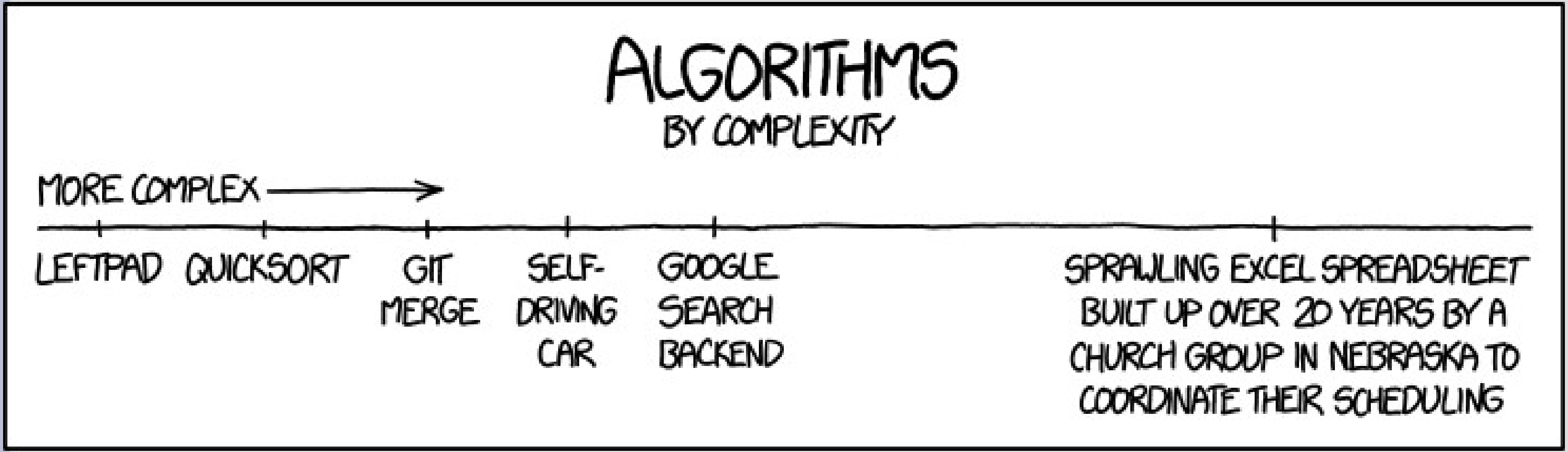
Sorting... more



Announcements

Homework posted, due next Sunday

Quicksort

Runtime:

Worst case?

Always pick lowest/highest element, so $O(n^2)$

Average?

Quicksort

Runtime:
Worst case?
Always pick lowest/highest element, so $O(n^2)$

Average?
Sort about half, so same as merge sort on average

Quicksort

Can bound number of checks against pivot:

Let $X_{i,j}$ = event A[i] checked to A[j]

$\mathrm{sum}_{i,j}\ X_{i,j}$ = total number of checks

$E[\mathrm{sum}_{i,j}\ X_{i,j}] = \mathrm{sum}_{i,j}\ E[X_{i,j}]$

$= \mathrm{sum}_{i,j}$ Pr(A[i] check A[j])

$= \mathrm{sum}_{i,j}$ Pr(A[i] or A[j] a pivot)

Quicksort

$= \text{sum}_{i,j}$ Pr(A[i] or A[j] a pivot)

$= \text{sum}_{i,j}$ (2 / j-i+1) // j-i+1 possibilties

$< \text{sum}_{i}$ O(lg n)

= O(n lg n)

Quicksort

Correctness:

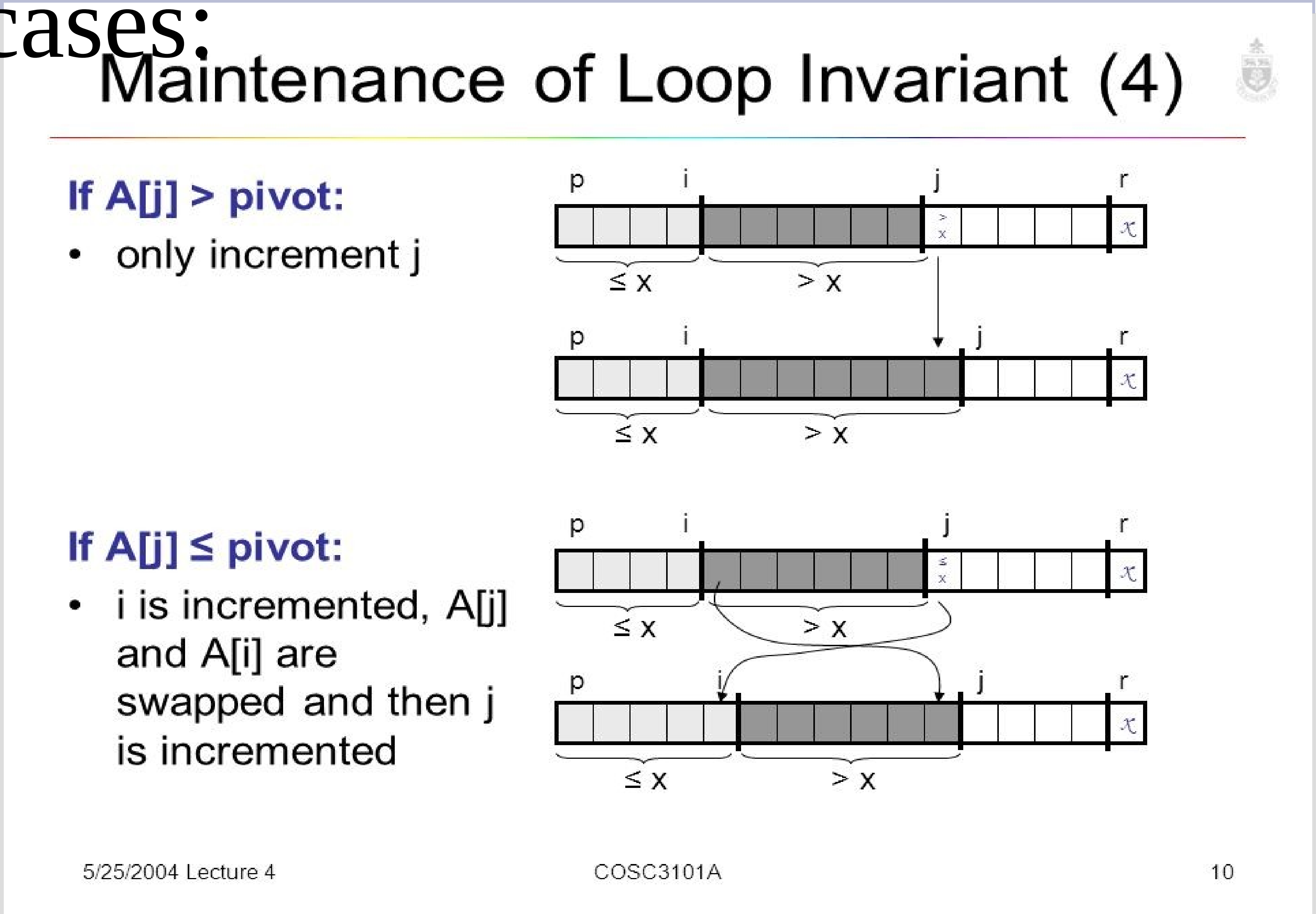
Base: Initially no elements are in the “smaller” or “larger” category

Step (loop): If A[j] < pivot it will be added to “smaller” and “smaller” will claim next spot, otherwise it it stays put and claims a “larger” spot

Termination: Loop on all elements...

