:

p	p+1	…			q	q+1							…	r-1	r

k=q-p+1 elements

r – q elements

i < k ⇒ QuickSelect i^{th} largest or i > k ⇒ QuickSelect $(i\text{-}k)^{th}$ largest

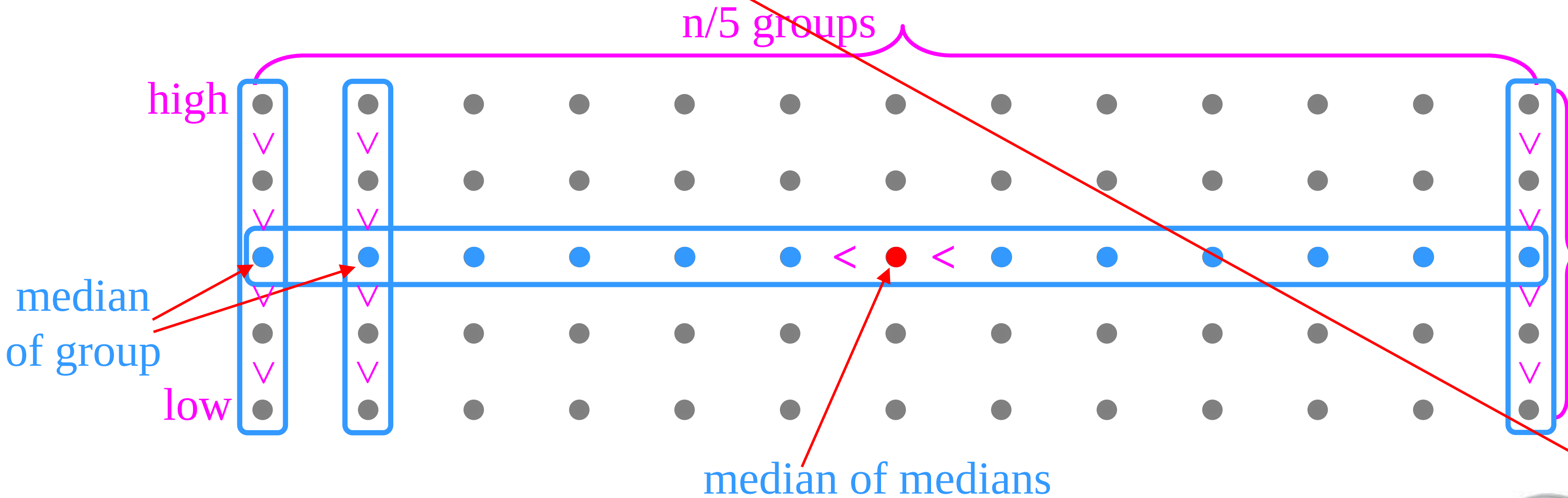
```
QuickSelect(X,p,r,i)
   if p == r then return(X[p])
   q = RandomPartition(X,p,r)
   k = q – p + 1
   If i ≤ k then return(QuickSelect(X,p,q,i))
              else return(QuickSelect(X,q+1,r,i-k))
```

- O(n) time average-case (analysis like QuickSort's)
- $\Theta(n^2)$ worst-case time (very rare)

Median in Linear Time

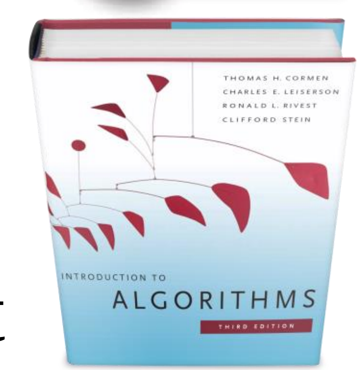
Idea: quickly eliminate a constant fraction & repeat

[Blum, Floyd, Pratt, Rivest, and Tarjan, 1973]



