Sorting, Searching, & Aligning

Michael Schatz

Bioinformatics Lecture 1
Quantitative Biology 2012

Short Read Applications

- Genotyping: Identify Variations

```
                                                GGTATAC…
…CCATAG        TATGCGCCC      CGGAAATTT   CGGTATAC
…CCAT      CTATATGCG          TCGGAAATT    CGGTATAC
…CCAT   GGCTATATG         CTATCGGAAA      GCGGTATA
…CCA  AGGCTATAT          CCTATCGGA      TTGCGGTA    C…
…CCA  AGGCTATAT      GCCCTATCG        TTTGCGGT      C…
…CC    AGGCTATAT     GCCCTATCG   AAATTTGC       ATAC…
…CC  TAGGCTATA   GCGCCCTA        AAATTTGC   GTATAC…
…CCATAGGCTATATGCGCCCTATCGGCAATTTGCGGTATAC…
```

- *-seq: Classify & measure significant peaks

```
                    GAAATTTGC
                  GGAAATTTG
               CGGAAATTT
               CGGAAATTT
              TCGGAAATT
         CTATCGGAAA
       CCTATCGGA     TTTGCGGT
    GCCCTATCG   AAATTTGC
…CC GCCCTATCG   AAATTTGC      ATAC…
…CCATAGGCTATATGCGCCCTATCGGCAATTTGCGGTATAC…
```

Short Read Alignment

- Given a reference and a set of reads, report at least one "good" local alignment for each read if one exists
  - Approximate answer to: where in genome did read originate?

- What is "good"? For now, we concentrate on:
  - Fewer mismatches is better
  - Failing to align a low-quality base is better than failing to align a high-quality base

```
...TGATCATA...   better than   ...TGATCATA...
   ||||||                         ||  ||
   GATCAA                         GAGAAT

...TGATATTA...   better than   ...TGATcaTA...
   |||  |                        |  |||
   GATcaT                         GTACAT
```

Exact Matching Review & Overview

Where is GATTACA in the human genome?

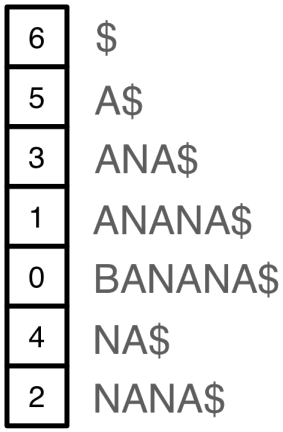
Brute Force (3 GB)

```
BANANA
BAN
  ANA
    NAN
      ANA
```

Naive

Slow & Easy

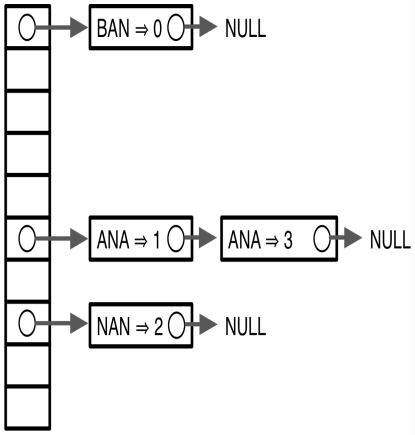
Suffix Array (>15 GB)



Vmatch, PacBio Aligner

Binary Search

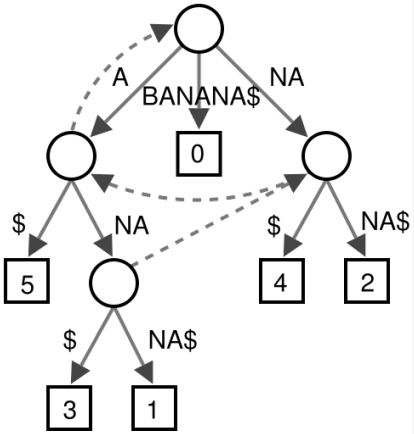
Hash Table (>15 GB)



BLAST, MAQ, ZOOM, RMAP, CloudBurst

Seed-and-extend

Suffix Tree (>51 GB)



MUMmer, MUMmerGPU

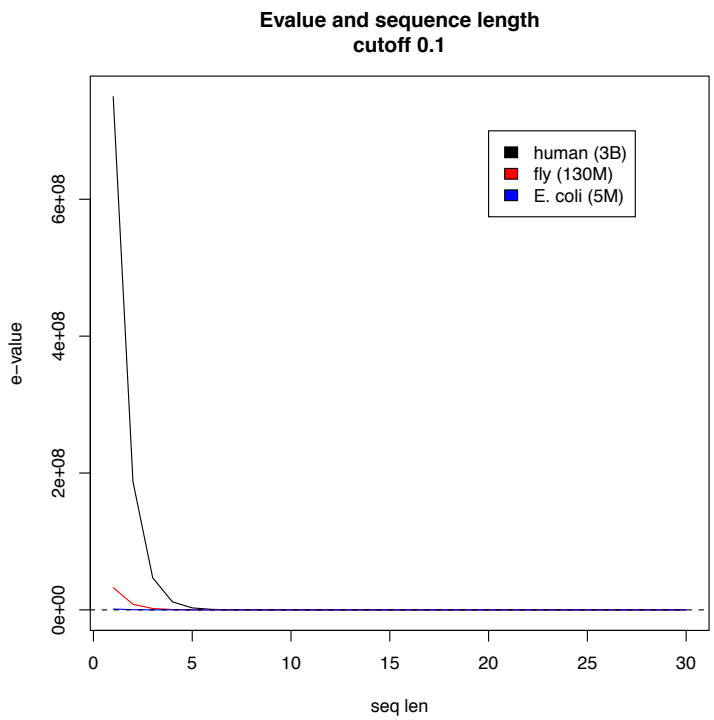
Tree Searching

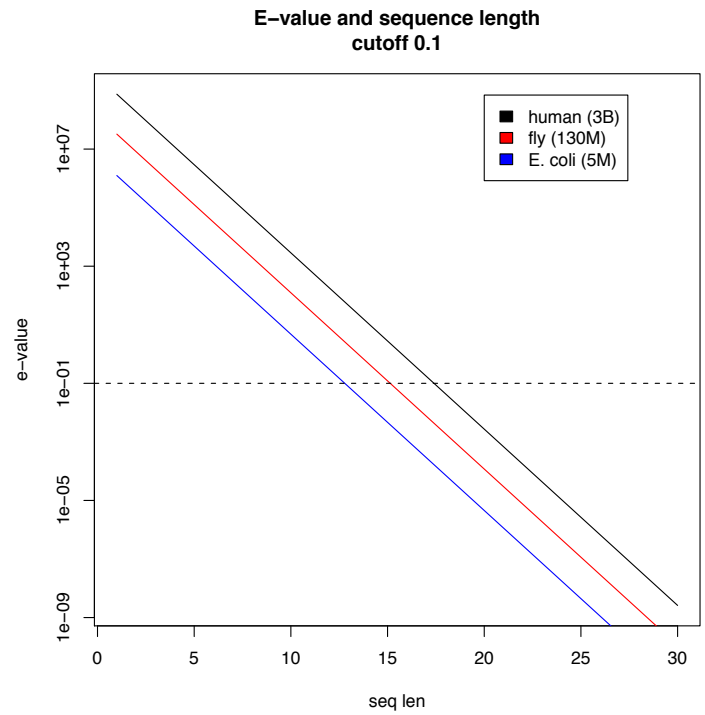
*** These are general techniques applicable to any search problem ***

Expected Occurrences

The expected number of occurrences (e-value) of a given sequence in a genome depends on the length of the genome and inversely on the length of the sequence

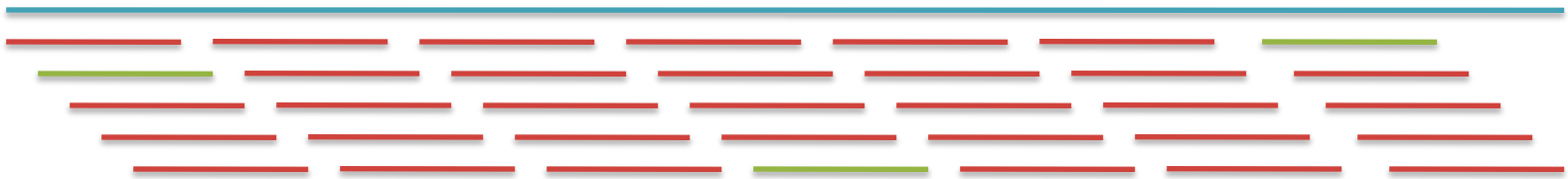
- 1 in 4 bases are G, 1 in 16 positions are GA, 1 in 64 positions are GAT
- 1 in 16,384 should be GATTACA
- $E=(n-m+1)/(4^m)$ [183,105 expected occurrences]





[Challenge Question: What is the expected distribution & variance?]

1. Brute Force



- Brute Force:
  - At every possible offset in the genome:
    - Do all of the characters of the query match?

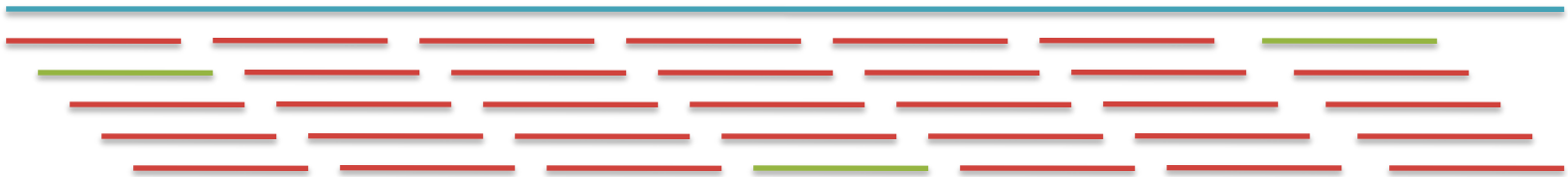
- Analysis
  - Simple, easy to understand
  - Genome length = n [3B]
  - Query length = m [7]
  - Comparisons: (n-m+1) * m [21B]

- Overall runtime: O(nm)

[How long would it take if we double the genome size, read length?]