Quicksort

Two cases:



Maintenance of Loop Invariant (4)

If A[j] > pivot:

- only increment j

If A[j] ≤ pivot:

- i is incremented, A[j] and A[i] are swapped and then j is incremented

Quicksort

Which is better for multi core, quicksort or merge sort?

If the average run times are the same, why might you choose quicksort?

Quicksort

Which is better for multi core, quicksort or merge sort?
Neither, quicksort front ends the processing, merge back ends

If the average run times are the same, why might you choose quicksort?

Quicksort

Which is better for multi core, quicksort or merge sort?

Neither, quicksort front ends the processing, merge back ends

If the average run times are the same, why might you choose quicksort?

Uses less space.

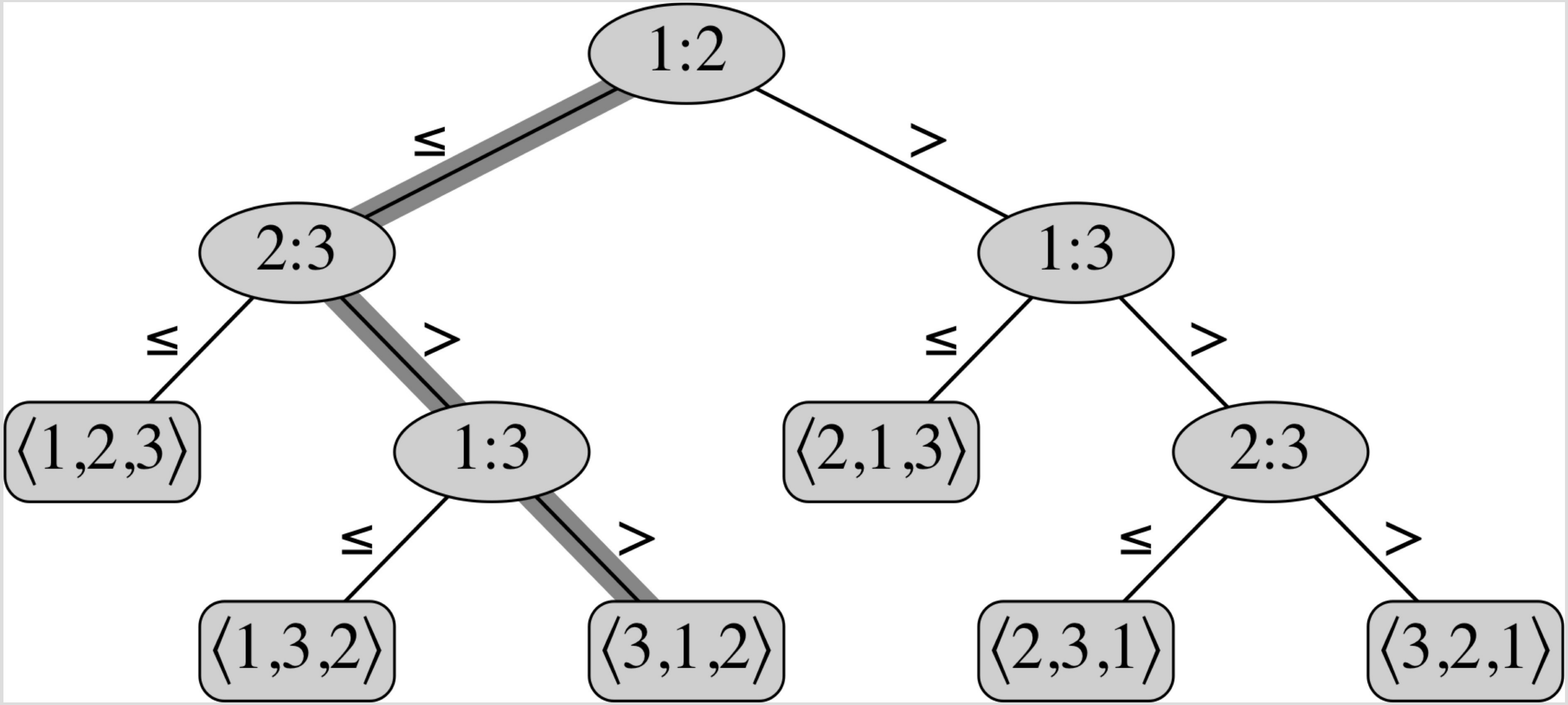
Sorting!

So far we have been looking at comparative sorts (where we only can compute $\leq$ or >, but have no idea on range of numbers)

The minimum running time for this type of algorithm is $\Theta(n \lg n)$

Sorting!

All n! permutations must be leaves



Worst case is tree height

Sorting!

A binary tree (either $<$ or $\geq$) of height h has 2^h leaves:

$2^h \geq n!$

$\lg(2^h) \geq \lg(n!)$ (Stirling's approx)

$h \geq (n \lg n)$

Comparison sort

Today we will make assumptions about the input sequence to get O(n) running time sorts

This is typically accomplished by knowing the range of numbers

Outline

Sorting... again!

- -Count sort
- -Bucket sort
- -Radix sort

Counting sort

1. Store in an array the number of times a number appears
2. Use above to find the last spot available for the number
3. Start from the last element, put it in the last spot (using 2.) decrease last spot array (2.)

Counting sort

A = input, B= output, C = count

```
for j = 1 to A.length
    C[ A[ j ]] = C[ A[ j ]] + 1
for i = 1 to k (range of numbers)
    C[ i ] = C[ i ] + C [ i – 1 ]
for j = A.length to 1
    B[ C[ A[ j ]]] = A[ j ]
    C[ A[ j ]] = C[ A[ j ]] - 1
```

Counting sort

You try!

k = range = 5 (numbers are 2-7)
Sort: {2, 7, 4, 3, 6, 3, 6, 3}

Counting sort

Sort: {2, 7, 4, 3, 6, 3, 6, 3}

1. Find number of times each number appears

C = {1, 3, 1, 0, 2, 1}

2, 3, 4, 5, 6, 7

Counting sort

Sort: {2, 7, 4, 3, 6, 3, 6, 3}

2. Change C to find last place of each element (first index is 1)

C = {1, 3, 1, 0, 2, 1}

{1, 4, 1, 0, 2, 1}

{1, 4, 5, 0, 2, 1}{1, 4, 5, 5, 7, 1}

{1, 4, 5, 5, 2, 1}{1, 4, 5, 5, 7, 8}

Counting sort

Sort: {2, 7, 4, 3, 6, 3, 6, 3}

3. Go start to last, putting each element into the last spot avail.

C = {1, 4, 5, 5, 7, 8}, last in list = 3
2 3 4 5 6 7
{ , , ,3, , , , }, C =
1 2 3 4 5 6 7 8 {1, 3, 5, 5, 7, 8}

