CS 600.226: Data Structures

Michael Schatz

Nov 26, 2016

Lecture 34: Advanced Sorting

Assignment 9: StringOmics

Out on: November 16, 2018

Due by: November 30, 2018 before 10:00 pm

Collaboration: None

Grading:

- Packaging 10%,
- Style 10% (where applicable),
- Testing 10% (where applicable),
- Performance 10% (where applicable),
- Functionality 60% (where applicable)

Overview

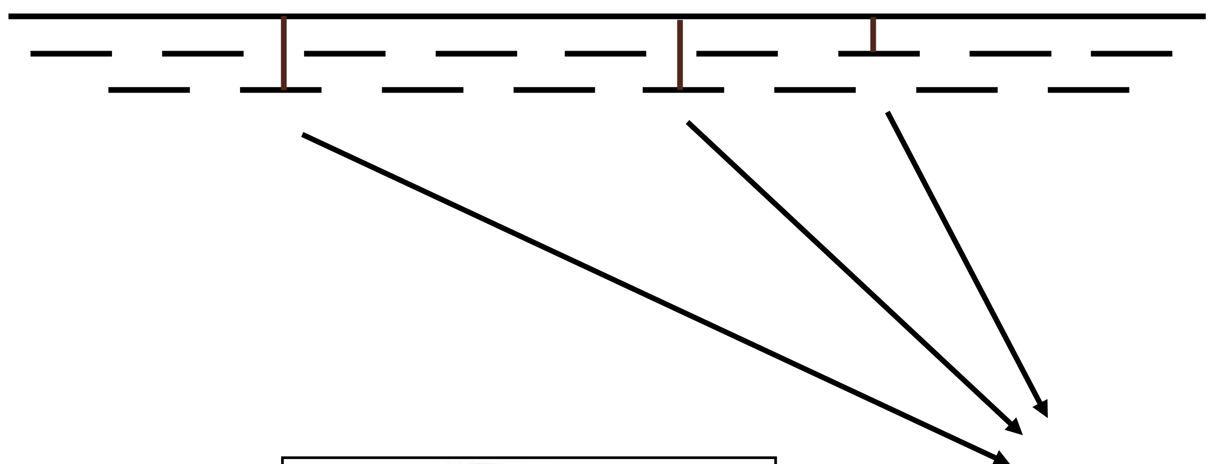
The ninth assignment focuses on data structures and operations on strings. In this assignment you will implement encoding and decoding using the Burrows Wheeler Transform as well as encoding and decoding in a simple form of run length encoding. In the final problem you will be asked to measure the space savings using run length encoding with and without applying the Burrows Wheeler Transform first.

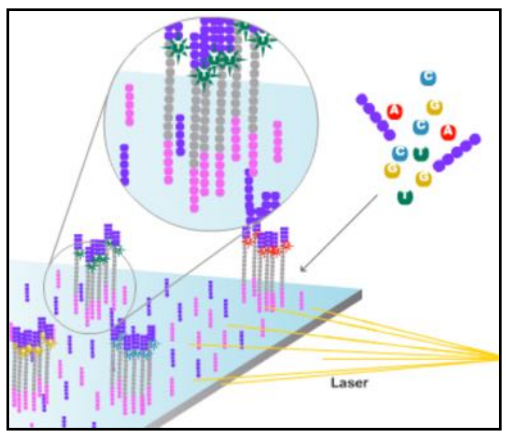
Remember: javac –Xlint:all & checkstyle *.java & Junit
(No JayBee)

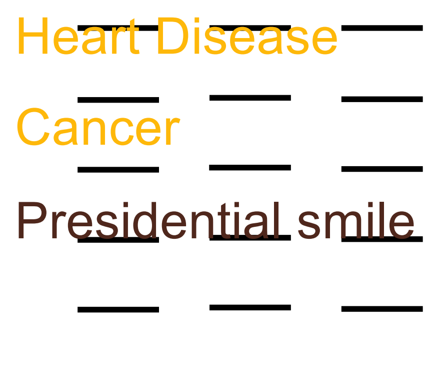
Part 1: StringOmics Recap

Personal Genomics

How does your genome compare to the reference?



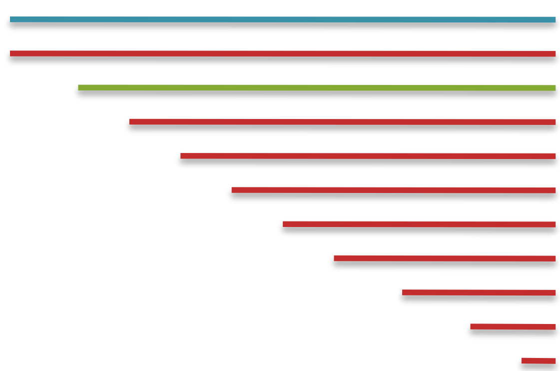




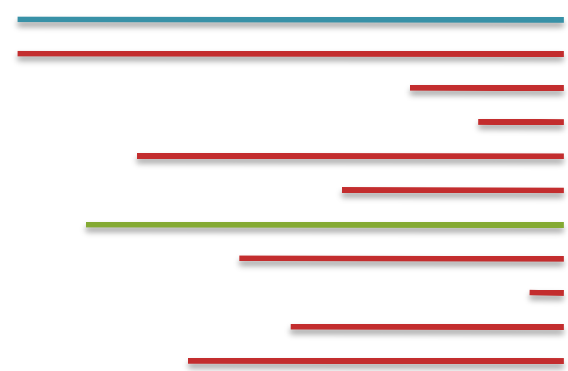
Suffix Arrays: Searching the Phone Book

- What if we need to check many queries?
  - We don't need to check every page of the phone book to find 'Schatz'
  - Sorting alphabetically lets us immediately skip 96% (25/26) of the book *without any loss in accuracy*

- Sorting the genome: Suffix Array (Manber & Myers, 1991)
  - Sort every suffix of the genome



Split into n suffixes



Sort suffixes alphabetically

[Challenge Question: How else could we split the genome?]

Binary Search Analysis

- Binary Search

  Initialize search range to entire list

  mid = (hi+lo)/2; middle = suffix[mid]

  if query matches middle: done

  else if query < middle: pick low range

  else if query > middle: pick hi range

  Repeat until done or empty range [WHEN?]

- Analysis
  - More complicated method
  - How many times do we repeat?
    - How many times can it cut the range in half?
    - Find smallest x such that: $n/(2^x) \leq 1$; $x = \lg_2(n)$ [32]

- Total Runtime: $O(m \lg n)$
  - More complicated, but much faster!
  - Looking up a query loops 32 times instead of 3B

[How long does it take to search 6B or 24B nucleotides?]

Exact Matching Review & Overview

Where is GATTACA in the human genome?

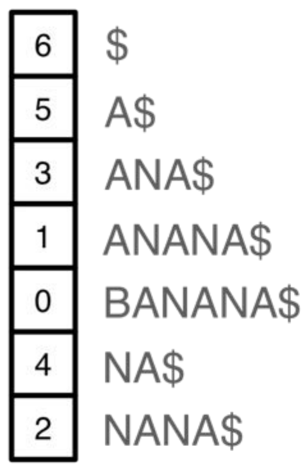
Brute Force
(3 GB)

```
BANANA
BAN
 ANA
  NAN
   ANA
```

O(m * n)

Slow & Easy

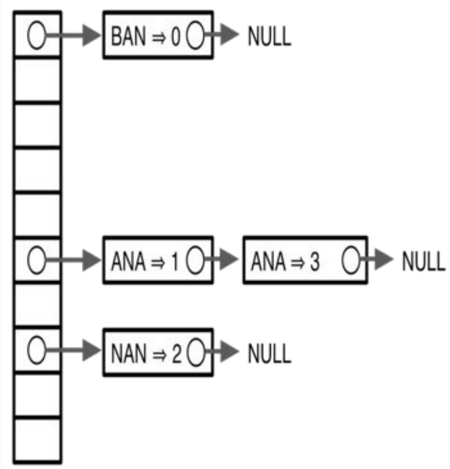
Suffix Array
(>15 GB)



O(m + lg n)

Full-text index

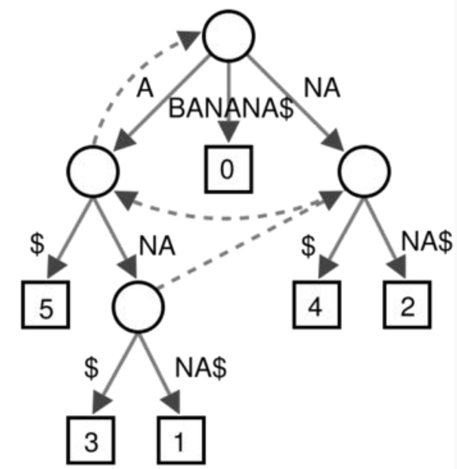
Hash Table
(>15 GB)



O(1)

Fixed-length lookup

Suffix Tree
(>51 GB)



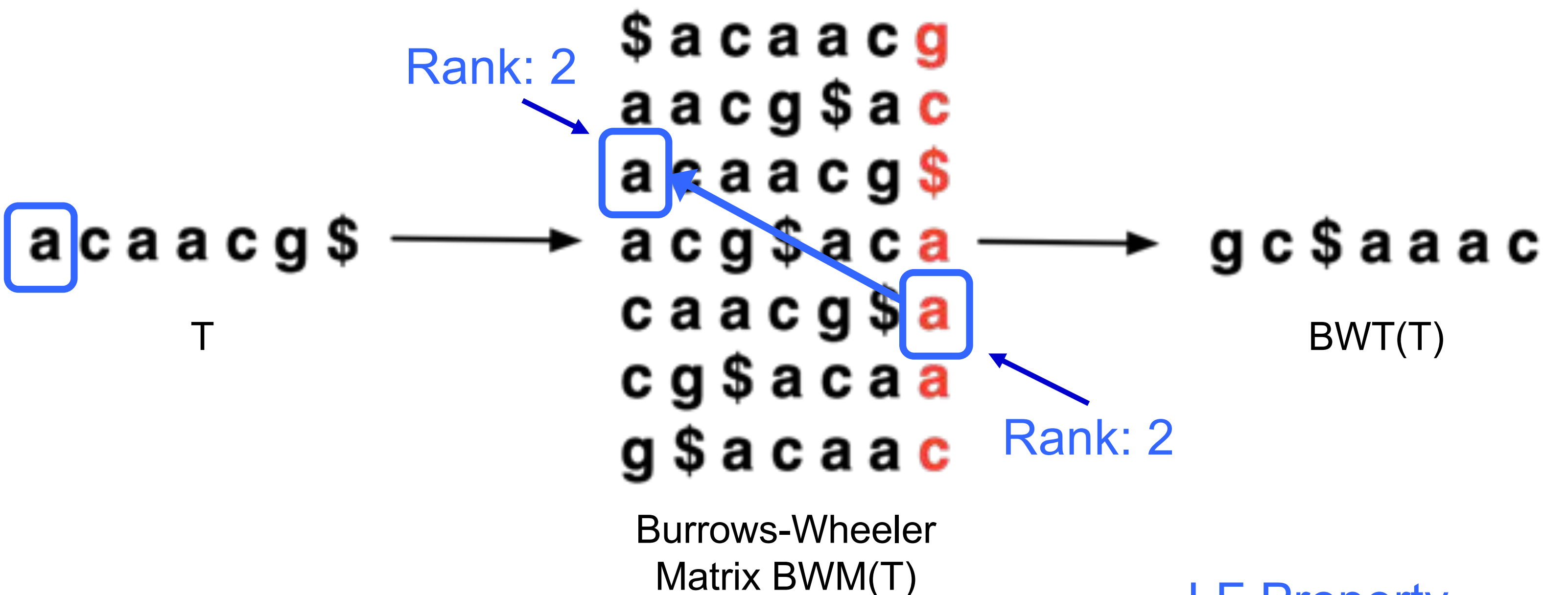
O(m)