[How long would it take if we double both?]

Brute Force in Matlab

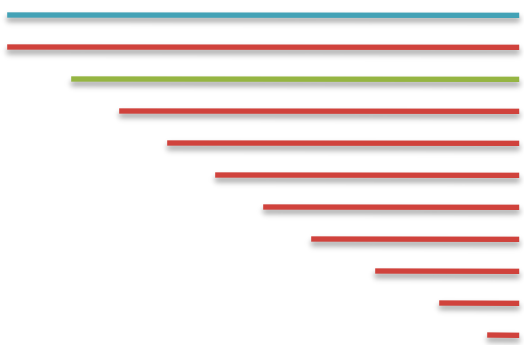


```
query  = 'GATTACA';
genome = 'TGATTACAGATTACC';

nummatches=0;

% At every possible offset
for offset=1:length(genome)-length(query)+1
    % Do all of the characters match?
    if (genome(offset:offset+length(query)-1) == query)
          disp(['Match at offset ', num2str(offset)])
          nummatches = nummatches+1;
    else
          %Uncomment to see every non-match
          %disp(['No match at offset ', num2str(offset)])
    end
end

disp(['Found ', num2str(nummatches),' matches of ', query, ' in genome of length ',
    num2str(length(genome))])


disp(['Expected number of occurrences: ', num2str((length(genome)-length(query)+1)/
    (4^length(query)))])
```

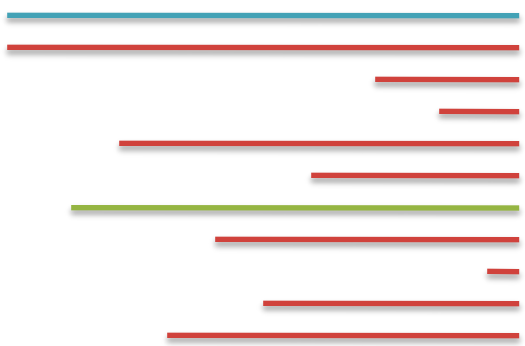
2. Suffix Arrays

- 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

Searching the Index

- Strategy 2: Binary search
  - Compare to the middle, refine as higher or lower

- Searching for GATTACA
  - Lo = 1; Hi = 15; Mid = (1+15)/2 = 8
  - Middle = Suffix[8] = CC
    => Higher: Lo = Mid + 1

  - Lo = 9; Hi = 15; Mid = (9+15)/2 = 12
  - Middle = Suffix[12] = TACC
    => Lower: Hi = Mid - 1

  - Lo = 9; Hi = 11; Mid = (9+11)/2 = 10
  - Middle = Suffix[10] = GATTACC
    => Lower: Hi = Mid - 1

  - Lo = 9; Hi = 9; Mid = (9+9)/2 = 9
  - Middle = Suffix[9] = GATTACA…
    => Match at position 2!

	#	Sequence	Pos
	1	ACAGATTACC…	6
	2	ACC…	13
	3	AGATTACC…	8
	4	ATTACAGATTACC…	3
	5	ATTACC…	10
	6	C…	15
	7	CAGATTACC…	7
Lo	8	CC…	14
Hi	9	GATTACAGATTACC…	2
	10	GATTACC…	9
	11	TACAGATTACC…	5
	12	TACC…	12
	13	TGATTACAGATTACC…	1
	14	TTACAGATTACC…	4
	15	TTACC…	11

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

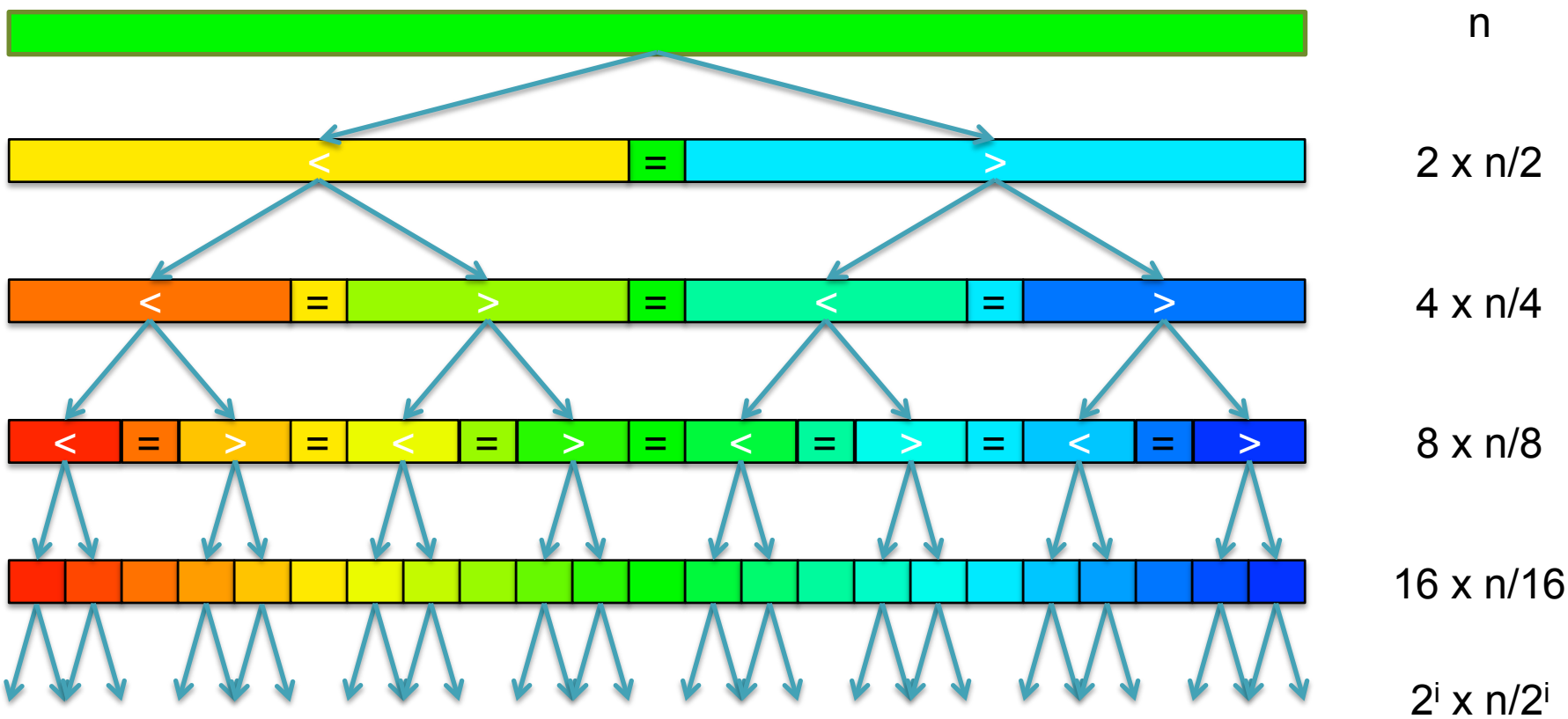
[How long does it take to search 6B or 24B nucleotides?]

Binary Search in Matlab

```
%% create our “sorted” list of 100 numbers
seq=1:100;
query=33;

%% initialize search range
lo=1;
hi=length(seq);
steps=0;

%% search
while (lo<=hi)
  steps = steps+1;
  mid=floor((lo+hi)/2);
  middle=seq(mid);
  disp(['Step ', num2str(steps), ' checking seq[', num2str(mid), ']=', num2str(middle)])
  if (query == middle)
   disp(['Found at ', num2str(mid), ' in ', num2str(steps), ' steps'])
   break
  elseif (query < middle)
    disp(['less than ', num2str(middle)])
    hi=mid-1;
  else
    disp(['greater than ', num2str(middle)])
    lo=mid+1;
  end
end
```

Divide and Conquer

- 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

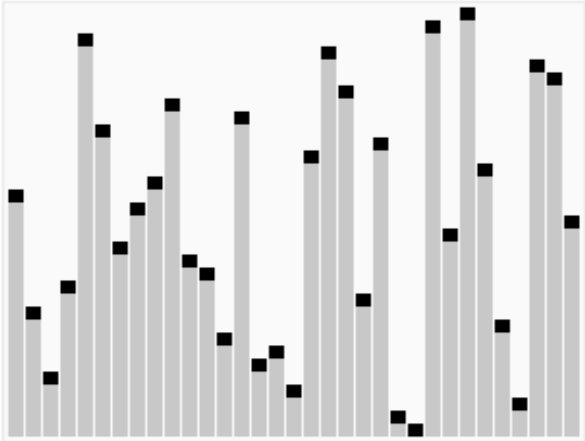


[How many times can we split a list in half?]