Counting sort

Sort: {2, 7, 4, 3, 6, 3, 6, 3}

3. Go start to last, putting each element into the last spot avail.

C = {1, 4, 5, 5, 7, 8}, last in list = 6
2 3 4 5 6 7

{ , , ,3, , ,6, }, C =
1 2 3 4 5 6 7 8 {1, 3, 5, 5, 6, 8}

Counting sort

Sort: {2, 7, 4, 3, 6, 3, 6, 3}
1 2 3 4 5 6 7 8   2,3,4,5,6,7
{ ,   , ,3, , ,6, }, C={1,3,5,5,6,8}
{ ,   ,3,3, , ,6, }, C={1,2,5,5,6,8}
{ ,   ,3,3, ,6,6, }, C={1,2,5,5,5,8}
{ , 3,3,3, ,6,6, }, C={1,1,5,5,5,8}
{ , 3,3,3,4,6,6, }, C={1,1,4,5,5,8}
{ , 3,3,3,4,6,6,7}, C={1,1,4,5,5,7}

Counting sort

Run time?

Counting sort

Run time?

Loop over C once, A twice

k + 2n = O(n) as k a constant

Counting sort

Does counting sort work if you find the first spot to put a number in rather than the last spot?

If yes, write an algorithm for this in loose pseudo-code

If no, explain why

Counting sort

Sort: {2, 7, 4, 3, 6, 3, 6, 3}

C = {1,3,1,0,2,1} -> {1,4,5,5,7,8}
instead $C[\,i\,] = \text{sum}_{j<i}\, C[\,j\,]$

C' = {0, 1, 4, 5, 5, 7}
Add from start of original and increment

Counting sort

```
A = input, B= output, C = count
for j = 1 to A.length
    C[ A[ j ]] = C[ A[ j ]] + 1
for i = 2 to k (range of numbers)
    C'[ i ] = C'[ i-1 ] + C [ i – 1 ]
for j = A.length to 1
    B[ C[ A[ j ]]] = A[ j ]
    C[ A[ j ]] = C[ A[ j ]] + 1
```

Counting sort

Counting sort is stable, which means the last element in the order of repeated numbers is preserved from input to output

(in example, first '3' in original list is first '3' in sorted list)

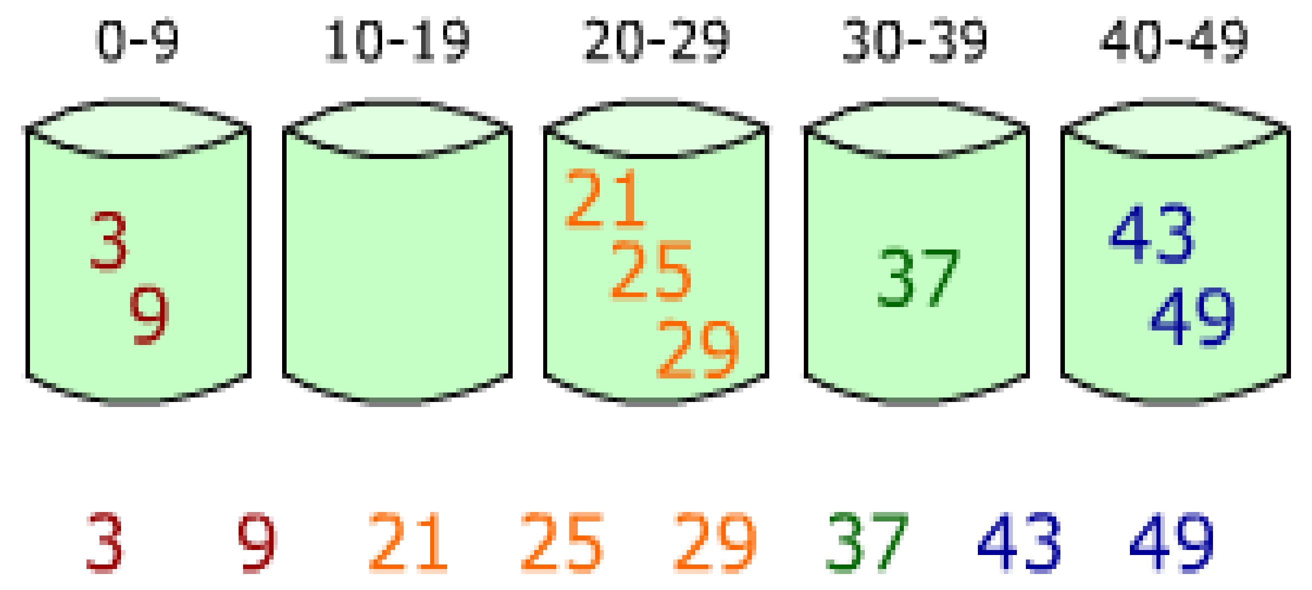
Bucket sort

1. Group similar items into a bucket

2. Sort each bucket individually

3. Merge buckets

Bucket sort



As a human, I recommend this sort if you have large n

Bucket sort

(specific to fractional numbers)

(also assumes n buckets for n numbers)

```
for i = 1 to n // n = A.length
  insert A[ i ] into B[floor(n A[ i ])+1]
for i = 1 to n // n = B.length
  sort list B[ i ] with insertion sort
concatenate B[1] to B[2] to B[3]...
```

Bucket sort

Run time?

Bucket sort

Run time?

$\Theta(n)$

Proof is gross... but with n buckets each bucket will have on average a constant number of elements