Full-text, but bulky

*** These are general techniques applicable to any text search problem ***

Burrows-Wheeler Transform

- Reversible permutation of the characters in a text



LF Property
implicitly encodes
Suffix Array

- BWT(T) is the index for T

A block sorting lossless data compression algorithm.
Burrows M, Wheeler DJ (1994) *Digital Equipment Corporation.* Technical Report 124

Burrows-Wheeler Transform

- Recreating T from BWT(T)
  - Start in the first row and apply **LF** repeatedly, accumulating predecessors along the way

[Decode this BWT string: ACTGA$TTA]

Run Length Encoding

ref[614]:

```
It_was_the_best_of_times,_it_was_the_worst_of_times,_it_was_the_age_
of_wisdom,_it_was_the_age_of_foolishness,_it_was_the_epoch_of_belief
,_it_was_the_epoch_of_incredulity,_it_was_the_season_of_Light,_it_wa
s_the_season_of_Darkness,_it_was_the_spring_of_hope,_it_was_the_wint
er_of_despair,_we_had_everything_before_us,_we_had_nothing_before_us
,_we_were_all_going_direct_to_Heaven,_we_were_all_going_direct_the_o
ther_way_-_in_short,_the_period_was_so_far_like_the_present_period,_
that_some_of_its_noisiest_authorities_insisted_on_its_being_received
,_for_good_or_for_evil,_in_the_superlative_degree_of_comparison_only.$
```

rle(bwt)[464]:

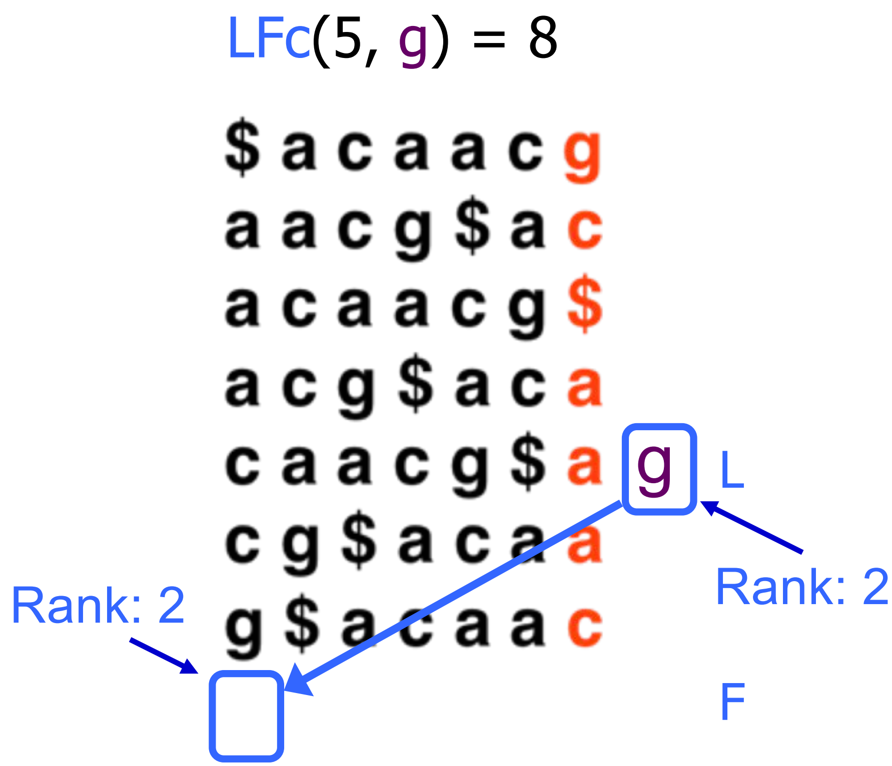
```
.dlms2ftysesdtrsns_y_2$_yfofe4tg2sfefefg2e2drofr,l2re2f-,fs,9nfrsdn2
hereghet2edndete2ge2nste2,s5t,es3ns2f2te2dt10r,4e3feh2_2p_2fpDw11e2h
l_ew_5eo2_ne3oa2eo2_4seph2r2hvh2w2egmgh7kr2w2h2s2Hr3vtr2ib2dbcbvs_2t
hw2p3vm2irdn2ib_2eo12_4e2n6a2i_3ec2_2t18s_tsgltsLlvt2_3h2o2re_wr2ad2
wlors_9r_2lteiril2re_oua2no2i2oeo4i3hki6o_2ieitsp2ioi_12g2nodsc_s3_g
fhf_f3hwh_nsmo_2ue2_sio3ae4o2_i2cgp2e2aoaeo2e2s2eu2teta11i_2ei_in_2a
2ie_e3rei
```

Saved 614-464 = 150 bytes (24%) with zero loss of information!

Common to save 50% to 90% on real world files with bzip2

BWT Exact Matching

- **LFc**(r, c) does the same thing as **LF**(r) but it ignores r's actual final character and "pretends" it's c:



BWT Exact Matching

- Start with a range, (**top**, **bot**) encompassing all rows and repeatedly apply **LFc**:

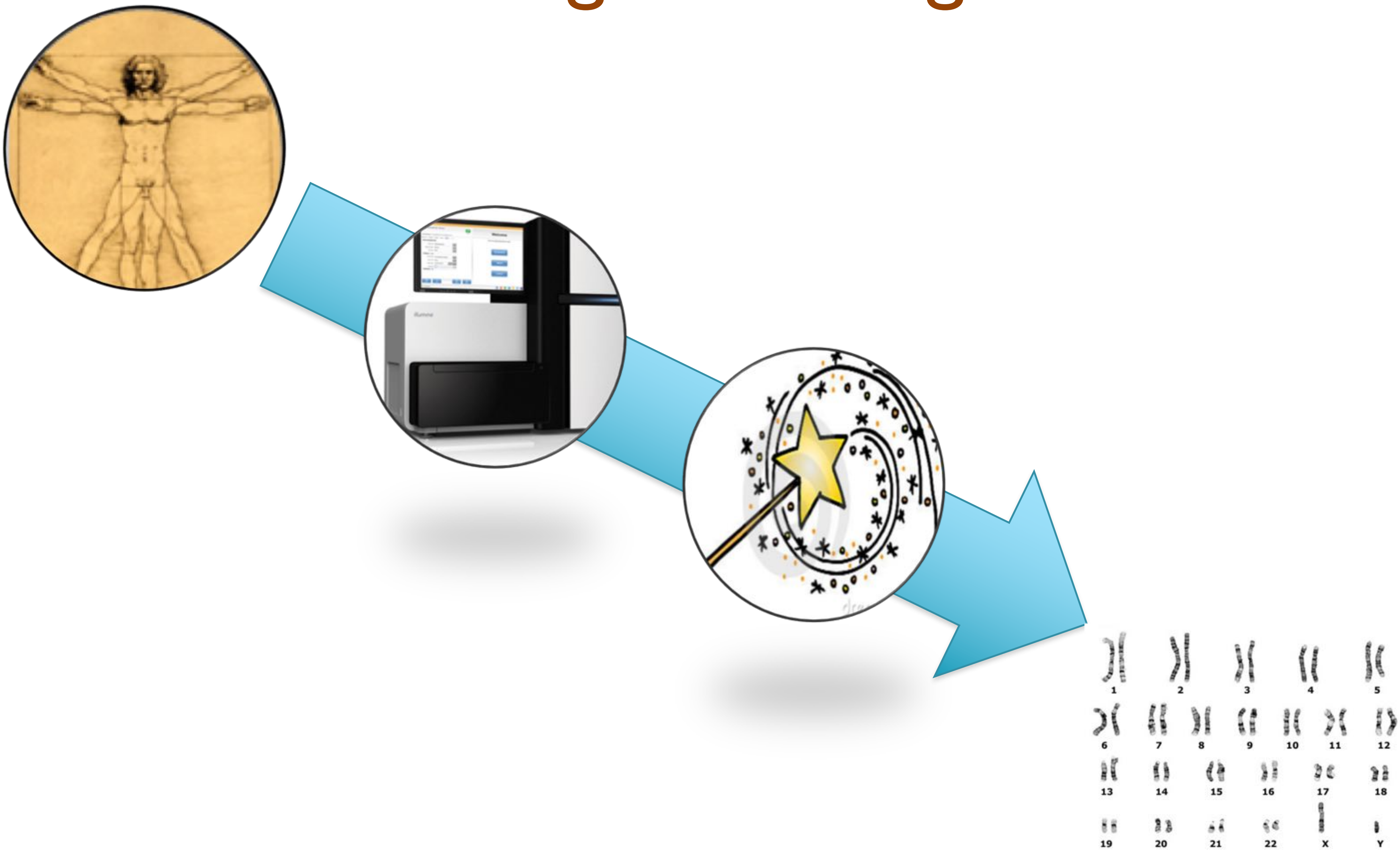
  top = **LFc**(**top**, **qc**); **bot** = **LFc**(**bot**, **qc**)

  qc = the next character to the left in the query

Ferragina P, Manzini G: Opportunistic data structures with applications. *FOCS. IEEE Computer Society; 2000.*

[Search for TTA this BWT string: ACTGA$TTA]

Assembling the first genome



Like Dickens, we must computationally reconstruct a genome from short fragments

de Bruijn Graph Construction

- $D_k = (V,E)$
  - V = All length-k subfragments ($k < l$)
  - E = Directed edges between consecutive subfragments
    - Nodes overlap by k-1 words