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))

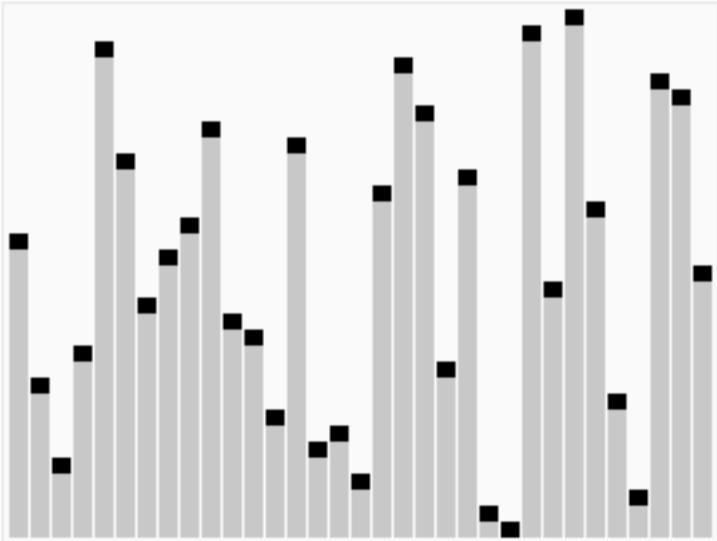
$$T(n) = \begin{cases} O(1) & \text{if } n \leq 1 \\ O(n) + 2T(n/2) & \text{else} \end{cases}$$

$$T(n) = n + 2(\frac{n}{2}) + 4(\frac{n}{4}) + \cdots + n(\frac{n}{n}) = \sum_{i=0}^{lg(n)} \frac{2^i n}{2^i} = \sum_{i=0}^{lg(n)} n = O(n \lg n)$$ [~94B]

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))

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

$$T(n) = n + 2(\frac{n}{2}) + 4(\frac{n}{4}) + \cdots + n(\frac{n}{n}) = \sum_{i=0}^{lg(n)} \frac{2^i n}{2^i} = \sum_{i=0}^{lg(n)} n = O(n \lg n)$$ [~94B]

QuickSort in Matlab

```
sort(seq)
```

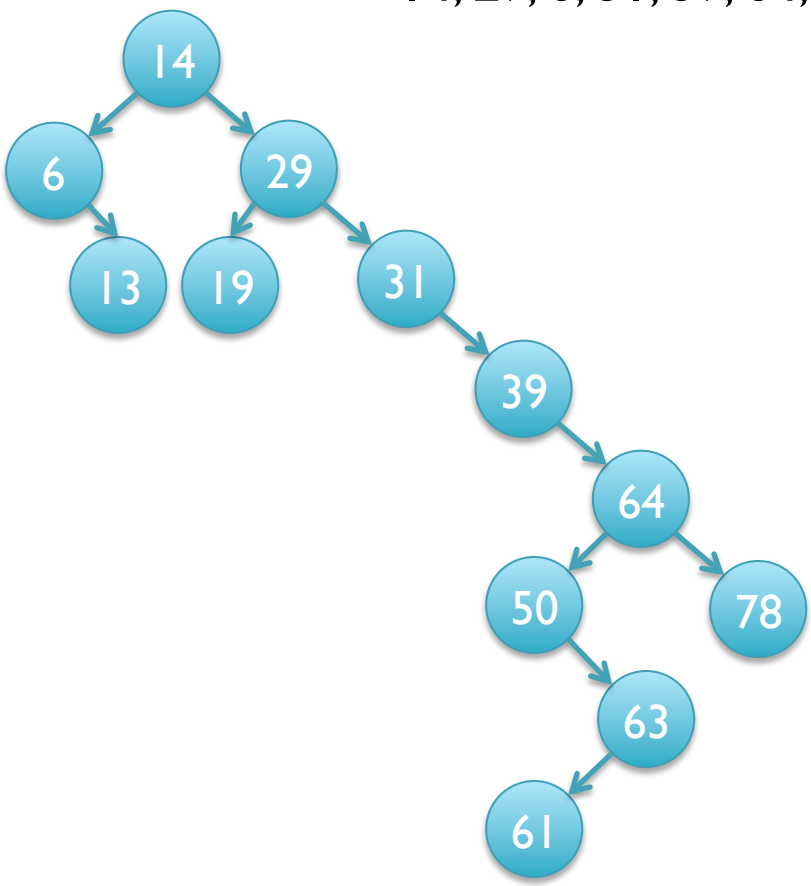
- The goal of software engineering is to build libraries of correct reusable functions that implement higher level ideas
  - Build complex software out of simple components
  - Software tends to be 90% plumbing, 10% research
  - You still need to know how they work
    - Matlab requires an explicit representation of the strings

Binary Search Trees

Trees are useful for storing all kinds of data

- Nodes contain 1 element indexed by the key
- Left branch has elements that are smaller, right branch larger
- Generally very fast to build or search or **modify**

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19



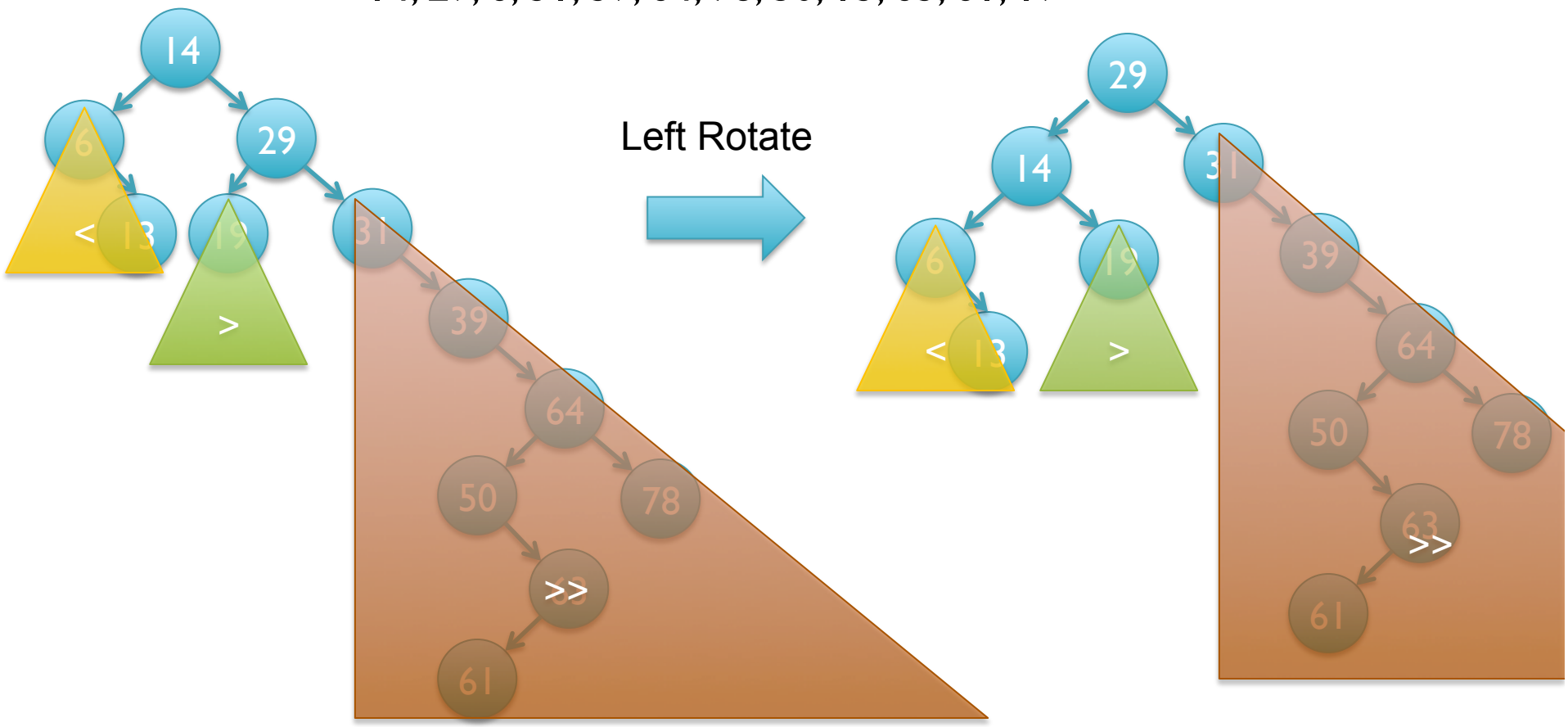
- Binary search trees are generally very fast and are on average:
  - O(lg n) search [why?]
  - O(n lg n) construction [why?]
- But…
  - They may degenerate into O(n) search (brute force!) if the tree is unbalanced into a long chain
  - Fortunately, we can rebalance the tree in constant time

Binary Search Trees

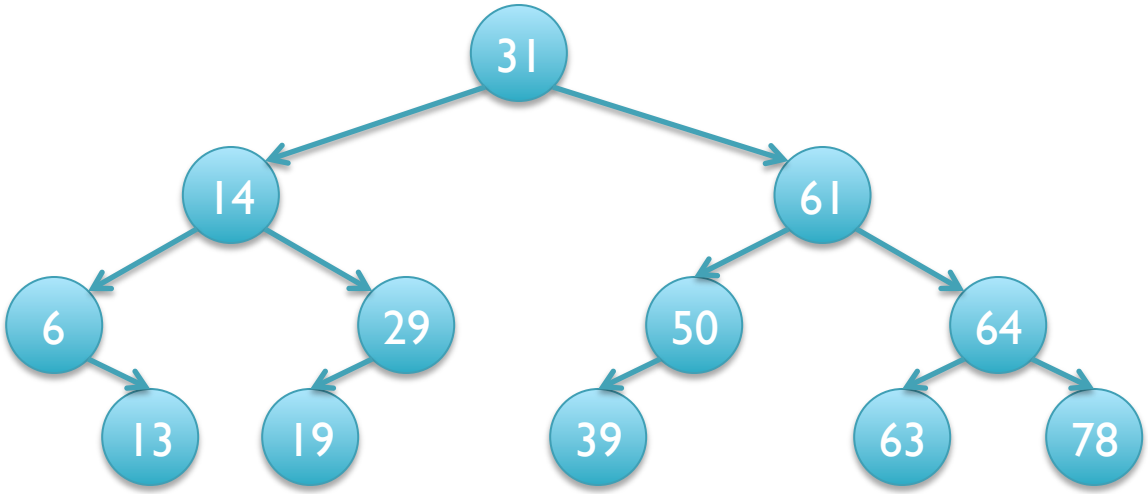
Trees are useful for storing all kinds of data

- Nodes contain 1 element indexed by the key
- Left branch has elements that are smaller, right branch larger
- Generally very fast to build or search or **modify**

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19



Balanced Binary Search Trees



- Using the tree rotate operators, maintain a balanced binary search tree
  - Height: O(lg(n)), Leaves: O(n/2)
  - Print the sorted the values in linear time O(n) [How?]