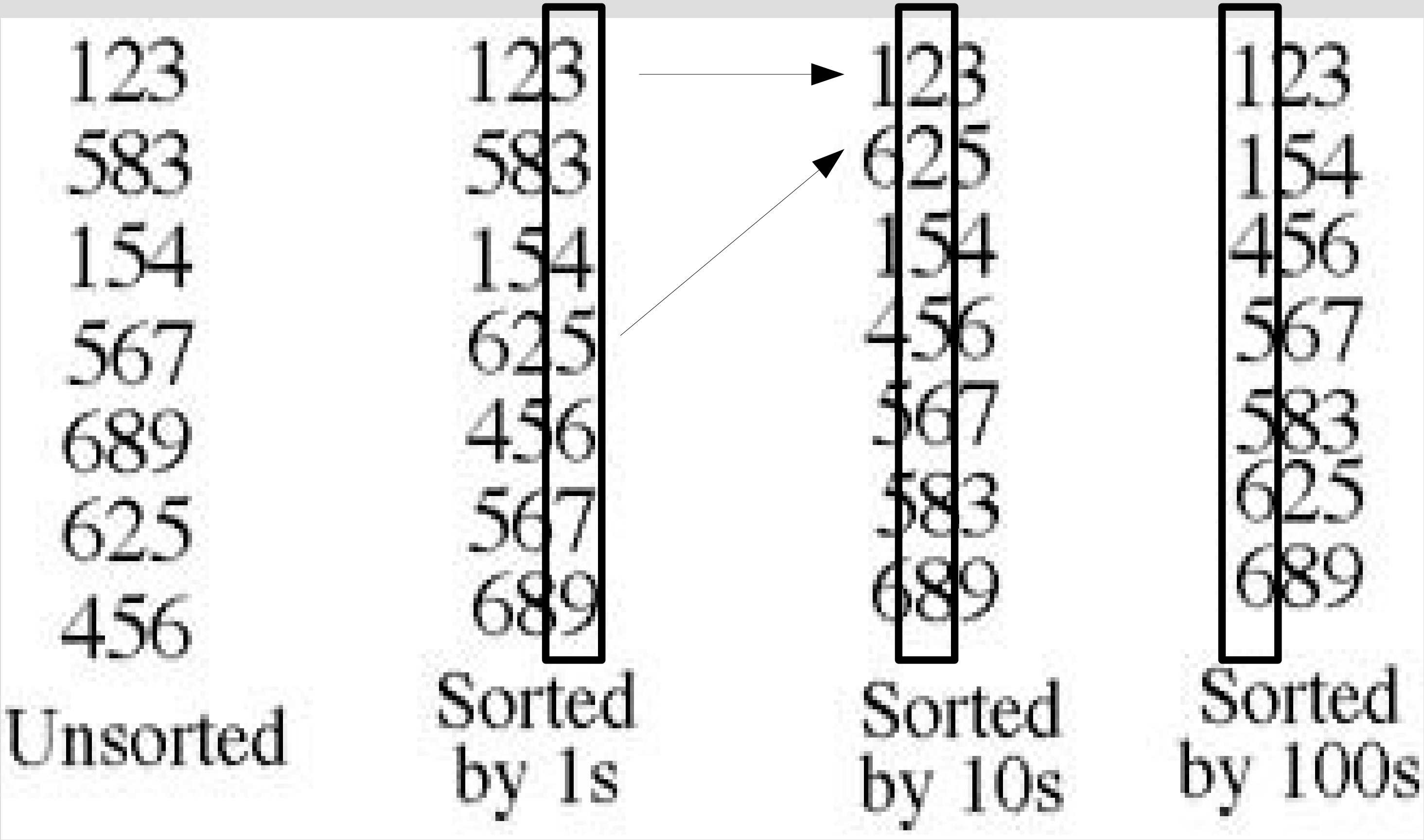
Radix sort

Use a **stable** sort to sort from the least significant digit to most

Psuedo code: (A=input)

```
for i = 1 to d
    stable sort of A on digit i
    // i.e. use counting sort
```

Radix sort



Stable means you can draw lines without crossing for a single digit

Radix sort

Run time?

Radix sort

Run time?

$O(\ (b/r)\ (n+2^r)\ )$

b-bits total, r bits per 'digit'

$d = b/r$ digits

Each count sort takes $O(n + 2^r)$

runs count sort d times...

$O(\ d(n+2^r)) = O(\ b/r\ (n + 2^r))$

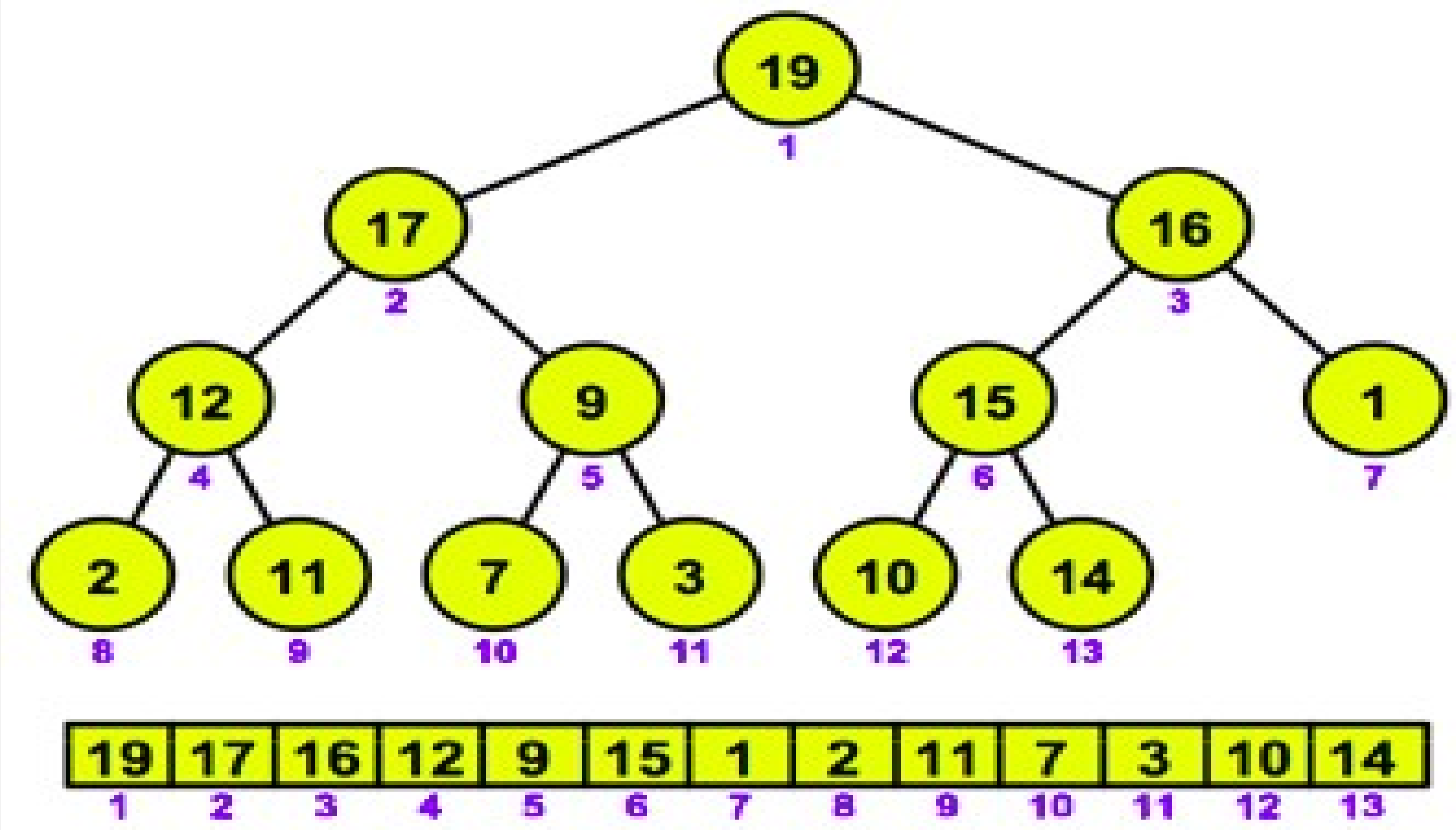
Radix sort

Run time?