RSA

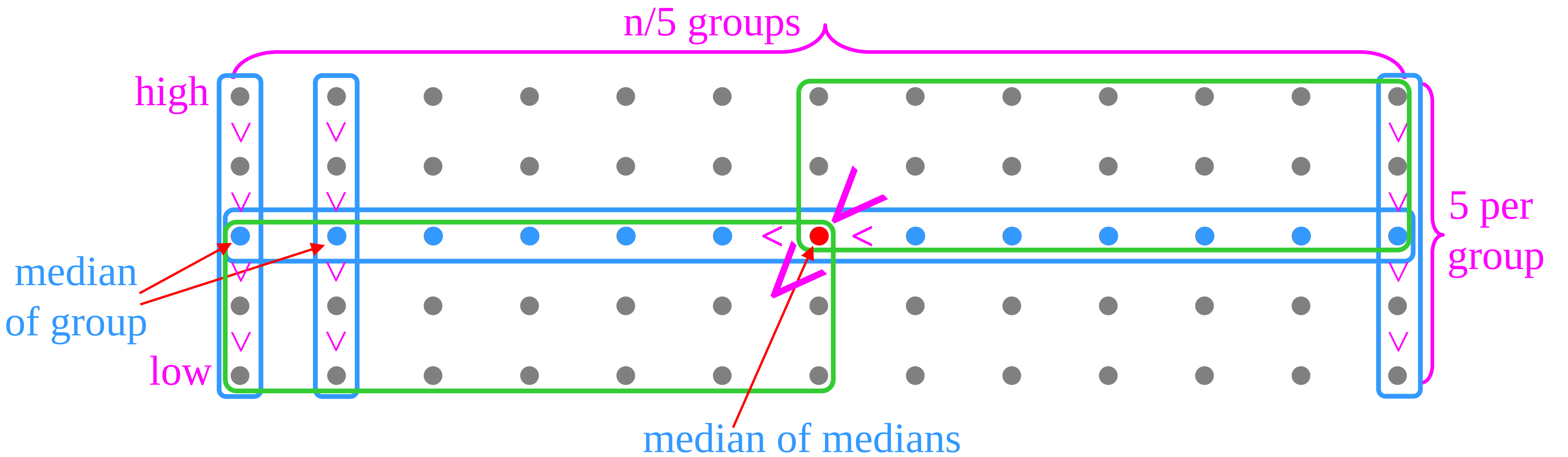
- Partition into n/5 groups of 5 each
- Sort each group (high to low)
- Compute median of medians (recursively)
- Move columns with larger medians to right
- Move columns with smaller medians to left



Median in Linear Time

Idea: quickly eliminate a constant fraction & repeat

[Blum, Floyd, Pratt, Rivest, and Tarjan, 1973]

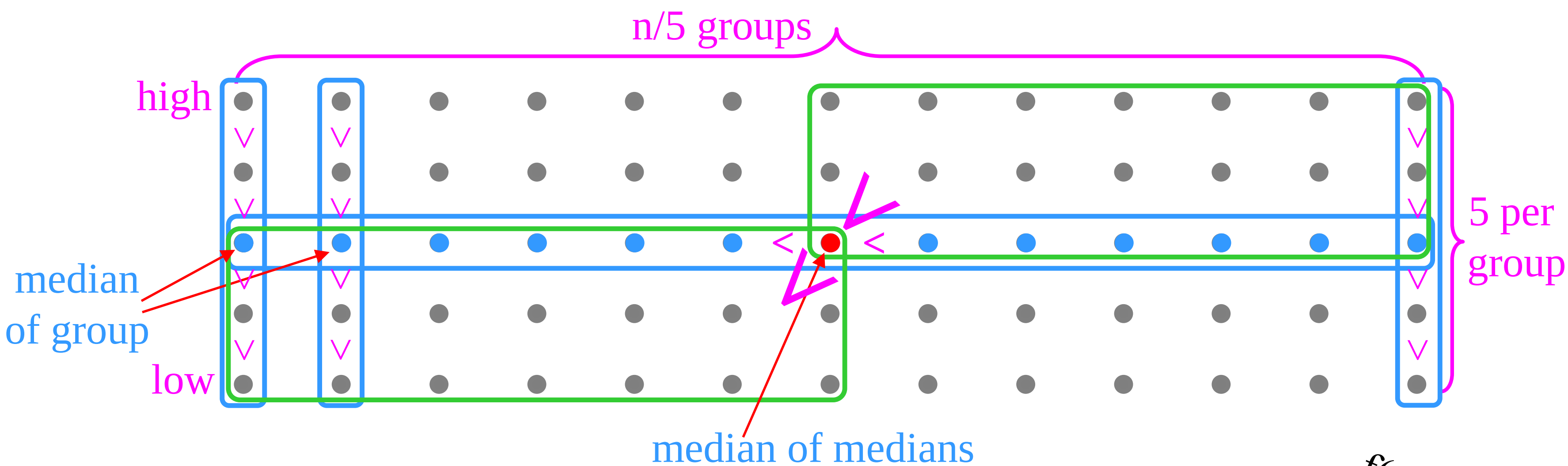


- > 3/10 of elements larger than median of medians
- > 3/10 of elements smaller than median of medians
- Partition all elements around median of medians
- Each partition contains at most 7n/10 elements
- Recurse on the proper partition (like in QuickSelect)

Median in Linear Time

Idea: quickly eliminate a constant fraction & repeat

[Blum, Floyd, Pratt, Rivest, and Tarjan, 1973]