Original Fragment

It was the best of

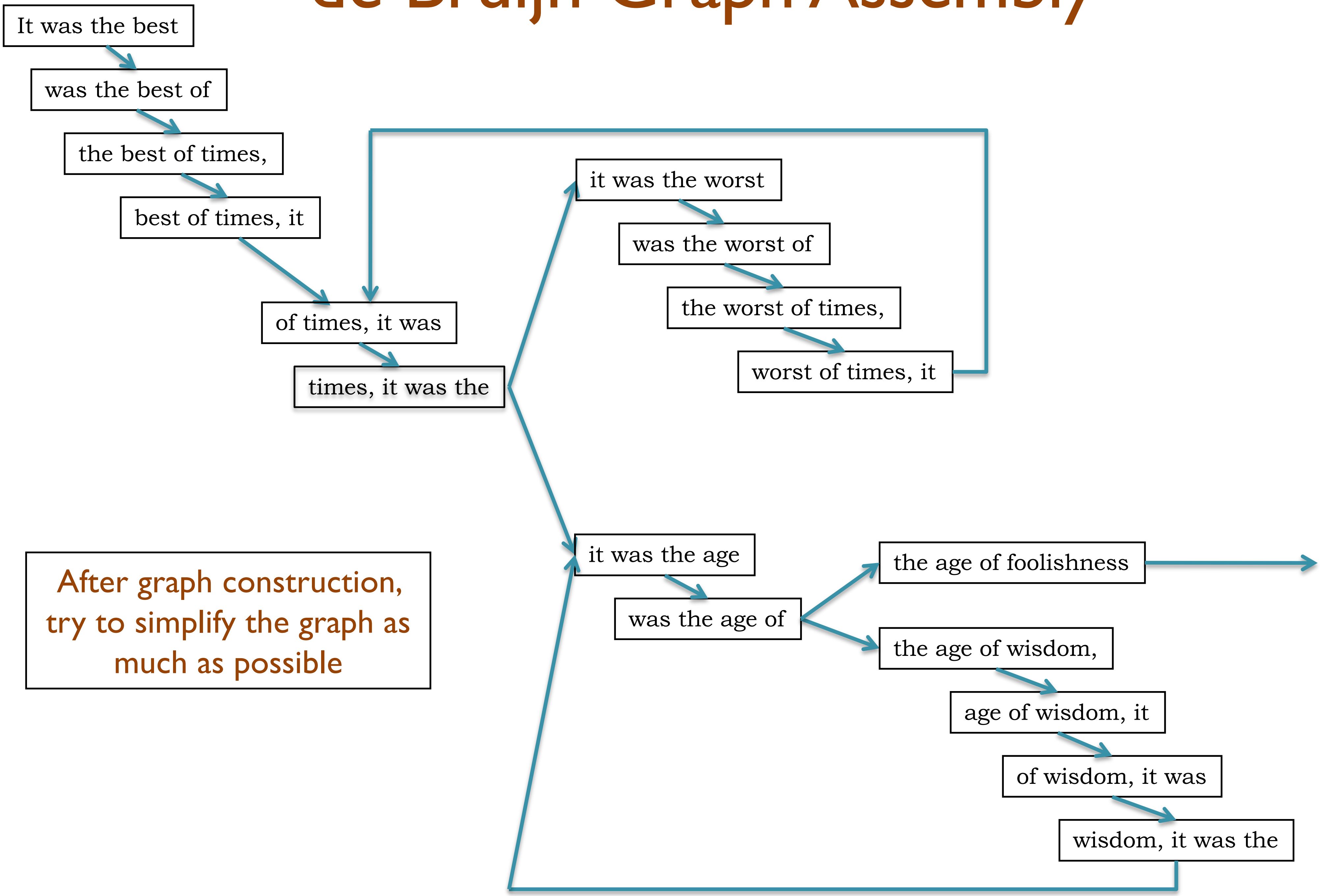
Directed Edge

It was the best → was the best of

- Locally constructed graph reveals the global sequence structure
  - Overlaps between sequences implicitly computed

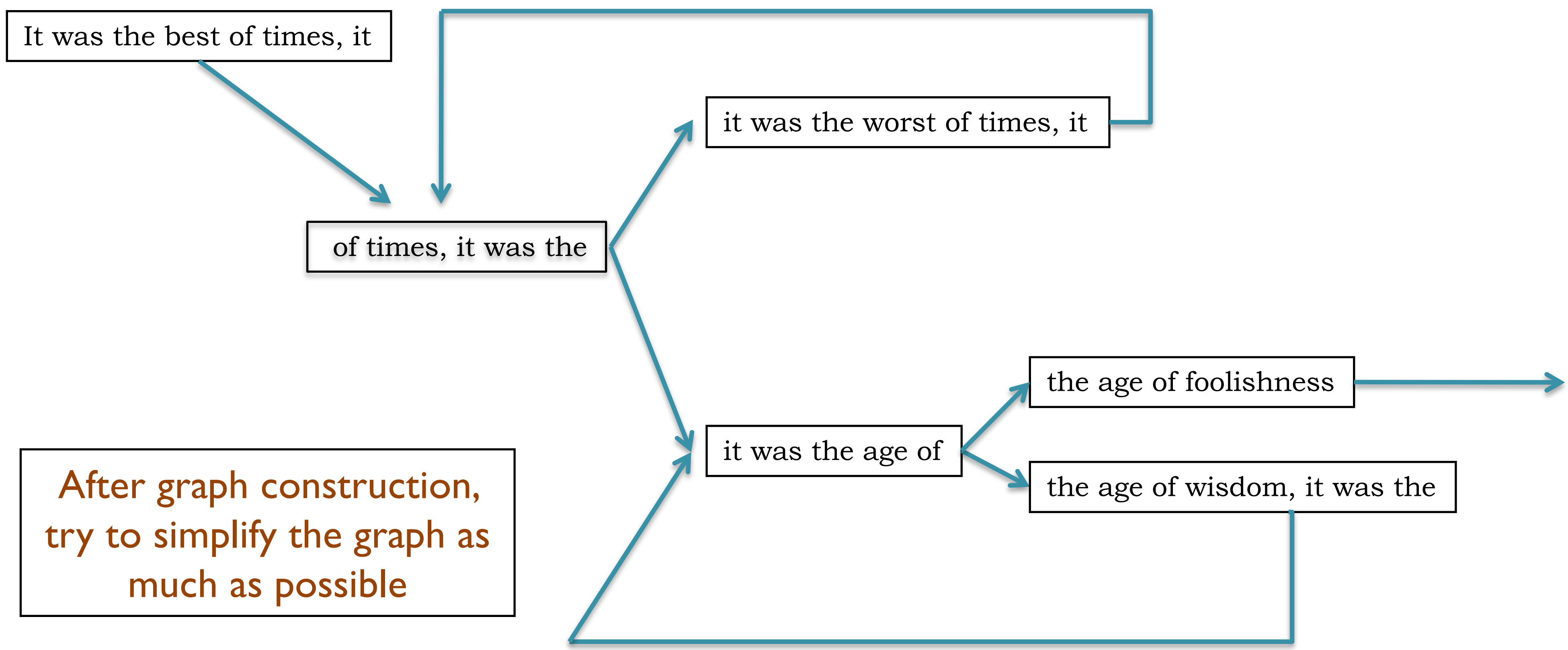
de Bruijn, 1946
Idury and Waterman, 1995
Pevzner, Tang, Waterman, 2001

de Bruijn Graph Assembly



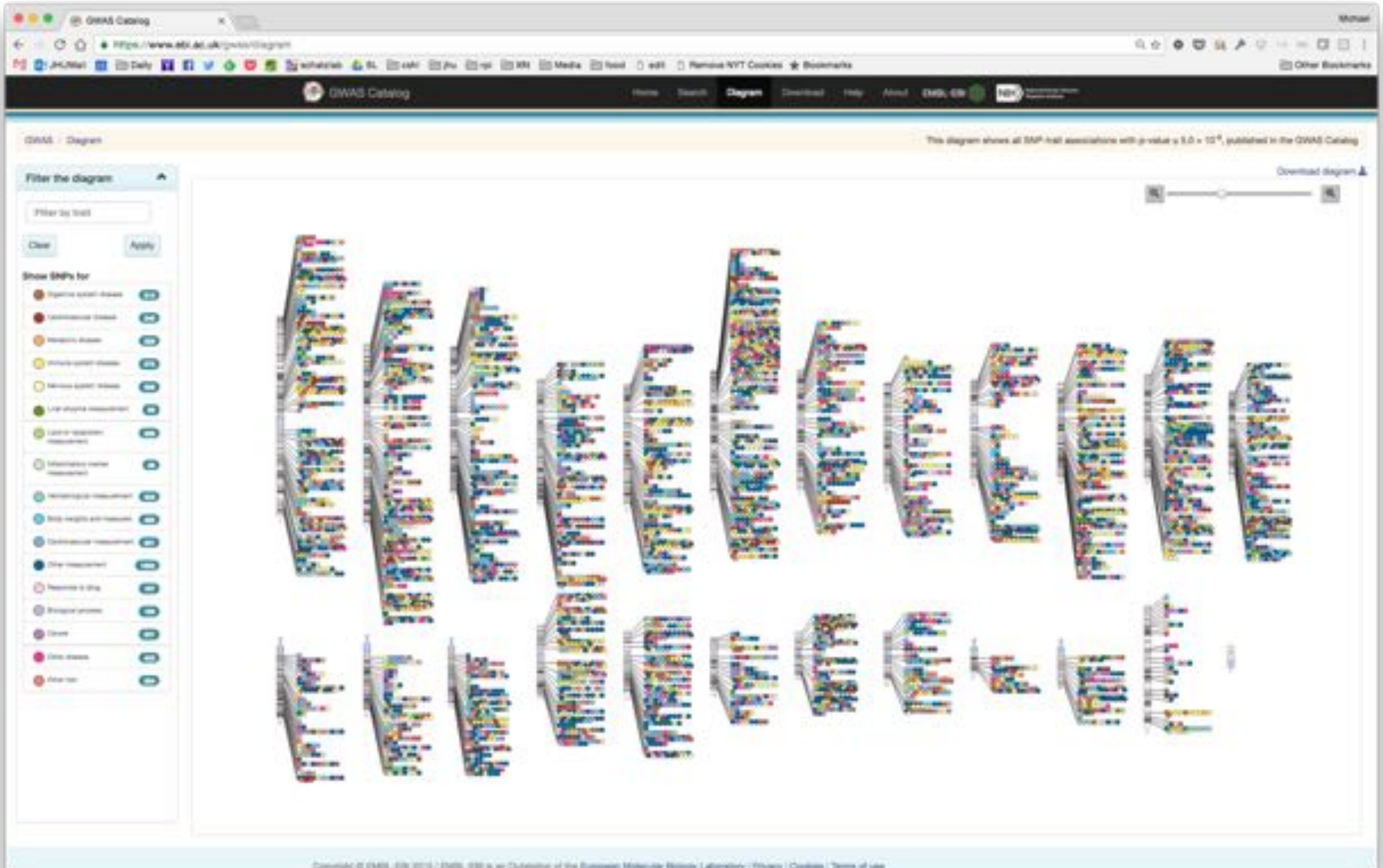
After graph construction, try to simplify the graph as much as possible

de Bruijn Graph Assembly



After graph construction, try to simplify the graph as much as possible

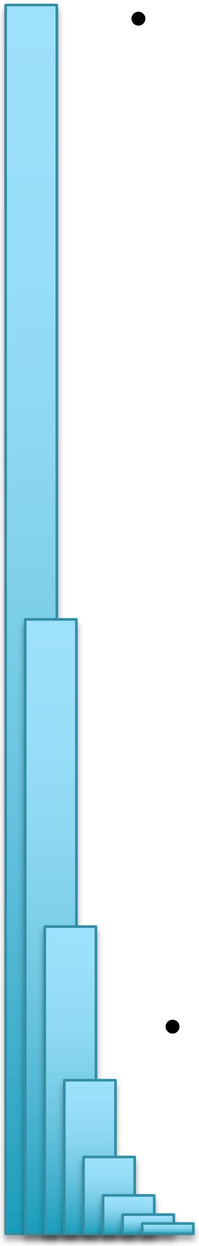
Genetic Associations



https://www.ebi.ac.uk/gwas/diagram

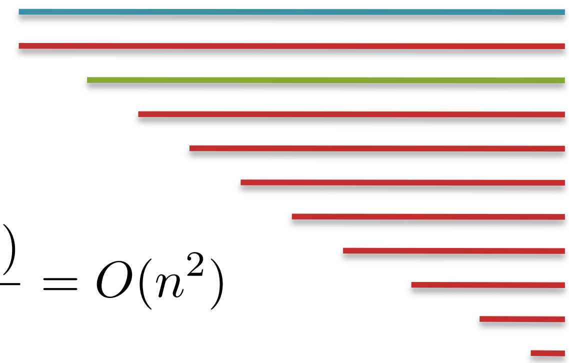
Part 2: Advanced Sorting

Suffix Array Construction



- How can we store the suffix array?