if b < lg(n), $\Theta(n)$

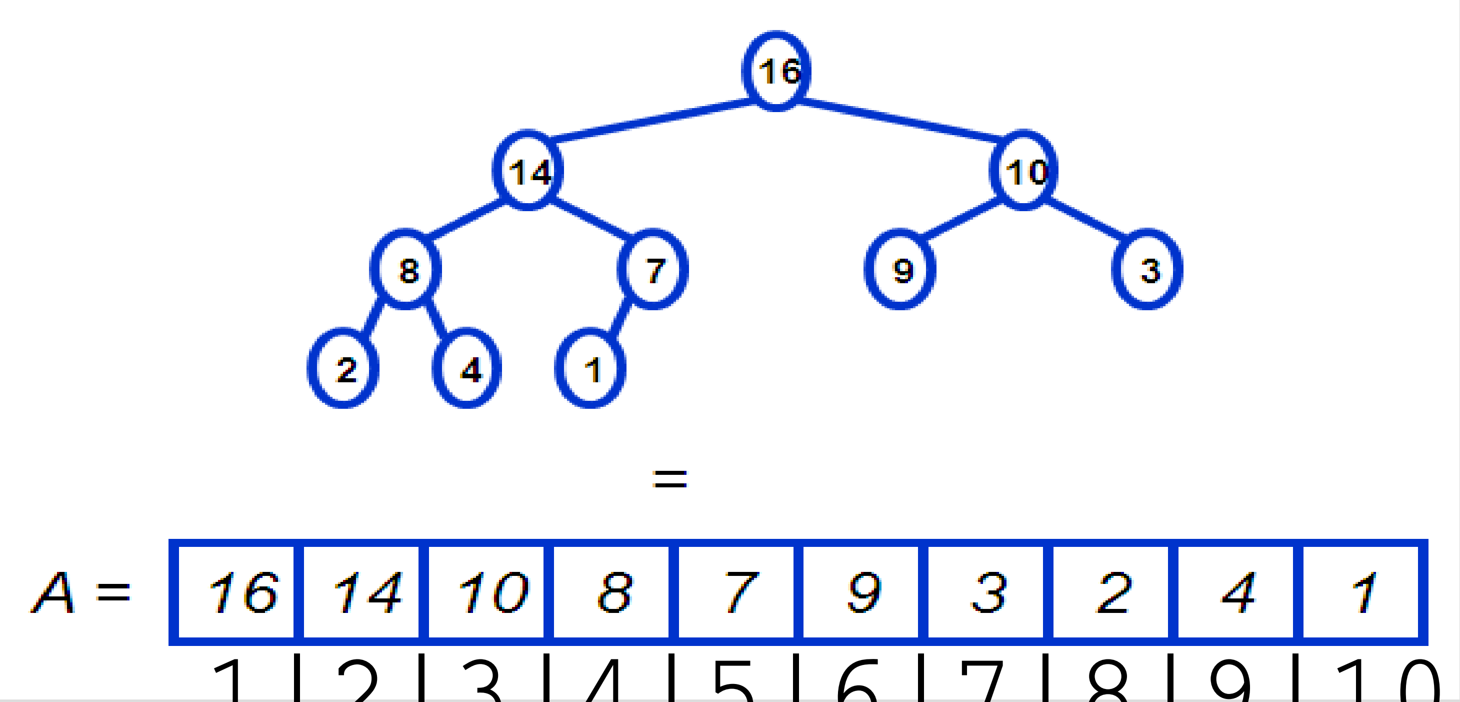
if b $\geq$ lg(n), $\Theta(n \lg n)$

Heapsort


19
1
17
2
16
3
12
4
9
5
15
6
1
7
2
8
11
9
7
10
3
11
10
12
14
13
19 17 16 12 9 15 1 2 11 7 3 10 14
1 2 3 4 5 6 7 8 9 10 11 12 13

Binary tree as array

It is possible to represent binary trees as an array

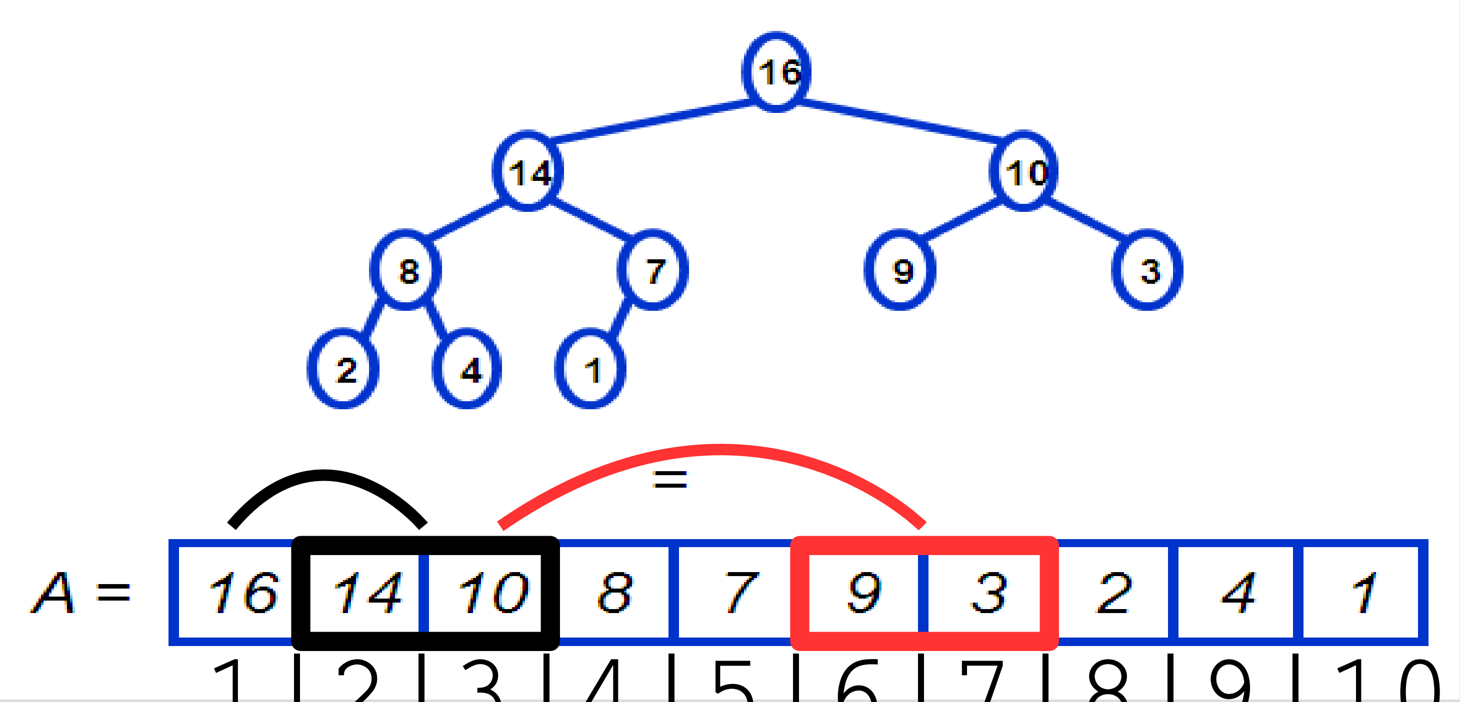


A =	16	14	10	8	7	9	3	2	4	1

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

Binary tree as array

index 'i' is the parent of '2i' and '2i+1'



1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

Binary tree as array

Is it possible to represent any tree with a constant branching factor as an array?