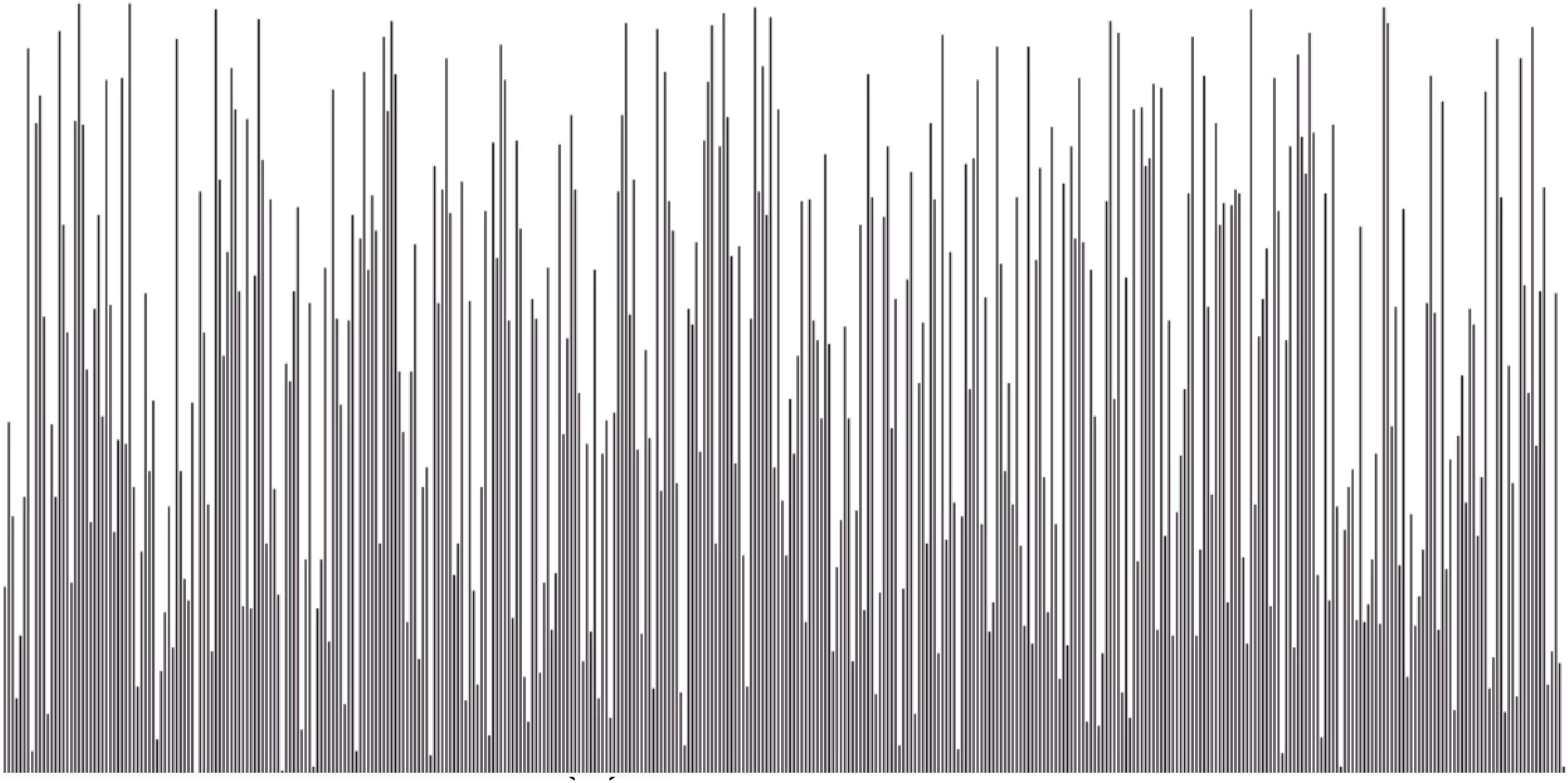
$$T(n) = T(n/5) + T(7n/10) + O(n)$$
$$= T(2n/10) + T(7n/10) + O(n)$$
$$\leq T(2n/10 + 7n/10) + O(n) \text{ since } T(n) = \Omega(n)$$
$$= T(9n/10) + O(n) \Rightarrow T(n) = O(n)$$

$f(n) = \Omega(n) \Rightarrow f(x + y) \geq f(x) + f(y)$

- Median is found in $\Theta(n)$ time worst-case!

Large constant Overhead!

Median in Linear Time



Exact upper bounds: < 24n, 5.4n, 3n, 2.95n, …+ o(n)

Exact lower bounds: >1.5n, 1.75n, 1.8n, 1.837n, 2n,…+ O(1)

Closing this comparisons gap further is still an open problem!