  [How many characters are in all suffixes combined?]



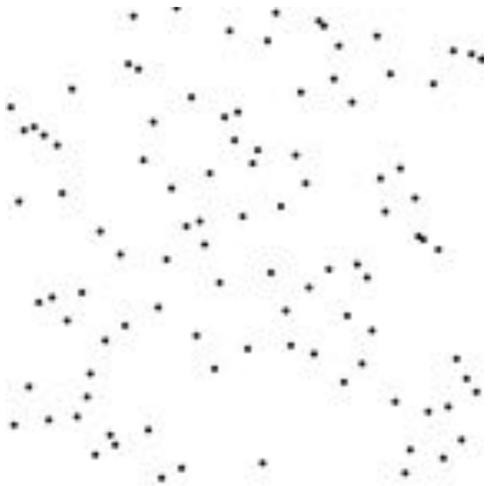
$$S = 1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

- Hopeless to explicitly store 4.5 billion billion characters
- Instead use implicit representation
  - Keep 1 copy of the genome, and a list of sorted offsets
  - Storing 3 billion offsets fits on a server (12GB)
- Searching the array is very fast, but it takes time to construct
  - This time will be amortized over many, many searches
  - Run it once "overnight" and save it away for all future queries

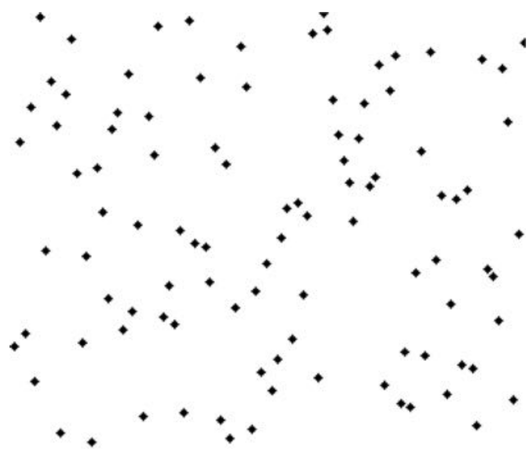
Pos
6
13
8
3
10
15
7
14
2
9
5
12
1
4
11

TGATTACAGATTACC

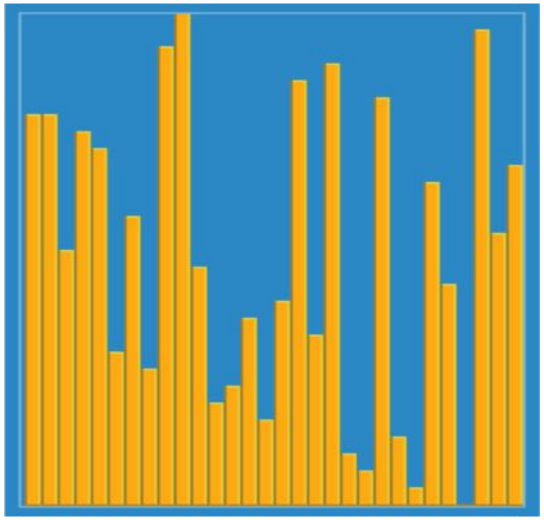
Quadratic Sorting Algorithms



Selection Sort
Move next smallest
into position



Bubble Sort
Swap up bigger
values over smaller



Insertion Sort
Slide next value into
correct position

Asymptotically all three have the same performance: O(n^2)

Can you do any better than this?

Monkey Sort

```
MonkeySort(int [] input)
   foreach p in allPermutations(input)
      if p is correctly sorted
         return p
```

😨 ***O(n!)***

☺ ***O(n)***

```
$ tail -f heights.16.log
tree[3393804000000]: 3 10 16 4 6 1 11 8 2 13 15 12 9 14 7 5 maxheight: 7
tree[3393805000000]: 3 10 16 4 6 1 13 14 5 9 8 7 15 2 12 11 maxheight: 7
tree[3393806000000]: 3 10 16 4 6 1 5 12 15 8 13 9 14 2 11 7 maxheight: 7
tree[3393807000000]: 3 10 16 4 7 6 9 2 13 15 5 12 11 14 8 1 maxheight: 6
tree[3393808000000]: 3 10 16 4 7 6 12 8 5 11 9 13 15 1 2 14 maxheight: 7
tree[3393809000000]: 3 10 16 4 7 6 14 15 2 5 11 8 9 12 13 1 maxheight: 7
tree[3393810000000]: 3 10 16 4 7 5 6 13 9 2 11 12 8 14 15 1 maxheight: 6
tree[3393811000000]: 3 10 16 4 7 5 2 6 15 12 1 13 9 11 8 14 maxheight: 7
tree[3393812000000]: 3 10 16 4 7 5 13 8 6 15 2 11 14 12 9 1 maxheight: 6
tree[3393813000000]: 3 10 16 4 7 5 15 6 12 11 8 9 13 14 2 1 maxheight: 7
...
```

Considering that there are n! possible permutations, maybe we should be happy with $O(n^2)$ time

Nah!

Heap Sort

```
HeapSort(int [] input)
   h  = new MaxHeap()
   foreach item in input
      h.add(item)

   output = new int[]
   while(!h.empty())
      output.add(h.max())
```

☺ O(n lg n)

☺ O(n lg n)

Improved selection sort:

Divide the input into sorted and unsorted regions, then iteratively shrink the unsorted region by extracting the smallest/largest element. Use a heap rather than a linear-time search to find the min/max in O(lg n) time instead of O(n) time.

Yields an overall O(n lg n) runtime ☺

Invented by J. W. J. Williams in 1964

In Place Heap Sort (MaxHeap)

Original Array	8	6	7	5	3	0	9
Heapify	8	6	7	5	3	0	9
Upheap to fix	8	6	9	5	3	0	7
Upheap to fix	9	6	8	5	3	0	7

Convert unsorted array to heap without any extra space in O(n)

In Place Heap Sort (MaxHeap)

Starting heap	9	6	8	5	3	0	7
Swap/remove	7	6	8	5	3	0	9
Sift down	8	6	7	5	3	0	9
Swap/remove	0	6	7	5	3	8	9

In Place Heap Sort (MaxHeap)

Swap/remove

0	6	7	5	3	8	9

Sift down

7	6	0	5	3	8	9

Swap/remove

3	6	0	5	7	8	9

…

Final

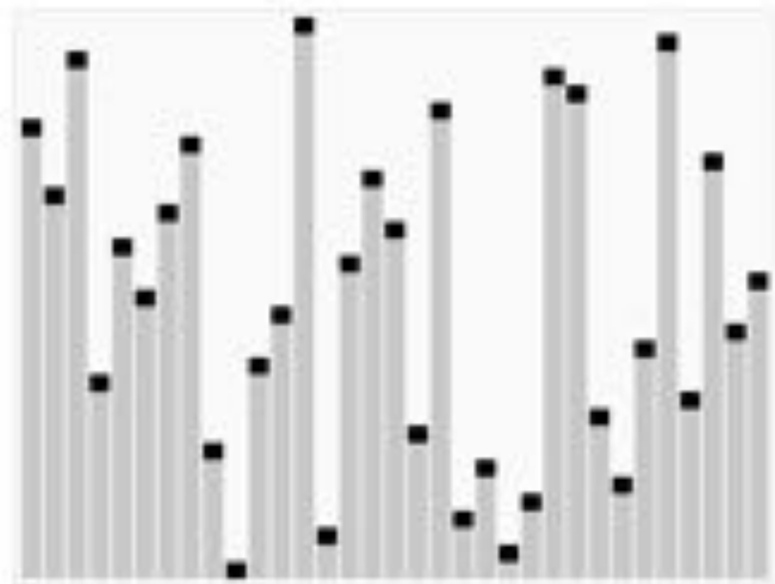
0	3	5	6	7	8	9

☺ O(n lg n) => Provably optimal performance

Heap Sort

```
HeapSort(int [] input)
   h  = new MaxHeap()
   foreach item in input
      h.add(item)

   output = new int[]
   while(!h.empty())
      output.add(h.max())
```



In-place algorithm, worst-case O(n log n) runtime.

Fast running time and constant upper bound on its auxiliary storage
⇒ Embedded systems with real-time constraints or systems concerned with security like the linux kernel often use heapsort

While "provably optimal" often outperformed by alternative algorithms on real world data sets

Merge Sort

Uses the powerful divide-and-conquer recursive strategy

Key idea: Merging two sorted lists into a new sorted list is easy

Merge these two lists:

List A: 6, 7, 8   List B: 0, 3, 5, 9

Merge Sort

Uses the powerful divide-and-conquer recursive strategy

Key idea: Merging two sorted lists into a new sorted list is easy

Merge these two lists:

List A: 6, 7, 8   List B: 0, 3, 5, 9

List C:

Merge Sort

Uses the powerful divide-and-conquer recursive strategy

Key idea: Merging two sorted lists into a new sorted list is easy

Merge these two lists:

List A: 6, 7, 8   List B: 0, 3, 5, 9

List C: 0

Merge Sort

Uses the powerful divide-and-conquer recursive strategy

Key idea: Merging two sorted lists into a new sorted list is easy

Merge these two lists:

List A: 6, 7, 8 List B: 0, 3, 5, 9

List C: 0,3

Merge Sort

Uses the powerful divide-and-conquer recursive strategy

Key idea: Merging two sorted lists into a new sorted list is easy

Merge these two lists:

List A: 6, 7, 8   List B: 0, 3, 5, 9

List C: 0,3,5

Merge Sort

Uses the powerful divide-and-conquer recursive strategy

Key idea: Merging two sorted lists into a new sorted list is easy

Merge these two lists:

List A: 6, 7, 8 List B: 0, 3, 5, 9

List C: 0,3,5,6

Merge Sort

Uses the powerful divide-and-conquer recursive strategy

Key idea: Merging two sorted lists into a new sorted list is easy

Merge these two lists:

List A: 6, 7, 8   List B: 0, 3, 5, 9

List C: 0,3,5,6,7

Merge Sort

Uses the powerful divide-and-conquer recursive strategy

Key idea: Merging two sorted lists into a new sorted list is easy

Merge these two lists:

List A: 6, 7, 8   List B: 0, 3, 5, 9

List C: 0,3,5,6,7,8

Merge Sort

Uses the powerful divide-and-conquer recursive strategy

Key idea: Merging two sorted lists into a new sorted list is easy

Merge these two lists:

List A: 6, 7, 8   List B: 0, 3, 5, 9

List C: 0,3,5,6,7,8,9

Merge two sorted lists in linear time
O(sum of the length of the individual lists) ☺

Where do these sorted lists come from?