- Red-Black tree
  - Whenever the tree becomes unbalanced, rotate until balanced
  - http://www.youtube.com/watch?v=vDHFF4wjWYU

- Splay tree
  - Whenever you search for an item, rotate it towards the root
  - http://www.link.cs.cmu.edu/cgi-bin/splay/splay-cgi.pl

Sorting in Linear Time

- Can we sort faster than O(n lg n)?
  - No – Not if we have to compare elements to each other
  - Yes – But we have to 'cheat' and know the structure of the data

Sort these numbers into ascending order:

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19

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	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100

Sorting in Linear Time

- Can we sort faster than O(n lg n)?
  - No – Not if we have to compare elements to each other
  - Yes – But we have to 'cheat' and know the structure of the data

Sort these numbers into ascending order:

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19

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	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100

Sorting in Linear Time

- Can we sort faster than O(n lg n)?
  - No – Not if we have to compare elements to each other
  - Yes – But we have to 'cheat' and know the structure of the data

Sort these numbers into ascending order:

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19

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	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100

6,13,14,19,29,31,39,50,61,63,64,78

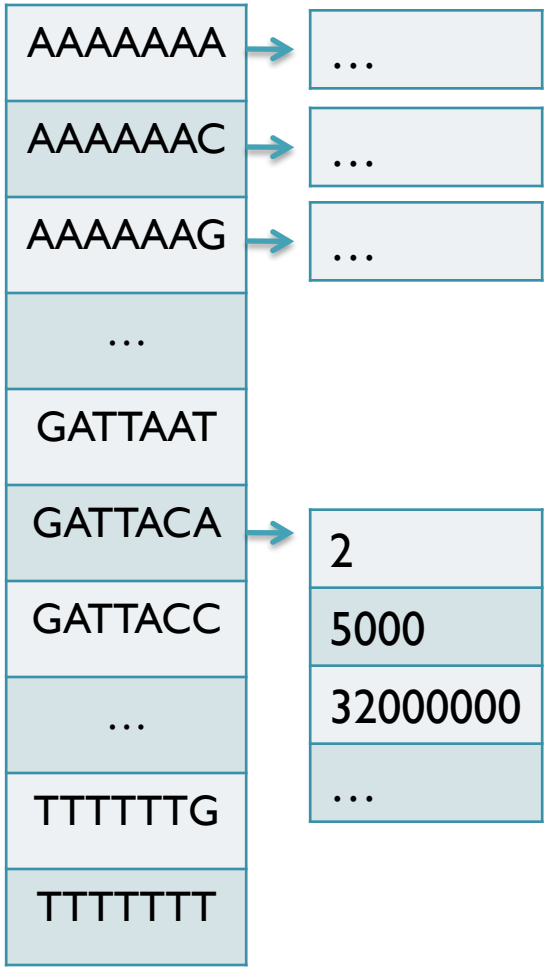
```
for(i = 1 to 100) { cnt[i] = 0; }
for(i = 1 to n) { cnt[list[i]]++; }
for(i = 1 to 100) { while (cnt[i] > 0){print i; cnt[i]--}}
```

[3B instead of 94B]

3. Hashing

- Where is GATTACA in the human genome?
  - Build an inverted index (table) of every kmer in the genome

- How do we access the table?
  - We can only use numbers to index
    - table[GATTACA] <- error, does not compute
  - Encode sequences as numbers
    - Simple: A = 0, C = 1, G = 2, T = 3
      - GATTACA = 2 0 3 3 0 1 0
    - Smart: A = 00_2, C = 01_2, G = 10_2, T = 11_2
      - GATTACA = 10 00 11 11 00 01 00_2 = 9156_{10}
  - Running time
    - Construction: O(n)
    - Lookup: O(1) + O(z)
    - Sorts the genome mers in linear time



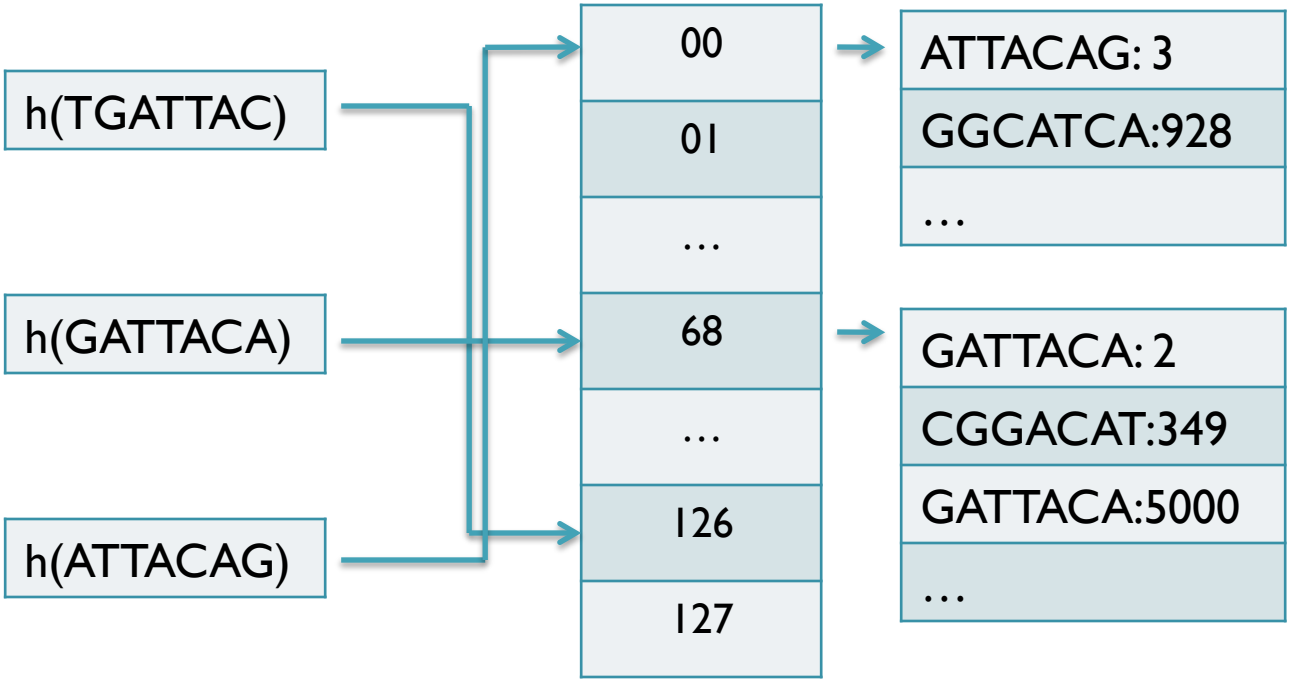
Hash Tables and Hash Functions

- Number of possible sequences of length k = 4^k
  - 4^7 = 16,384 (easy to store)
  - 4^{20} = 1,099,511,627,776 (impossible to directly store in RAM)
    - There are only 3B 20-mers in the genome
      - ⇒ Even if we could build this table, 99.7% will be empty
      - ⇒ But we don't know which cells are empty until we try
- Use a hash function to shrink the possible range
  - Maps a number n in [0,R] to h in [0,H]
    - » Use 128 buckets instead of 16,384, or 1B instead of 1T
  - Division: hash(n) = H * n / R;
    - » hash(GATTACA) = 128 * 9156 / 16384 = 71
  - Modulo: hash(n) = n % H
    - » hash(GATTACA) = 9156 % 128 = 68

[What properties do we want in a hash functions?]

Hash Table Lookup

- By construction, multiple keys have the same hash value
  - Store elements with the same key in a bucket chained together
    - A good hash evenly distributes the values: R/H have the same hash value
  - Looking up a value scans the entire bucket
    - Slows down the search as a function of the hash table load
    - Warning: This complexity is usually hidden in the hash table code



[How many elements do we expect per bucket?]

Variable Length Queries

- Where are GATTACA and GATTACCA in the human genome?
  - s = min(length of all queries)
  - Build an inverted index of all s-mers (seeds) in the genome
    - GATTACA => 2, 5000, 32000000, …
    - GATTACC => 5500, 10101, 1000000, …

- Seed-and-extend to find end-to-end exact matches
  - Check every occurrence of the qry seed (first s characters)
    - ~1 in 4 are GATTACCA, 1 in 4 are GATTACCC, etc
  - The specificity of the seed depends on length(q) & s
    - Works best if max(length) =~ min(length)
    - Works best if e-value(m) is << 1