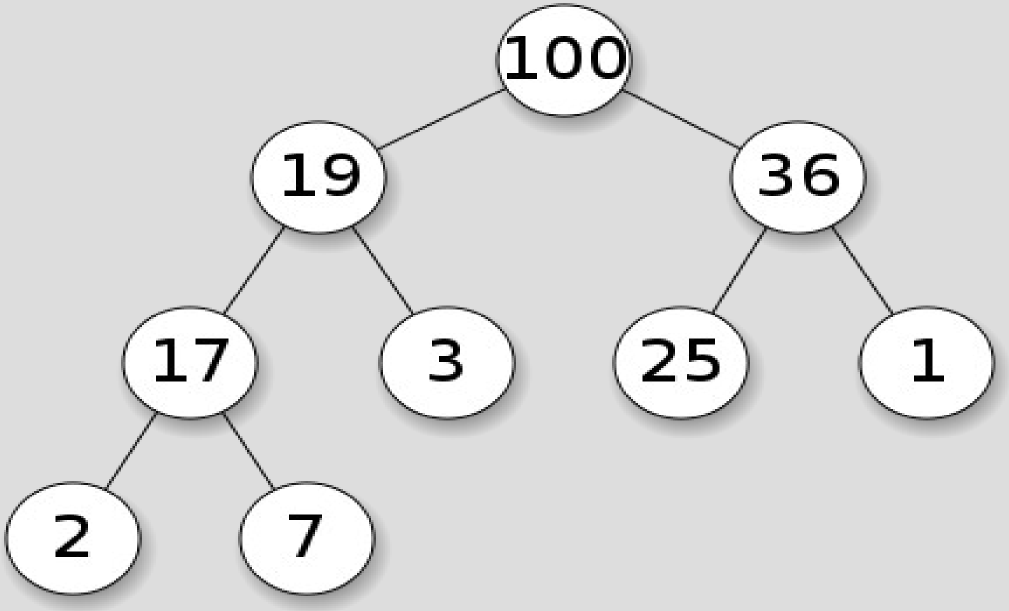
Binary tree as array

Is it possible to represent any tree with a constant branching factor as an array?

Yes, but the notation is awkward

Heaps

A max heap is a tree where the parent is larger than its children (A min heap is the opposite)



Heapsort

The idea behind heapsort is to:

1. Build a heap
2. Pull out the largest (root) and re-compile the heap
3. (repeat)

Heapsort

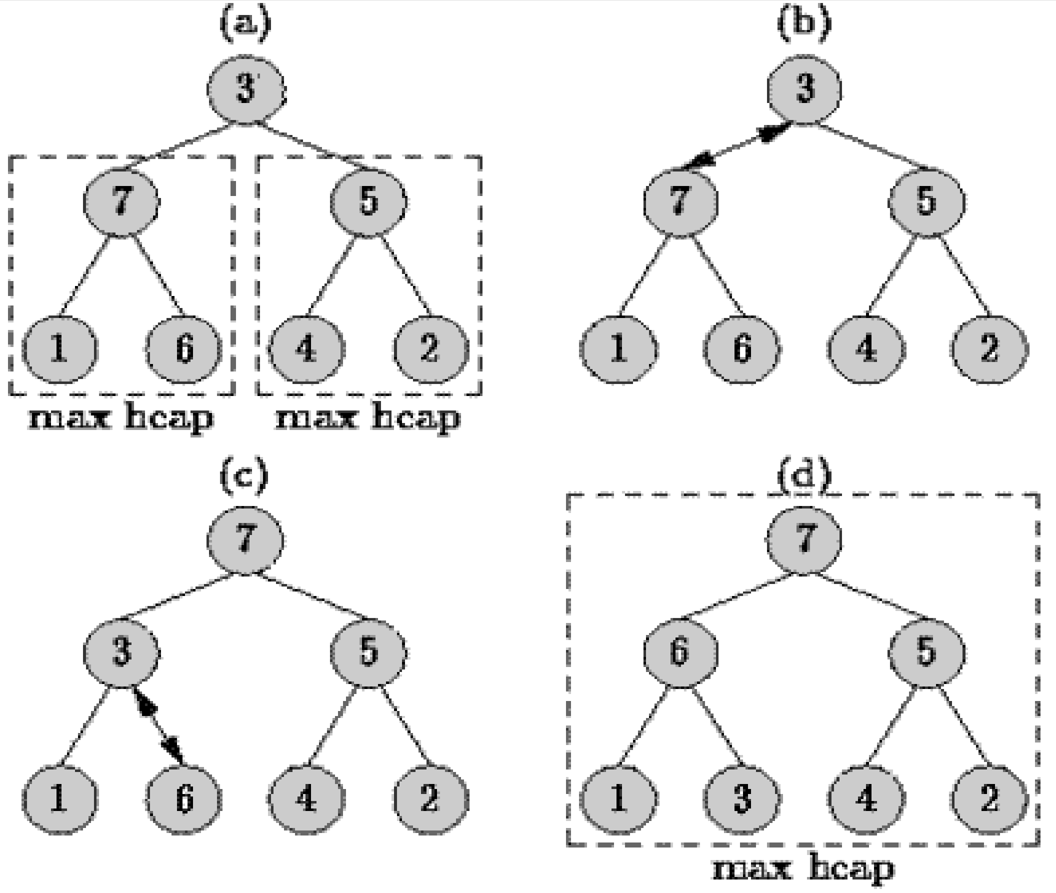
To do this, we will define subroutines:

1. Max-Heapify = maintains heap property

2. Build-Max-Heap = make sequence into a max-heap

Max-Heapify

Input: a root of two max-heaps
Output: a max-heap



Max-Heapify

```
Pseudocode Max-Heapify(A,i):
left = left(i)      // 2*i
right = right(i)  // 2*i+1
L = arg_max( A[left], A[right], A[ i ])
if (L not i)
   exchange A[ i ] with A[ L ]
   Max-Heapify(A, L)
// now make me do it!
```

Max-Heapify

Runtime?

Max-Heapify

Runtime?

Obviously (is it?): lg n

T(n) = T(2/3 n) + O(1) // why?

Or...

T(n) = T(1/2 n) + O(1)

Master's theorem

Master's theorem: (proof 4.6)

For $a \geq 1$, $b \geq 1$, $T(n) = a\,T(n/b) + f(n)$

If f(n) is... (3 cases)

$O(n^c)$ for $c < \log_b a$, $T(n)$ is $\Theta(n^{\log_b a})$

$\Theta(n^{\log_b a})$, then $T(n)$ is $\Theta(n^{\log_b a} \lg n)$

$\Omega(n^c)$ for $c > \log_b a$, $T(n)$ is $\Theta(f(n))$

Max-Heapify

Runtime?