Merge Sort

Uses the powerful **divide**-and-conquer recursive strategy

Original Array	8	6	7	5	3	0	9
First split	8	6	7	5	3	0	9
Second split	8	6	7	5	3	0	9
Third split	8	6	7	5	3	0	9

How many times can we split an array of length n?

After O(lg n) splits, have n lists each 1 element long
that are each trivially sorted

In practice, just start with n lists of 1 element ☺

Merge Sort

Uses the powerful divide-and-**conquer** recursive strategy

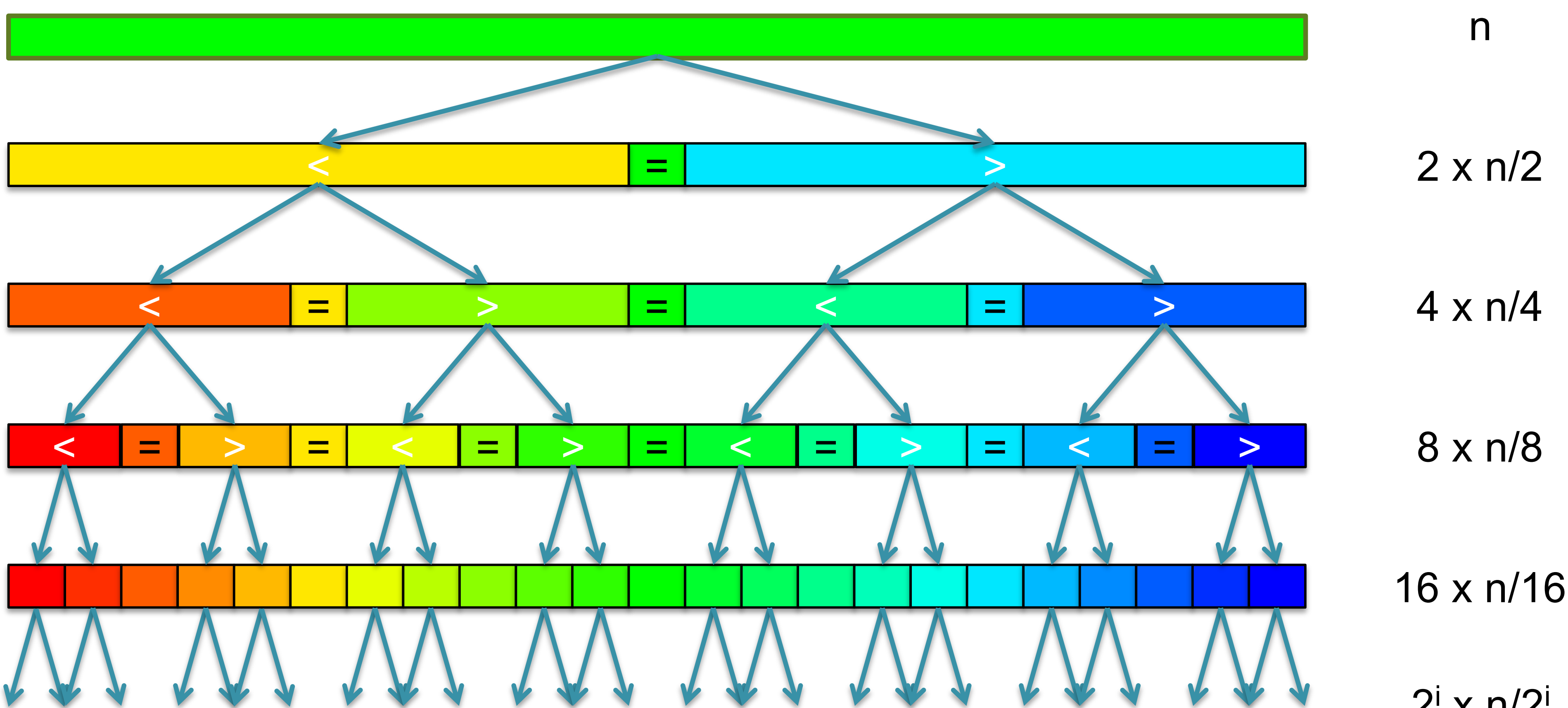
Leaf nodes	8	6	7	5	3	0	9
First merge	8	6	7	3	5	0	9
Second merge	6	7	8	0	3	5	9
Sorted array	0	3	5	6	7	8	9

After O(lg n) merges the entire array will be sorted

Very popular external sorting algorithm (huge data sets on disk) but less popular because it requires O(n) extra memory

Quicksort

- Selection sort is slow because it rescans the entire list for each element
  - How can we split up the unsorted list into independent ranges?
  - Hint 1: Binary search splits up the problem into 2 independent ranges (hi/lo)
  - Hint 2: Assume we know the median value of a list



[How many times can we split a list in half?]

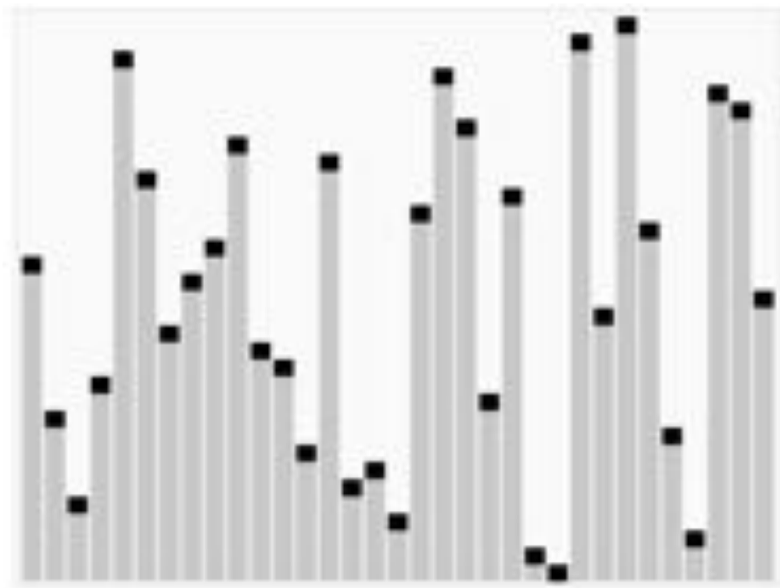
QuickSort Analysis

- QuickSort(Input: list of n numbers)

```
// see if we can quit
if (length(list)) <= 1): return list

// split list into lo & hi
pivot = median(list)
lo = {}; hi = {};
for (i = 1 to length(list))
    if (list[i] < pivot): append(lo, list[i])
    else:                 append(hi, list[i])

// recurse on sublists
return (append(QuickSort(lo), QuickSort(hi))
```



http://en.wikipedia.org/wiki/Quicksort

- Analysis (Assume we can find the median in O(n))

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

$$T(n) = n + 2(\frac{n}{2}) + 4(\frac{n}{4}) + \cdots + n(\frac{n}{n}) = \sum_{i=0}^{lg(n)} \frac{2^i n}{2^i} = \sum_{i=0}^{lg(n)} n = O(n \lg n)$$

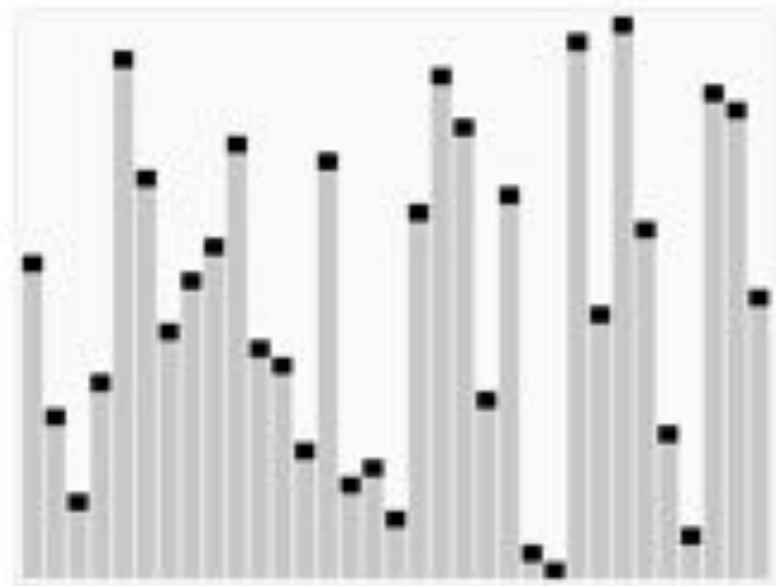
QuickSort Analysis

- QuickSort(Input: list of n numbers)

```
// see if we can quit
if (length(list)) <= 1): return list

// split list into lo & hi
pivot = median(list)
lo = {}; hi = {};
for (i = 1 to length(list))
        if (list[i] < pivot): append(lo, list[i])
        else:                 append(hi, list[i])

// recurse on sublists
return (append(QuickSort(lo), QuickSort(hi))
```