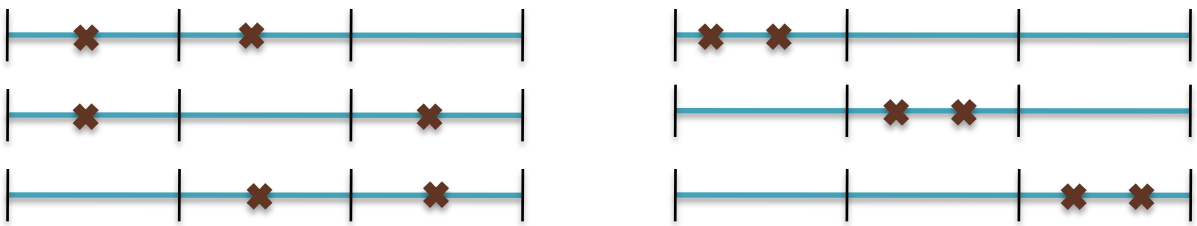
Seed-and-Extend Alignment

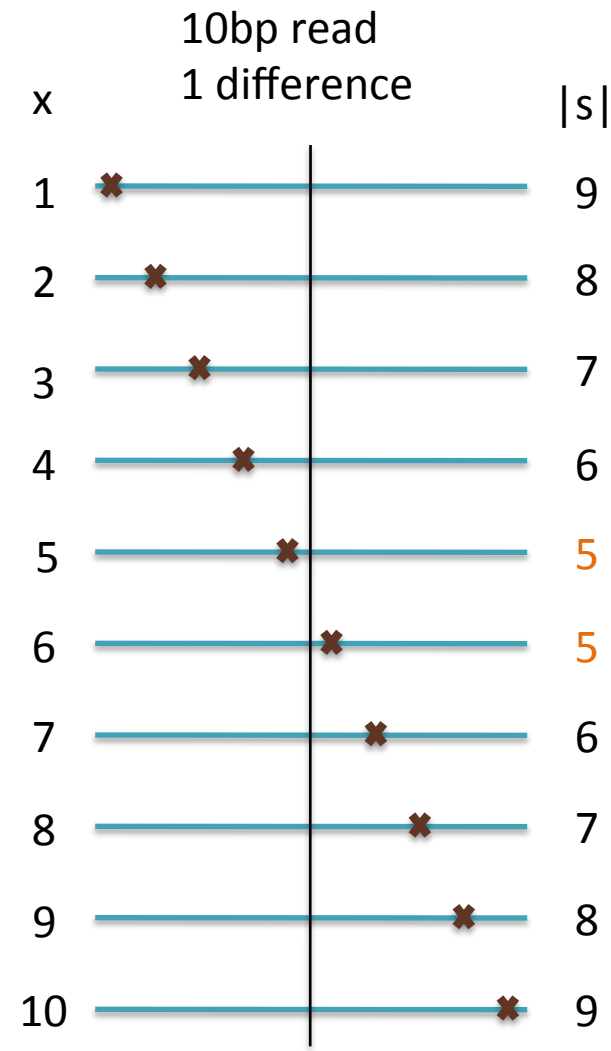
Theorem: An alignment of a sequence of length *m* with at most *k* differences ***must*** contain an exact match at least $s=m/(k+1)$ bp long

(*Baeza-Yates* and Perleberg, 1996)

Proof: Pigeon hole principle

K=2 pigeons (differences) can't fill all K+1 pigeon holes (seeds)

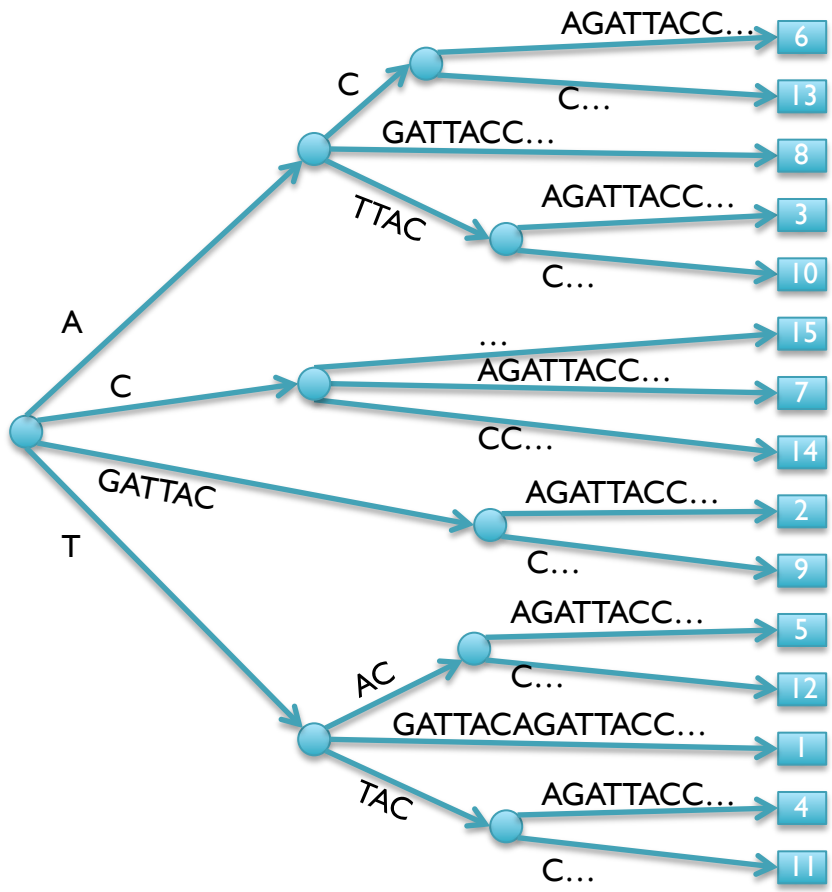




- Search Algorithm
  - Use an index to rapidly find short exact alignments to seed longer in-exact alignments
    - RMAP, CloudBurst, …
  - Length s seeds can also seed some lower quality alignments
    - Won't have perfect sensitivity, but avoids very short seeds

4. Suffix Trees (Optional)

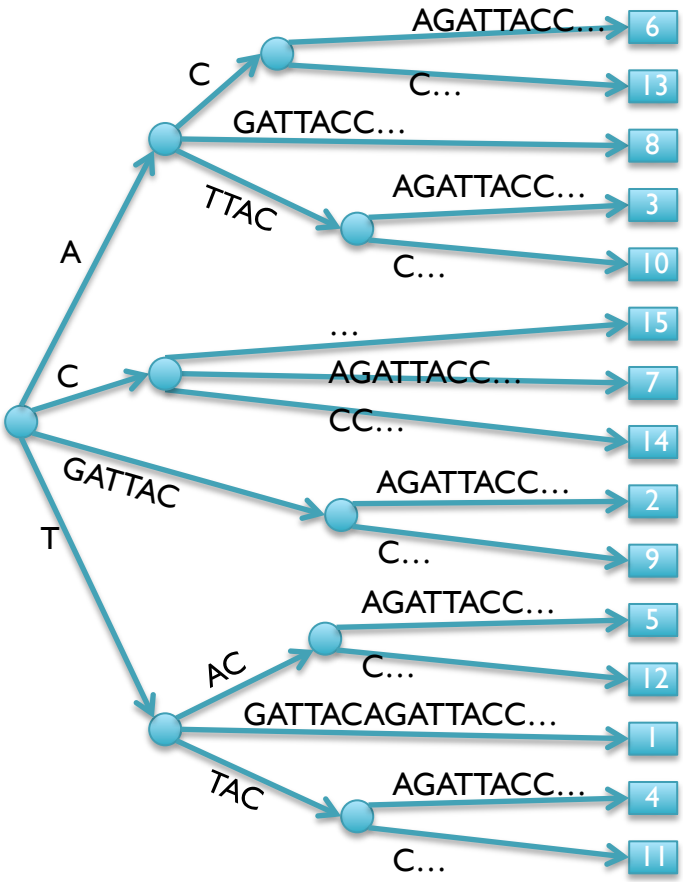
#	Sequence	Pos
1	ACAGATTACC…	6
2	ACC…	13
3	AGATTACC…	8
4	ATTACAGATTACC…	3
5	ATTACC…	10
6	C…	15
7	CAGATTACC…	7
8	CC…	14
9	GATTACAGATTACC…	2
10	GATTACC…	9
11	TACAGATTACC…	5
12	TACC…	12
13	TGATTACAGATTACC…	1
14	TTACAGATTACC…	4
15	TTACC…	11



- Suffix Tree = Tree of suffixes (indexes **all** substrings of a sequence)
  - 1 Leaf ($) for each suffix, path-label to leaf spells the suffix
  - Nodes have at least 2 and at most 5 children (A,C,G,T,$)

Suffix Trees Searching

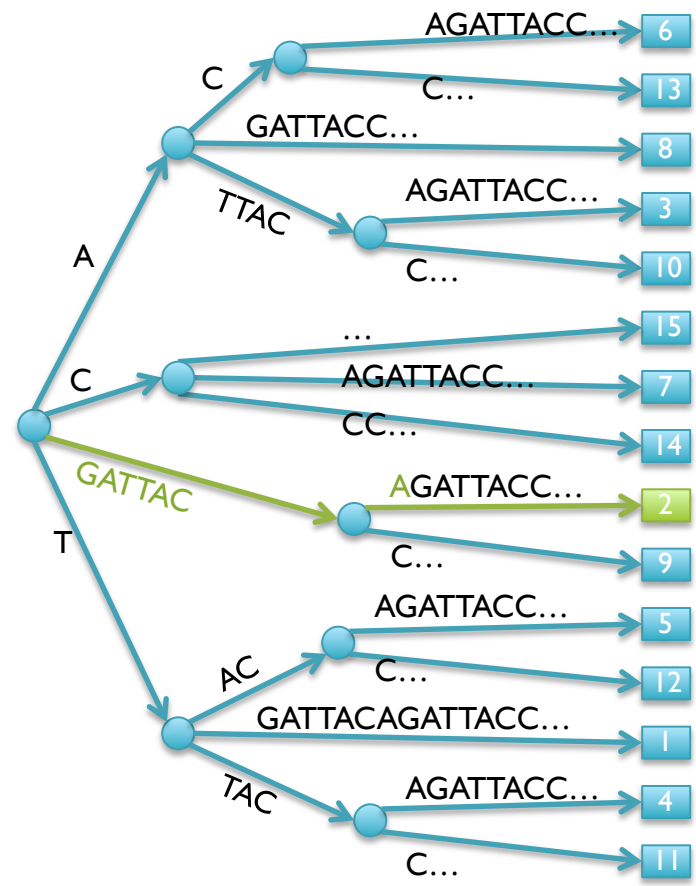
- Look up a query by "walking" along the edges of the tree
    - GATTACA