Obviously (is it?): lg n

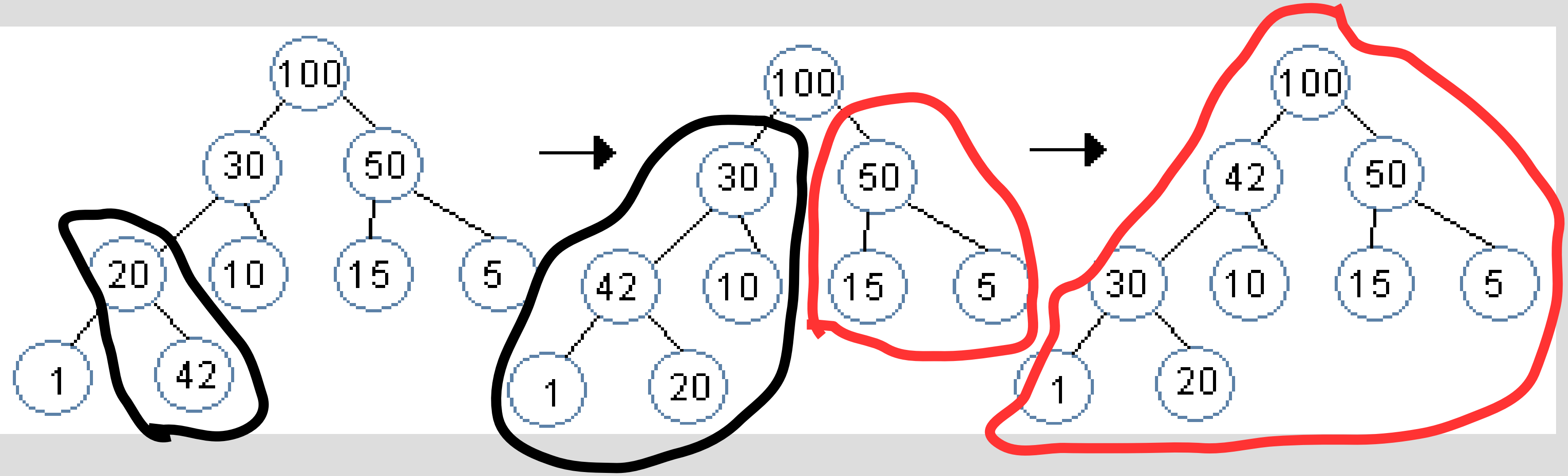
T(n) = T(2/3 n) + O(1) // why?
Or...
T(n) = T(1/2 n) + O(1) = O(lg n)

Build-Max-Heap

Next we build a full heap from an unsorted sequence

```
Build-Max-Heap(A)
for i = floor( A.length/2 ) to 1
   Heapify(A, i)
```

Build-Max-Heap



Red part is already Heapified

Build-Max-Heap

Correctness:

Base: Each alone leaf is a max-heap

Step: if A[i] to A[n] are in a heap, then Heapify(A, i-1) will make i-1 a heap as well

Termination: loop ends at i=1, which is the root (so all heap)

Build-Max-Heap

Runtime?

Build-Max-Heap

Runtime?

O(n lg n) is obvious, but we can get a better bound...

Show ceiling($n/2^{h+1}$) nodes at any height 'h'

Build-Max-Heap

Heapify from height 'h' takes O(h)

$\mathrm{sum}_{h=0}^{\lg n}$ ceiling($n/2^{h+1}$) O(h)

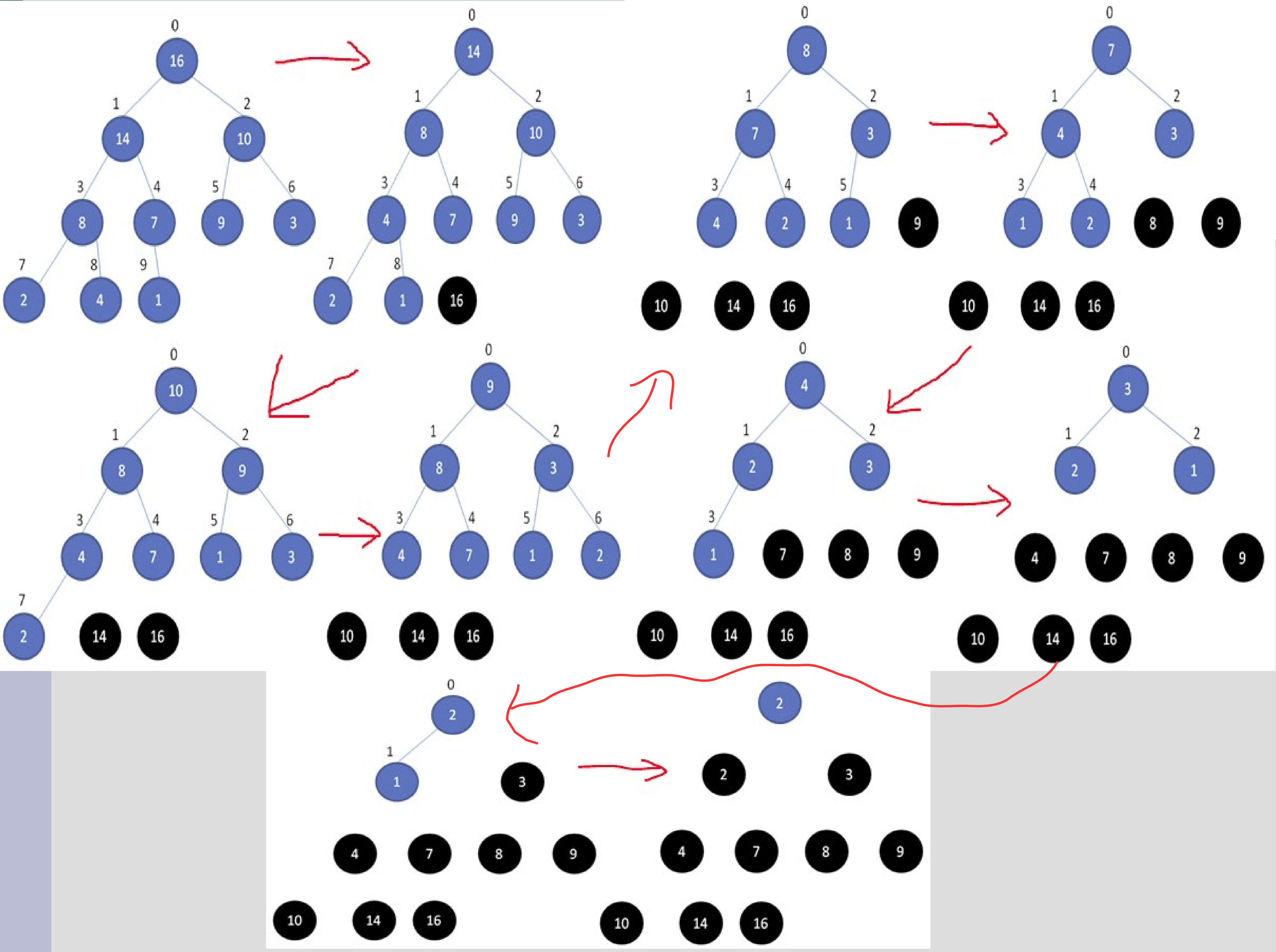
=O(n $\mathrm{sum}_{h=0}^{\lg n}$ ceiling($h/2^{h+1}$))