http://en.wikipedia.org/wiki/Quicksort

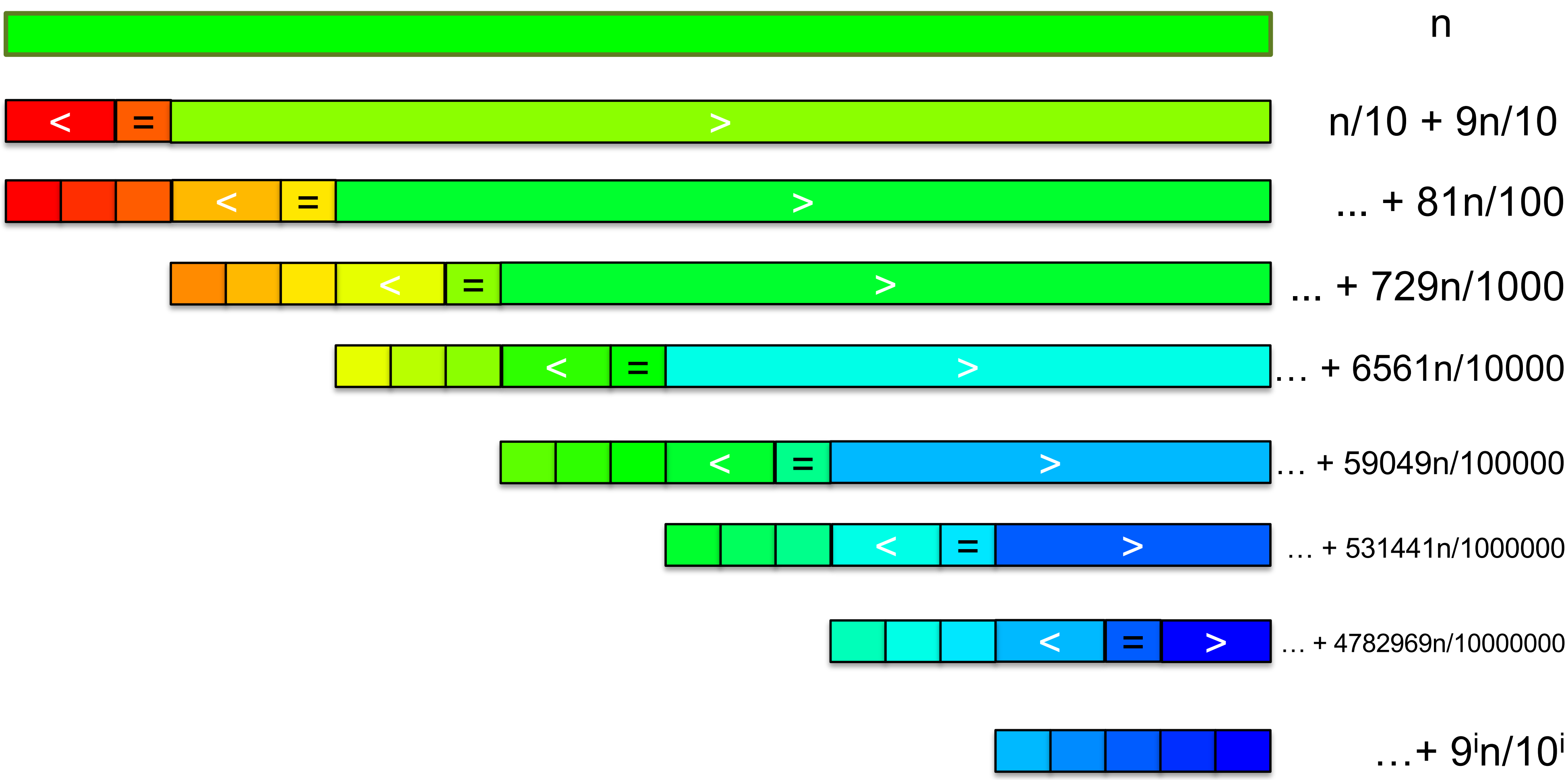
- Analysis (Assume we can find the median in O(n))

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

$$T(n) = n + 2(\frac{n}{2}) + 4(\frac{n}{4}) + \cdots + n(\frac{n}{n}) = \sum_{i=0}^{lg(n)} \frac{2^i n}{2^i} = \sum_{i=0}^{lg(n)} n = O(n \lg n)$$

Picking the Median

- What if we miss the median and do a 90/10 split instead?



[How many times can we cut 10% off a list?]

Randomized Quicksort

- 90/10 split runtime analysis

$$T(n) = n + T(\frac{n}{10}) + T(\frac{9n}{10})$$

$$T(n) = n + \frac{n}{10} + T(\frac{n}{100}) + T(\frac{9n}{100}) + \frac{9n}{10} + T(\frac{9n}{100}) + T(\frac{81n}{100})$$

$$T(n) = n + n + T(\frac{n}{100}) + 2T(\frac{9n}{100}) + T(\frac{81n}{100})$$

$$T(n) = \sum_{i=0}^{\log_{10/9}(n)} n = O(n \lg n)$$

Find smallest x s.t.

$$(9/10)^x n \leq 1$$

$$(10/9)^x \geq n$$

$$x \geq \log_{10/9} n$$

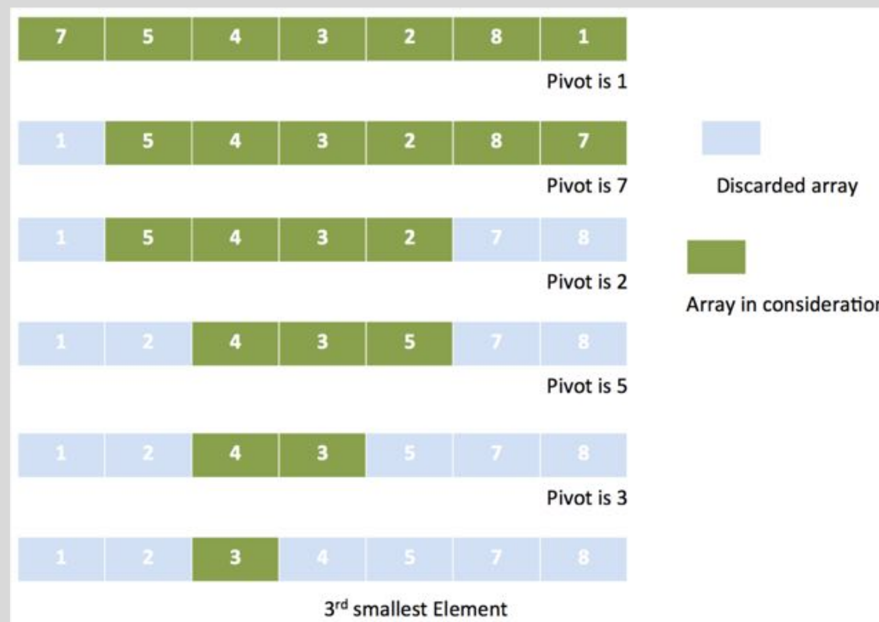
- If we randomly pick a pivot, we will get at least a 90/10 split with very high probability
  - If picking randomly is expensive, choose the median of 3 items, or median on N items (Ninther) or find overall median without sorting
    - The more time spent picking the pivot, the higher the constants will be
  - Everything is okay as long as we always slice off a fraction of the list

[Challenge Question: What happens if we slice 1 element]

QuickSelect

How can we find the median in O(n)?

```
// Returns the k-th smallest element of list within left..right inclusive
// partition() arranges the data between left and right if it is less than or greater than the pivot
function select(list, left, right, k)
    if left == right        // If the list contains only one element,
        return list[left]   // return that element
    pivotIndex  := ...      // select a pivotIndex between left and right,
                            // e.g., left + floor(rand() % (right - left + 1))
    pivotIndex  := partition(list, left, right, pivotIndex)
    // The pivot is in its final sorted position
    if k == pivotIndex
        return list[k]
    else if k < pivotIndex
        return select(list, left, pivotIndex - 1, k)
    else
        return select(list, pivotIndex + 1, right, k)
```



Finds median in O(n) expected time

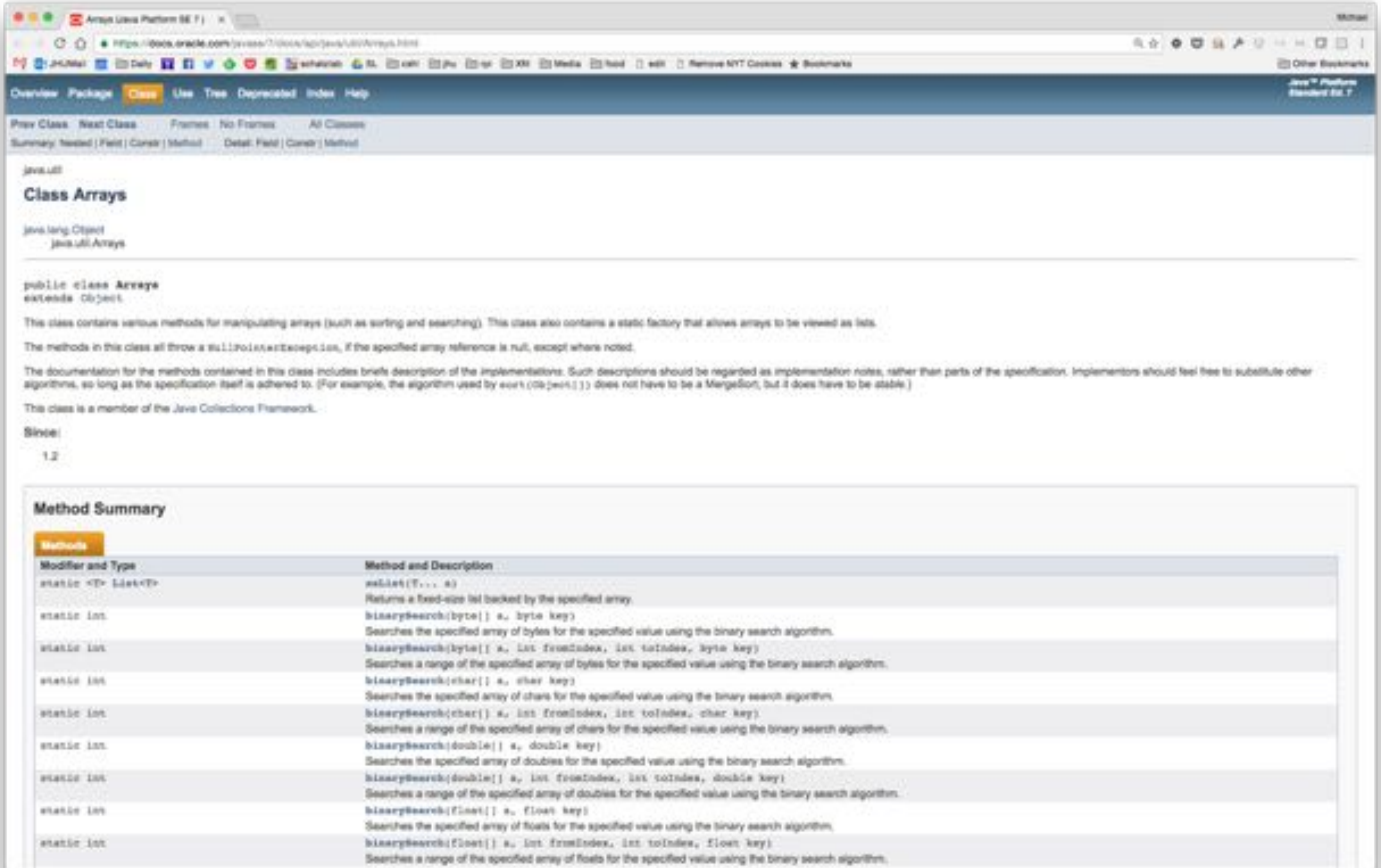
QuickSort in Java

`Arrays.sort()`

The goal of software engineering is to build libraries of correct reusable functions that implement higher level ideas

- Build complex software out of simple components
- Software tends to be 90% plumbing, 10% novel work
- You still need to know how they work
  - Java requires an explicit representation of the strings

java.util.Arrays



java.util

Class Arrays

java.lang.Object
java.util.Arrays

```
public class Arrays
extends Object
```

This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

The methods in this class all throw a `NullPointerException`, if the specified array reference is null, except where noted.

The documentation for the methods contained in this class includes briefs description of the implementations. Such descriptions should be regarded as implementation notes, rather than parts of the specification. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort(Object[])` does not have to be a MergeSort, but it does have to be stable.)

This class is a member of the Java Collections Framework.