Suffix Trees Searching

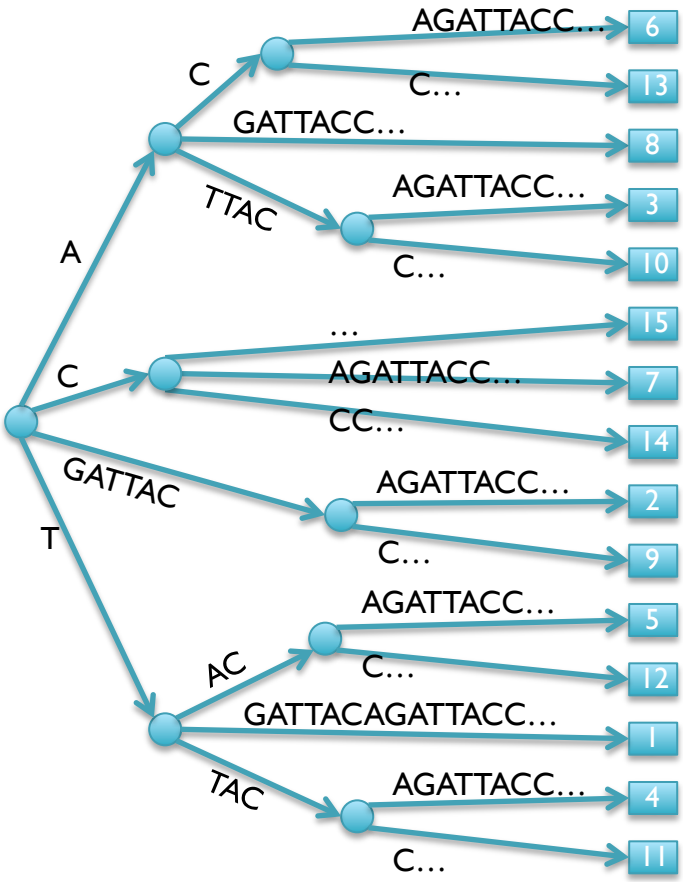
- Look up a query by "walking" along the edges of the tree
  - GATTACA
  - Matches at position 2

```
WalkTree
    cur = ST.Root;
    qrypos = 0;
    while (cur)
        edge = cur.getEdge(q[qrypos]);
        dist = matchstrings(edge, qry, qrypos)
        if (qrypos+dist == length(qry))
            print "end-to-end match"
        else if  (dist == length(edge))
            cur=cur.getNode(edge[0]);
            qrypos+=dist
        else
            print "no match"
```



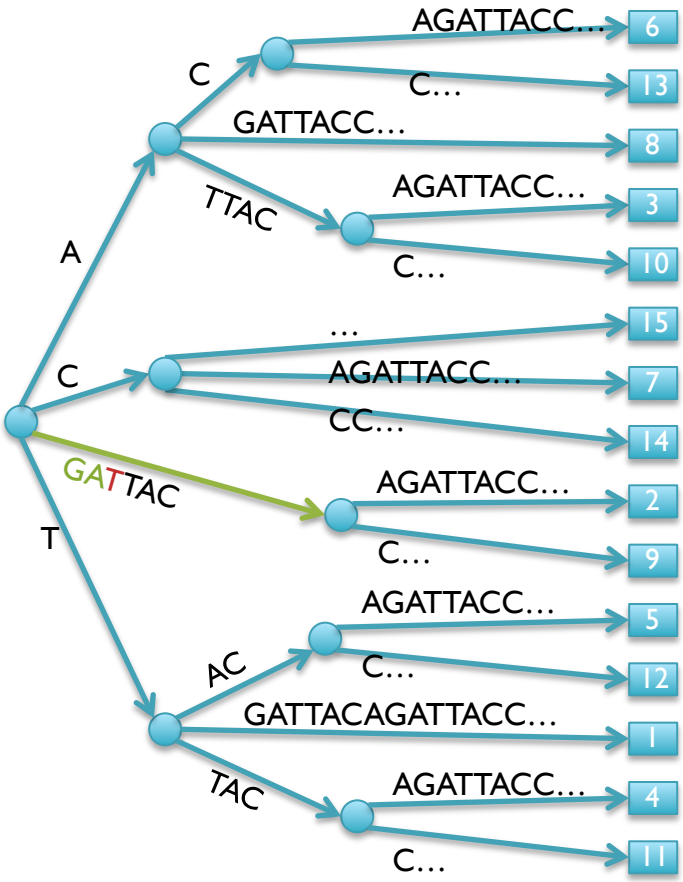
Suffix Trees Searching

- Look up a query by "walking" along the edges of the tree
    - GACTACA



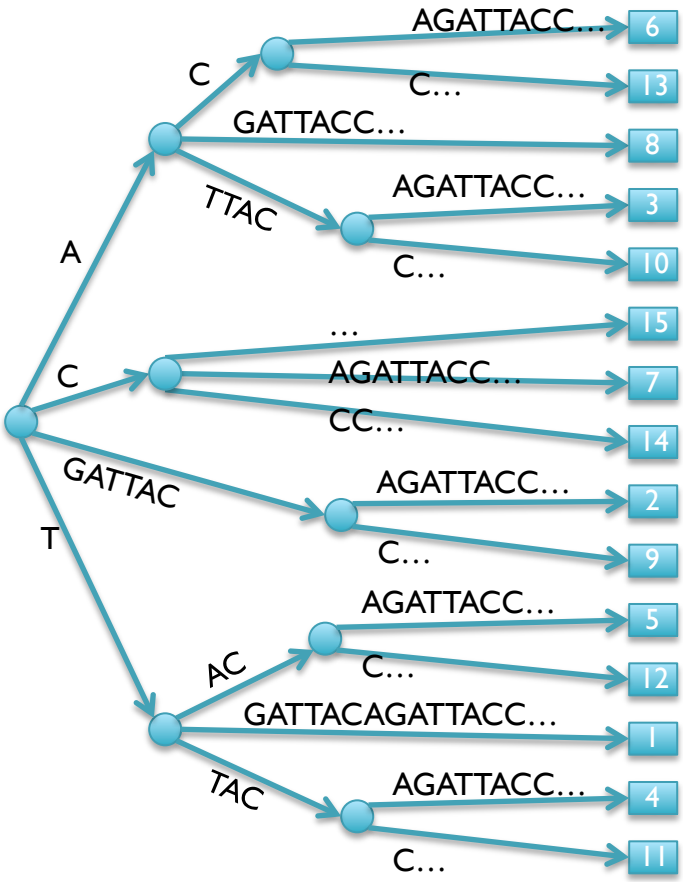
Suffix Trees Searching

- Look up a query by "walking" along the edges of the tree
  - GACTACA
  - Fell off tree – no match



Suffix Trees Searching

- Look up a query by "walking" along the edges of the tree
    - ATTAC



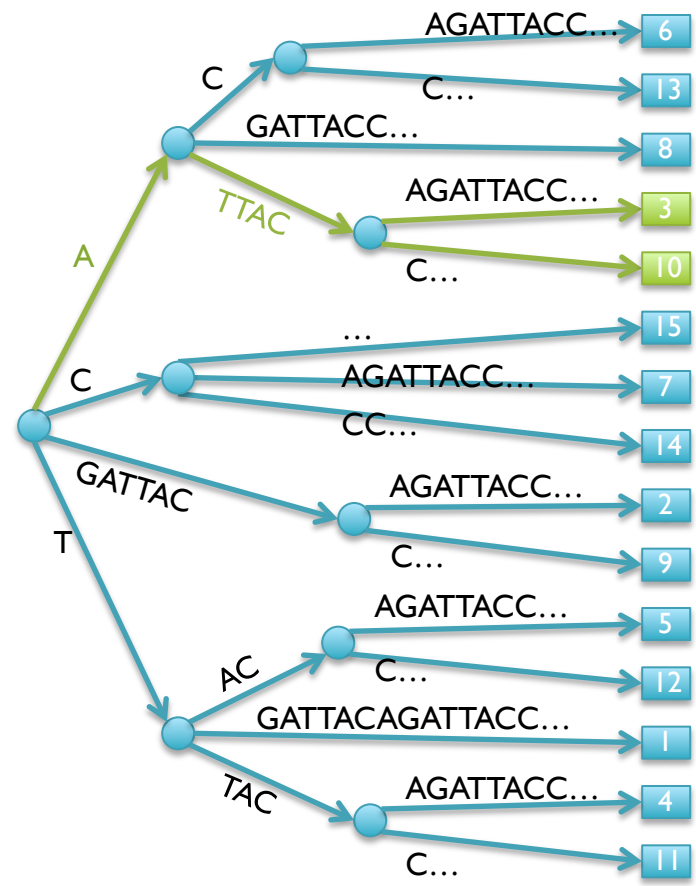
Suffix Trees Searching

- Look up a query by "walking" along the edges of the tree
  - ATTAC
  - Matches at 3 and 10

- Query Lookup in 2 phases:
  1. Walk along edges to find matches
  2. Walk subtree to find positions

```
DepthFirstPrint(Node cur)
if cur.isLeaf
    print cur.pos
else
    foreach child in cur.children
     DepthFirstPrint(child)
```

[What is the running time of DFP
=> How many nodes does the tree have?]