($\mathrm{sum}_{x=0}^{\infty}$ $k\ x^k = x/(1-x)^2$, x=1/2)

=O(n 4/2) = O(n)

Heapsort

```
Heapsort(A):
Build-Max-Heap(A)
for i = A.length to 2
    Swap A[ 1 ], A[ i ]
    A.heapsize = A.heapsize – 1
    Max-Heapify(A, 1)
```


16 14 10 8 7 9 3 2 4 1
14 8 10 4 7 9 3 2 1 16
8 7 3 4 2 1 9 10 14 16
7 4 3 1 2 8 9 10 14 16
10 8 9 4 7 1 3 2 14 16
9 8 3 4 7 1 2 10 14 16
4 2 3 1 7 8 9 10 14 16
3 2 1 4 7 8 9 10 14 16
2 1 3 4 7 8 9 10 14 16
2
2 3 4 7 8 9 10 14 16

Heapsort

Runtime?

Heapsort

Runtime?

Run Max-Heapify O(n) times
So... O(n lg n)

Priority queues

Heaps can also be used to implement priority queues (i.e. airplane boarding lines)

Operations supported are: Insert, Maximum, Exctract-Max and Increase-key

Priority queues

```
Maximum(A):
    return A[ 1 ]

Extract-Max(A):
max = A[1]
A[1] = A.heapsize
A.heapsize = A.heapsize – 1
Max-Heapify(A, 1),    return max
```

Priority queues

```
Increase-key(A, i, key):
A[ i ] = key
while ( i>1 and A [floor(i/2)] < A[i])
   swap A[ i ], A [floor(i/2)]
   i = floor(i/2)
```

Opposite of Max-Heapify... move high keys up instead of low down

Priority queues

```
Insert(A, key):
    A.heapsize = A.heapsize + 1
    A [ A.heapsize] = $-\infty$
    Increase-key(A, A.heapsize, key)
```

Priority queues

Runtime?

Maximum =
Extract-Max =
Increase-Key =
Insert =

Priority queues

Runtime?

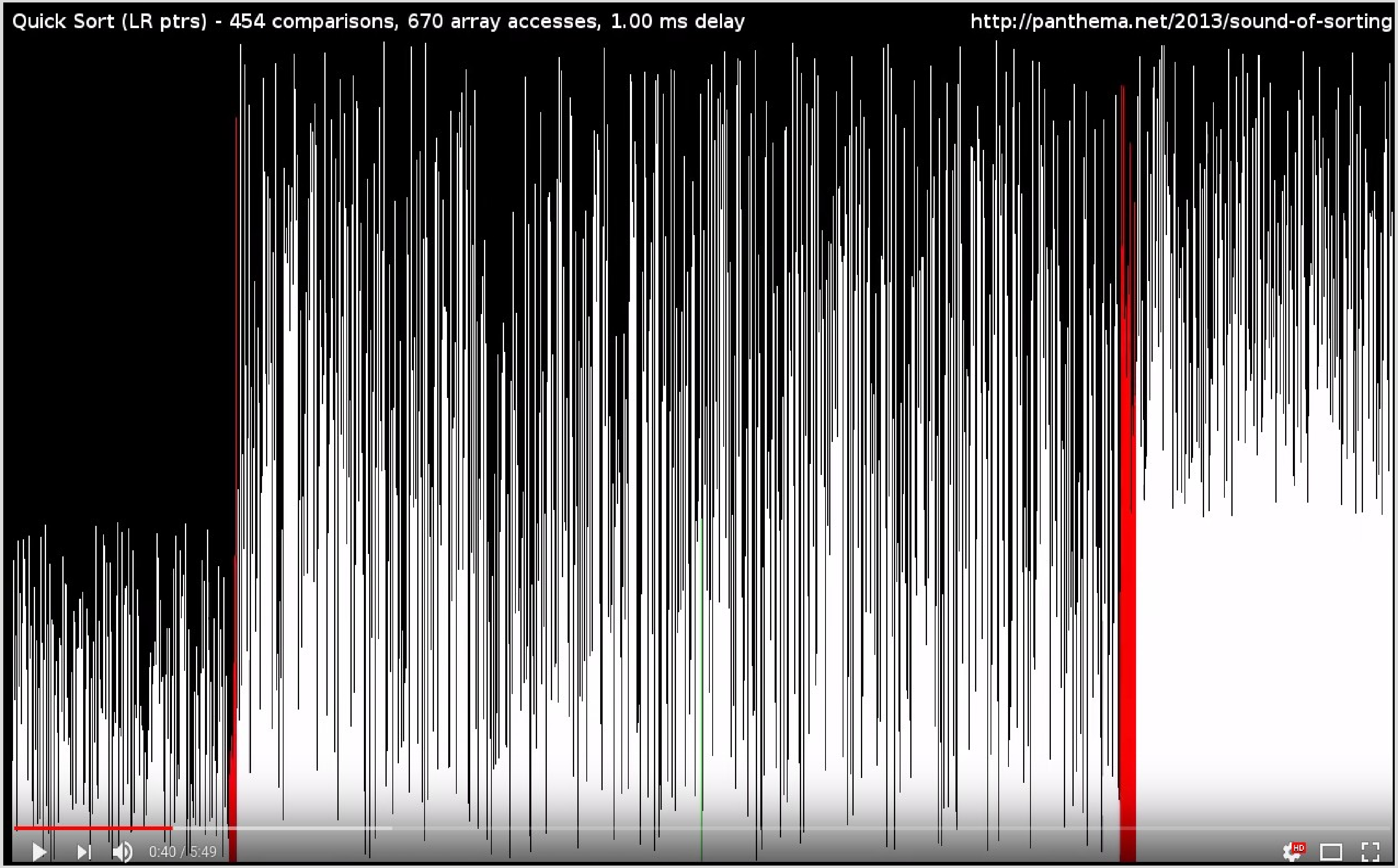
Maximum = O(1)
Extract-Max = O(lg n)
Increase-Key = O(lg n)
Insert = O(lg n)

Sorting comparisons:

Name	Average	Worst-case
Insertion[s,i]	O(n^2)	O(n^2)
Merge[s,p]	O(n lg n)	O(n lg n)
Heap[i]	O(n lg n)	O(n lg n)
Quick[p]	O(n lg n)	O(n^2)
Counting[s]	O(n + k)	O(n + k)
Radix[s]	O(d(n+k))	O(d(n+k)
Bucket[s,p]	O(n)	O(n^2)

Sorting comparisons:

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



Quick Sort (LR ptrs) - 454 comparisons, 670 array accesses, 1.00 ms delay

http://panthema.net/2013/sound-of-sorting