Since:

1.2

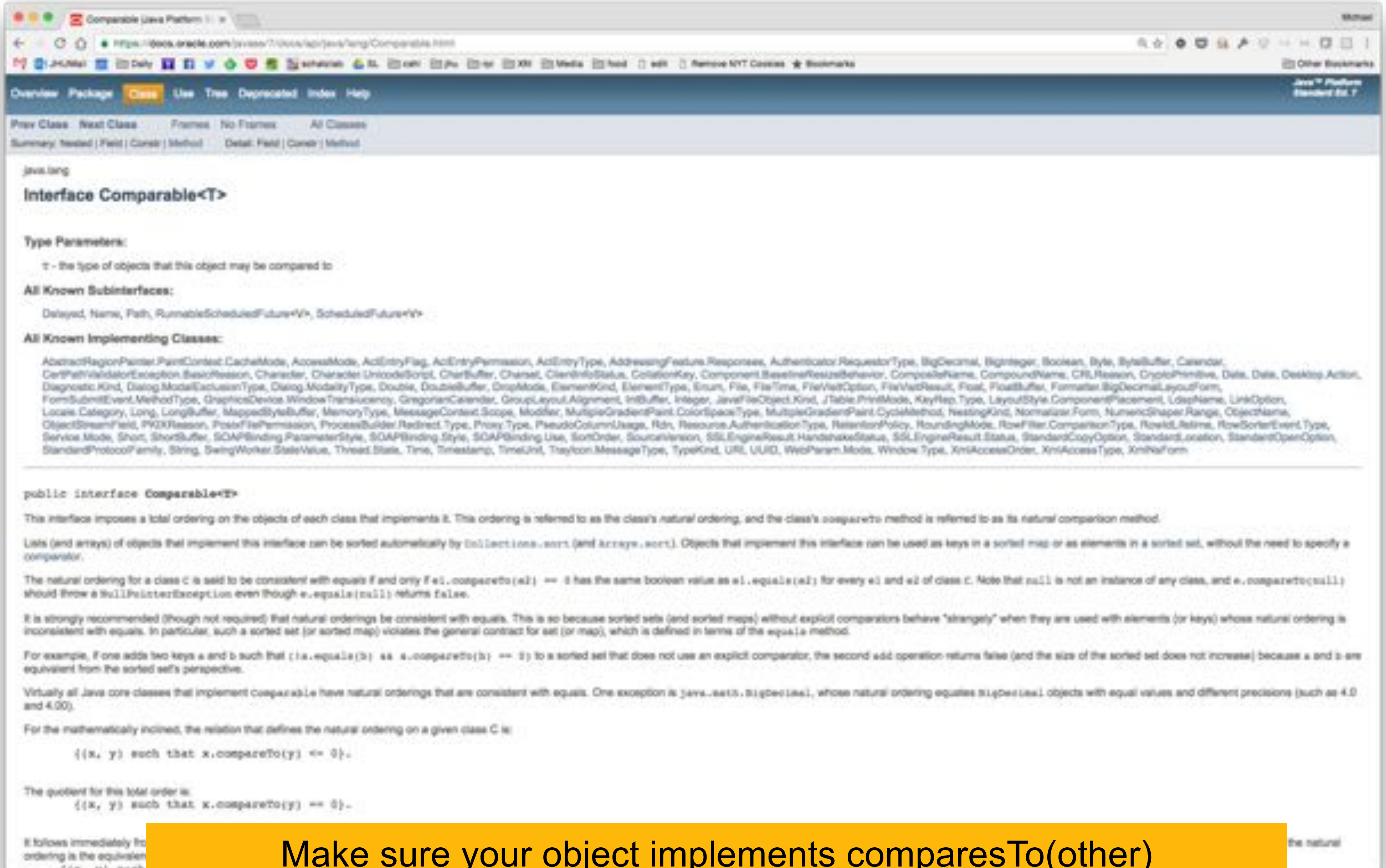
Method Summary

Methods

Modifier and Type	Method and Description
`static <T> List<T>`	`asList(T... a)` Returns a fixed-size list backed by the specified array.
`static int`	`binarySearch(byte[] a, byte key)` Searches the specified array of bytes for the specified value using the binary search algorithm.
`static int`	`binarySearch(byte[] a, int fromIndex, int toIndex, byte key)` Searches a range of the specified array of bytes for the specified value using the binary search algorithm.
`static int`	`binarySearch(char[] a, char key)` Searches the specified array of chars for the specified value using the binary search algorithm.
`static int`	`binarySearch(char[] a, int fromIndex, int toIndex, char key)` Searches a range of the specified array of chars for the specified value using the binary search algorithm.
`static int`	`binarySearch(double[] a, double key)` Searches the specified array of doubles for the specified value using the binary search algorithm.
`static int`	`binarySearch(double[] a, int fromIndex, int toIndex, double key)` Searches a range of the specified array of doubles for the specified value using the binary search algorithm.
`static int`	`binarySearch(float[] a, float key)` Searches the specified array of floats for the specified value using the binary search algorithm.
`static int`	`binarySearch(float[] a, int fromIndex, int toIndex, float key)` Searches a range of the specified array of floats for the specified value using the binary search algorithm.

Fast sorting for objects that implement the Comparable interface

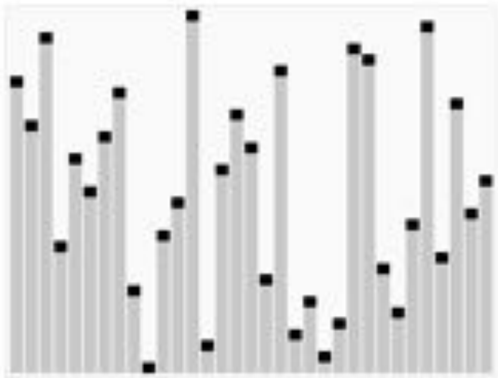
java.lang Interface Comparable<T>

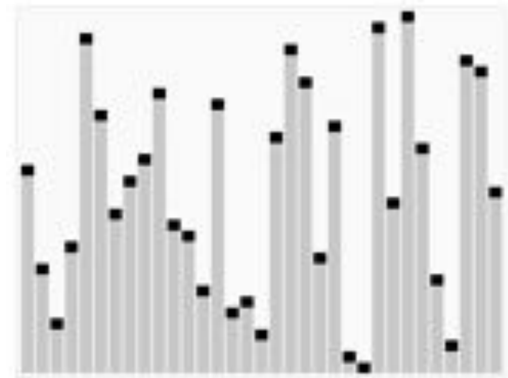


Make sure your object implements comparesTo(other)

<0: Im less than other; 0: Im equal to other; >0: Im greater than other

Advance Sorting Review



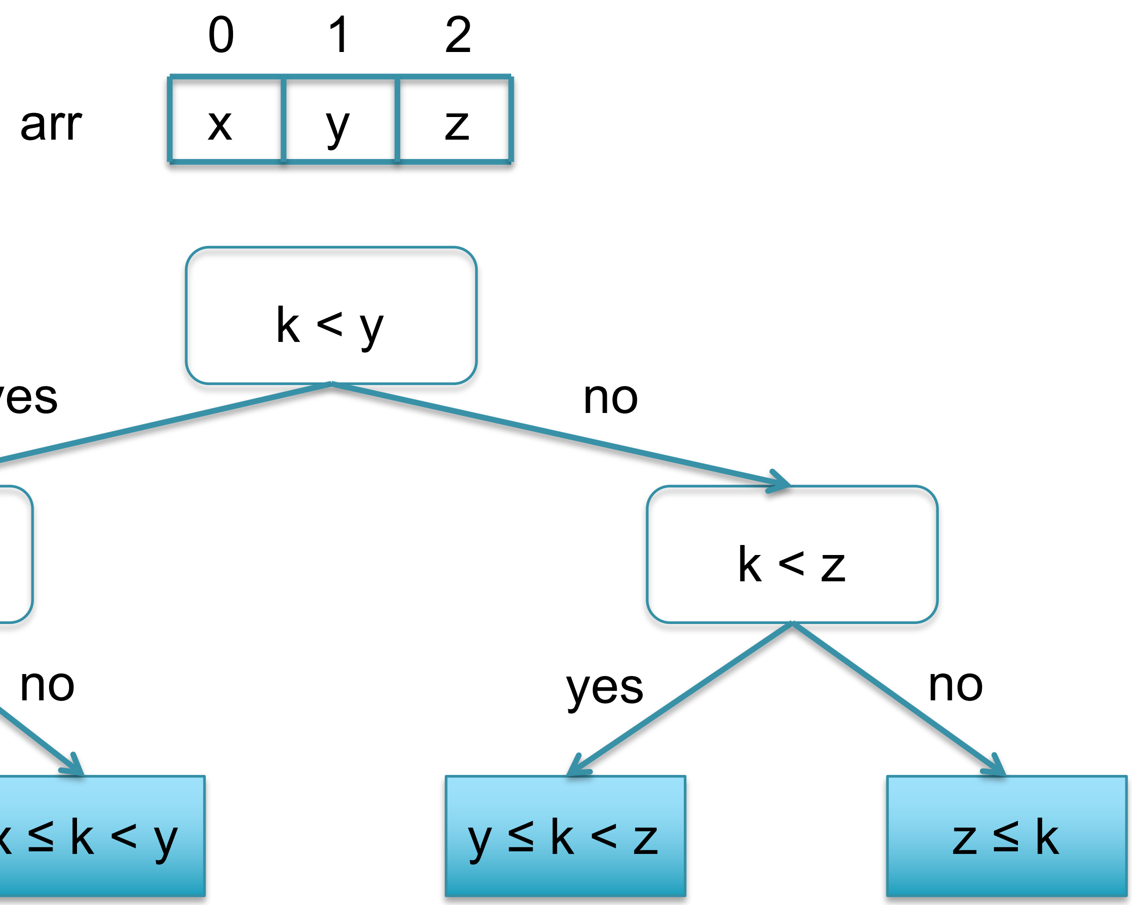


Heap Sort	Merge Sort	QuickSort
Add everything to a heap, remove from biggest to smallest	Divide input into n lists, merge pairs of sorted lists as a tree	Recursively partition input into low/high based on a pivot
O(n lg n) worst case	O(n lg n) worst case	$O(n^2)$ worse case, O(n lg n) typical
Big constants	O(n) space overhead	Very fast in practice

Part 3: Really Advanced Sorting

Decision Tree of Searching

How many comparisons are made to binary search in an array with 3 items?