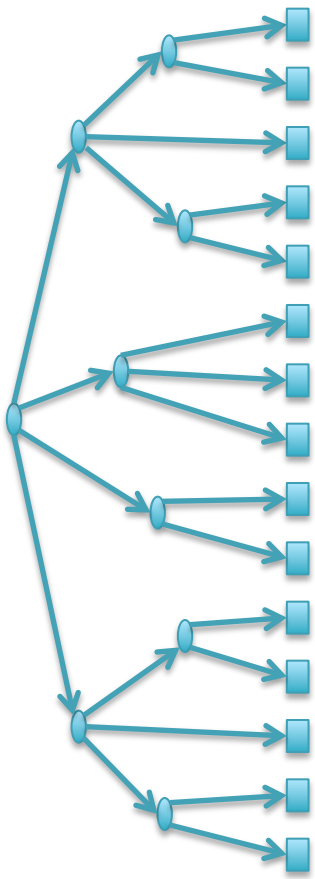
Suffix Tree Properties & Applications

Properties

- Number of Nodes/Edges: O(n)
- Tree Size: O(n)
- Max Depth: O(n)
- Construction Time: O(n)
  - Tricky to implement, prove efficiency
  - Brute force algorithm requires $O(n^2)$

Applications

- Sorting all suffixes: O(n) [HOW?]
- Check for query: O(m)
- Find all z occurrences of a query O(m + z)
- Find maximal exact matches O(m)
- Longest common substring O(m)



- Used for many string algorithms in linear time
  - Many can be implemented on suffix arrays using a little extra work

THE G-NOME PROJECT

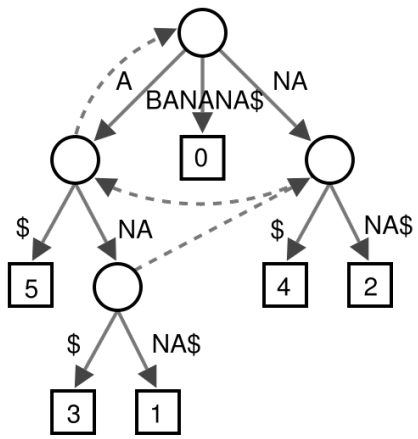
Break

Bowtie: Ultrafast and memory efficient alignment of short DNA sequences to the human genome

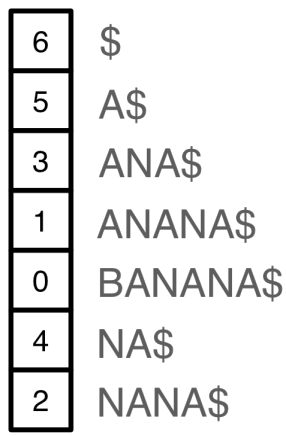
Slides Courtesy of Ben Langmead
(langmead@umiacs.umd.edu)

Indexing

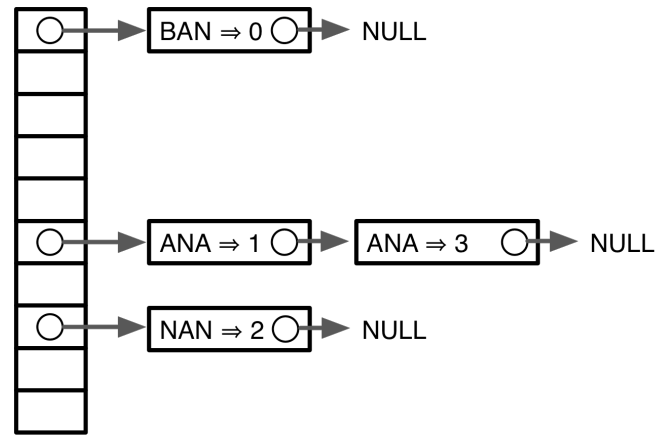
- Genomes and reads are too large for direct approaches like dynamic programming
  - Genome indices can be big. For human:



> 35 GBs



> 12 GBs

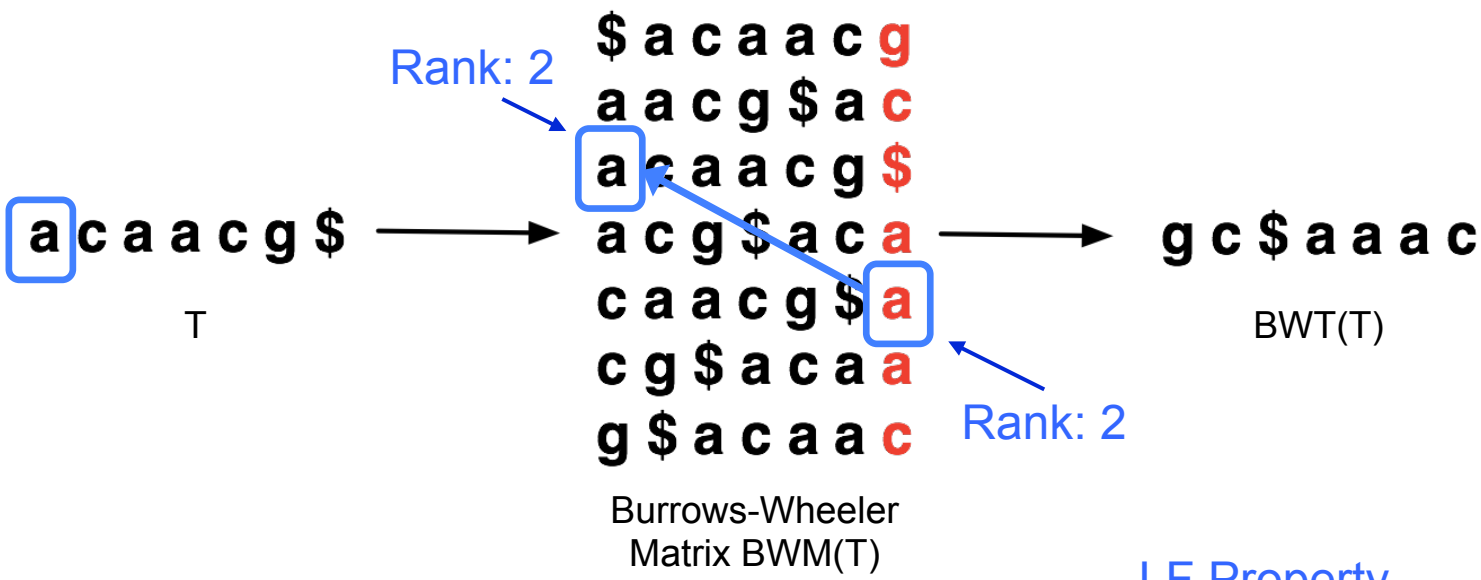


> 12 GBs

- Large indices necessitate painful compromises
  1. Require big-memory machine
  2. Use secondary storage
  3. Build new index each run
  4. Subindex and do multiple passes

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

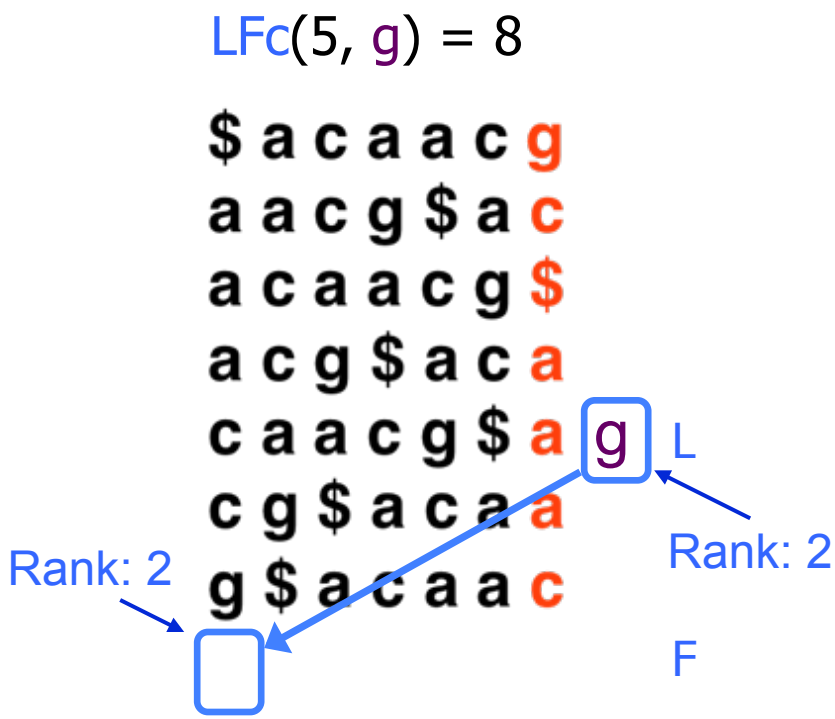
Burrows-Wheeler Transform

- 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]

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:

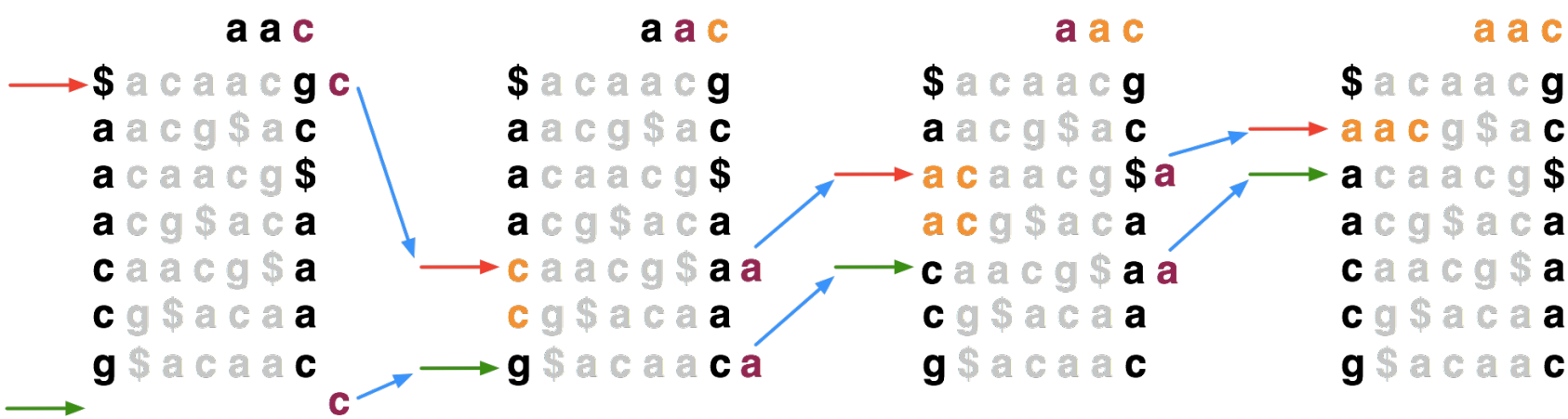


BWT Exact Matching

- 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]