The decision tree encodes the execution over all possible input values
The decision tree is a binary tree, each node encodes exactly 1 comparison
The decision tree for searching has n+1 leaf nodes (since there are n+1 "slots" for k)

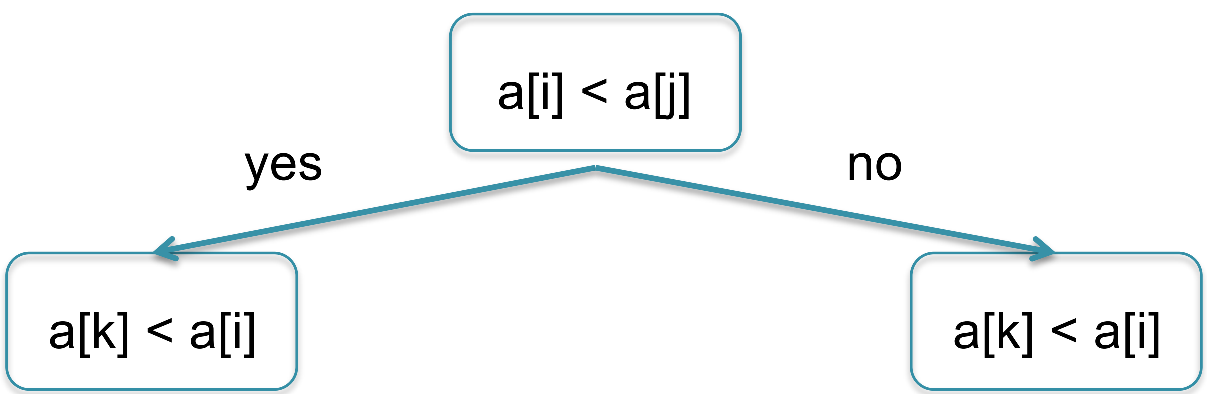
Searching by comparisons requires at least O(lg n) comparisons (lower bound)

Decision Tree of Sorting

Notice that sorting 3 items (a,b,c) may have 3! = 6 possible permutations

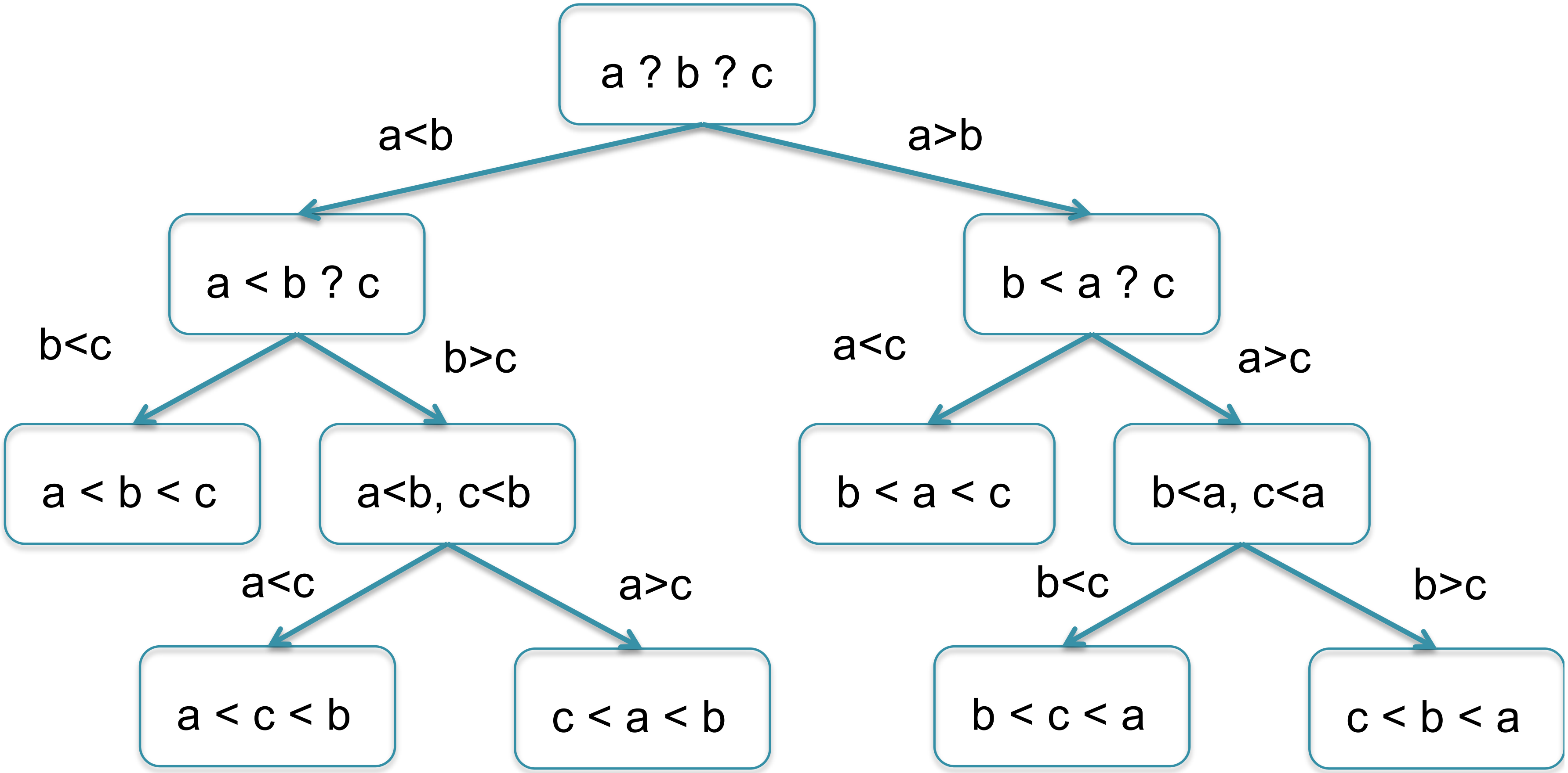
a < b < c
a < c < b
b < a < c
b < c < a
c < a < b
c < b < a

Initially we don't know which one of these permutations is the correctly sorted version, but we can make pairwise comparisons to figure it out

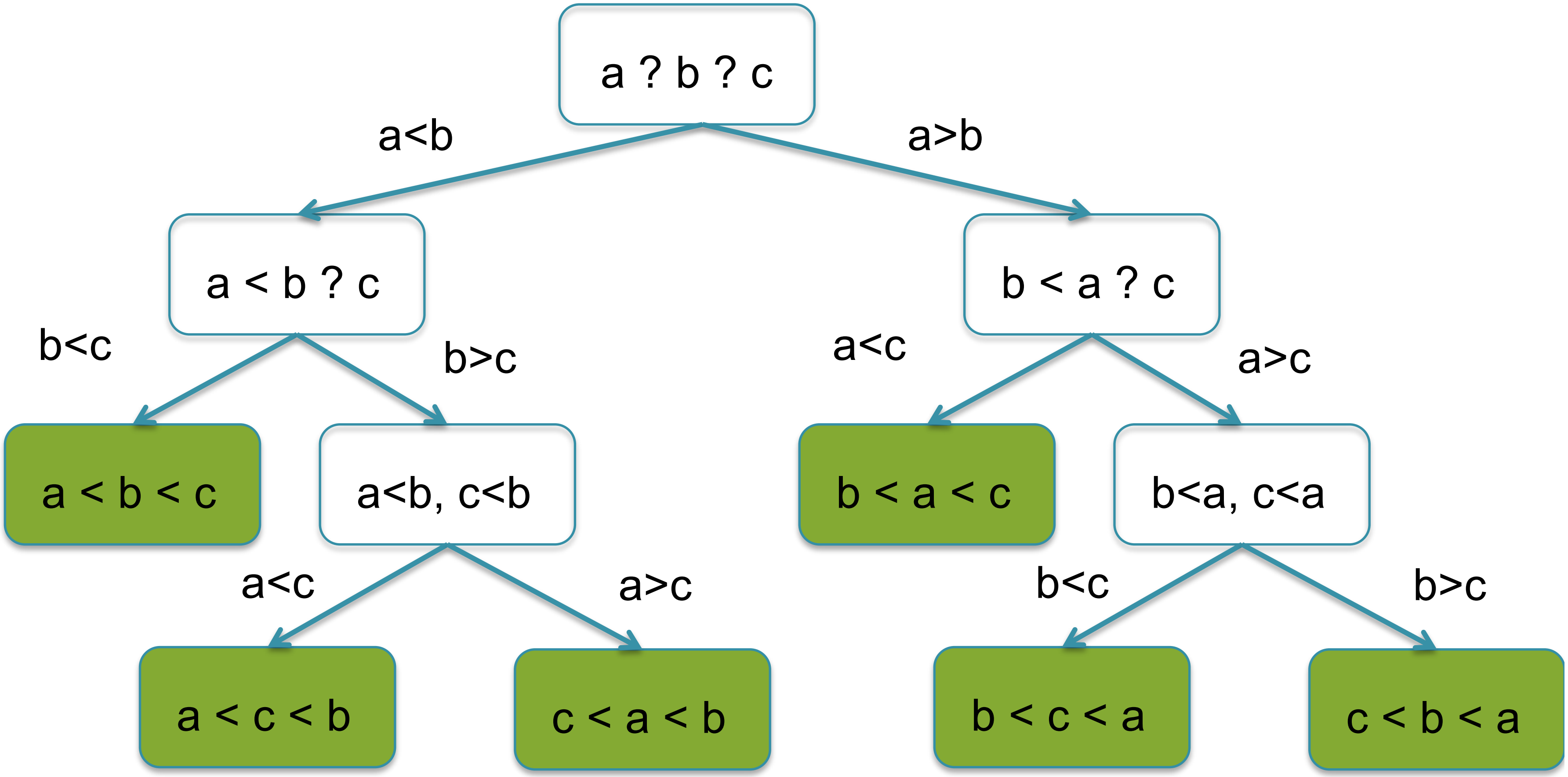


In the worst case, how many comparisons are needed?

Decision Tree of Sorting



Decision Tree of Sorting



Notice that we have all 3! = 6 possibilities as leaf nodes

How tall is a tree with n! leaf nodes?

O(lg n!)

whattt???

Decision Tree of Sorting

What is O(lg n!)

lg(n!) = lg(n * (n-1) * (n-2) * … * 2 * 1)

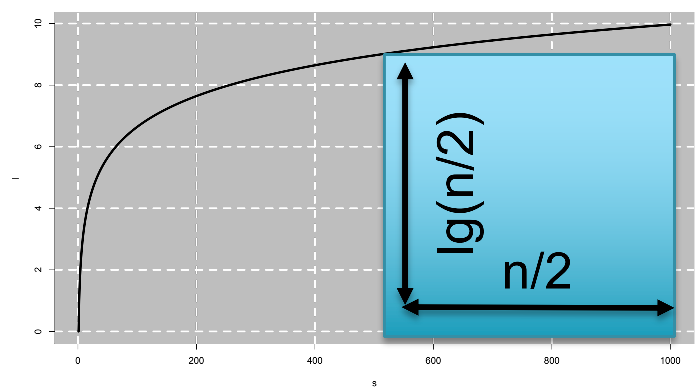
lg(n!) = lg(n) + lg(n-1) + lg(n-2) + … + lg(2) + lg(1)

lg(n!) ≤ lg(n) + lg(n) + lg(n) + … + lg(n) + lg(n)

lg(n!) ≤ n lg n

This is true, but the wrong direction to prove a lower bound.
Need to show lg(n!) ≥ something instead

$$
\begin{aligned}
&= \log n + \log(n-1) + \log(n-2) + \cdots + \log 2 \\
&= \sum_{i=2}^{n} \log i \\
&= \sum_{i=2}^{n/2-1} \log i + \sum_{i=n/2}^{n} \log i \\
&\geq 0 + \sum_{i=n/2}^{n} \log \frac{n}{2} \\
&= \frac{n}{2} \cdot \log \frac{n}{2} \\
&= \Omega(n \log n)
\end{aligned}
$$



Because lg grows so slowly,
Just sum from n/2 to n

***Any* comparison based sorting algorithm requires at least O(n lg n)**

Next Steps

1. Reflect on the magic and power of Sorting!

1. Assignment 9 due on Friday November 30 @ 10pm