Bowtie algorithm

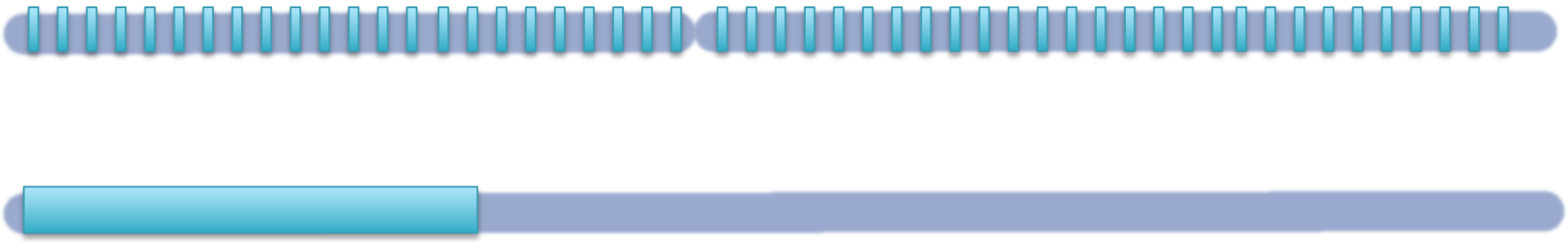
Reference

BWT(Reference)

Query:
A A T G A T A C G G C G A C C A C C G A G A T C T A

Bowtie algorithm

Reference

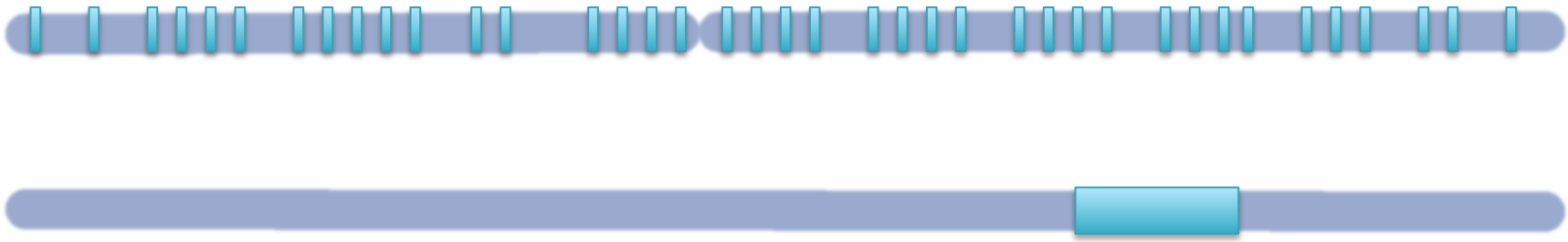


BWT(Reference)

Query:
A A T G A T A C G G C G A C C A C C G A G A T C T A

Bowtie algorithm

Reference

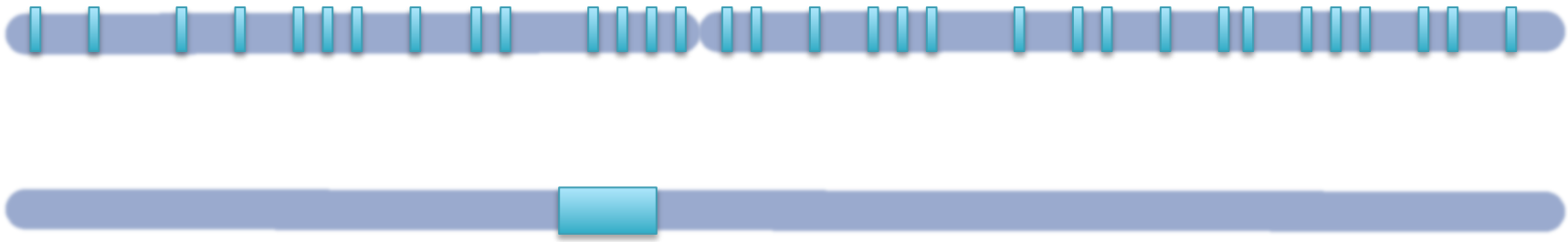


BWT(Reference)

Query:
A A T G A T A C G G C G A C C A C C G A G A T C T A

Bowtie algorithm

Reference

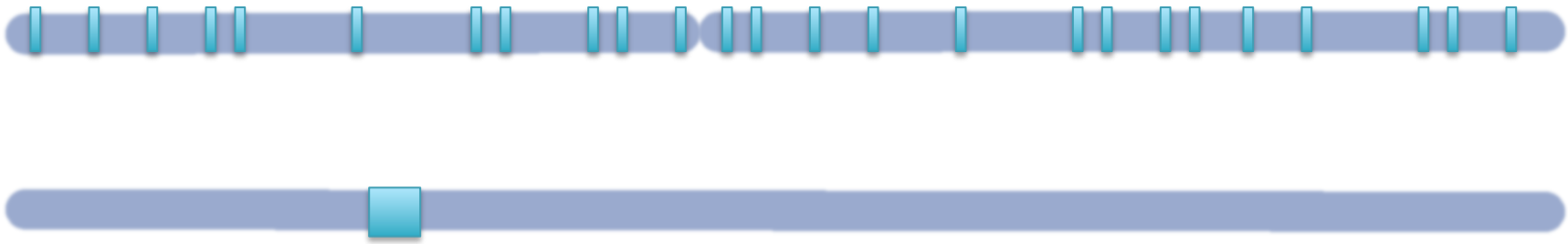


BWT(Reference)

Query:
A A T G A T A C G G C G A C C A C C G A G A T C T A

Bowtie algorithm

Reference



BWT(Reference)

Query:
A A T G A T A C G G C G A C C A C C G A G A T C T A

Bowtie algorithm

Reference

BWT(Reference)

Query:
A A T G A T A C G G C G A C C A C C G A G A T C T A

Bowtie algorithm

Reference

BWT(Reference)

Query:
A A T G A T A C G G C G A C C A C C G A G A T C T A

Bowtie algorithm

Reference

BWT(Reference)

Query:
A A T G T T A C G G C G A C C A C C G A G A T C T A

Bowtie algorithm

Reference

BWT(Reference)

Query:

A A T G T T A C G G C G A C C A C C G A G A T C T A

BWT Short Read Mapping

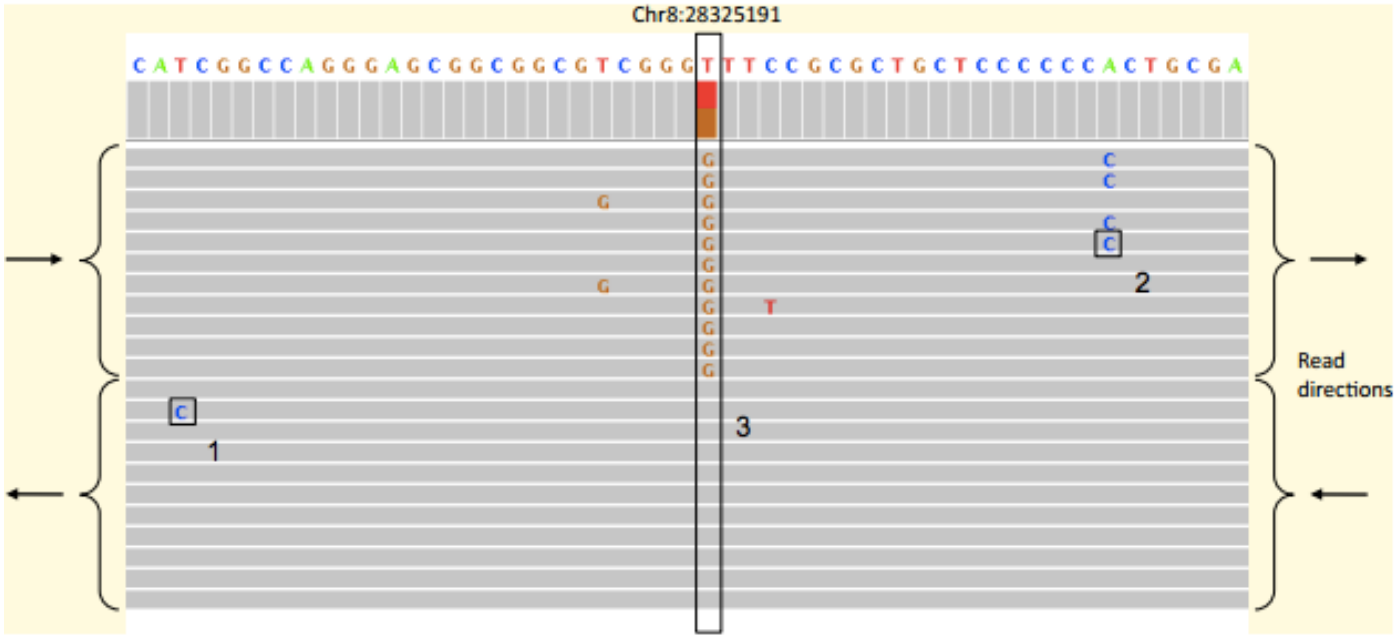
1. Trim off very low quality bases & adapters from ends of sequences

2. Execute depth-first-search of the implicit suffix tree represented by the BWT
   1. If we fail to reach the end, back-track and resume search
   2. BWT enables searching for good end-to-end matches entirely in RAM
      1. 100s of times faster than competing approaches

3. Report the "best" n alignments
   1. Best = fewest mismatches/edit distance, possibly weighted by QV
   2. Some reads will have millions of equally good mapping positions
   3. If reads are paired, try to find mapping that satisfies both

SNP calling

Beware of (Systematic) Errors



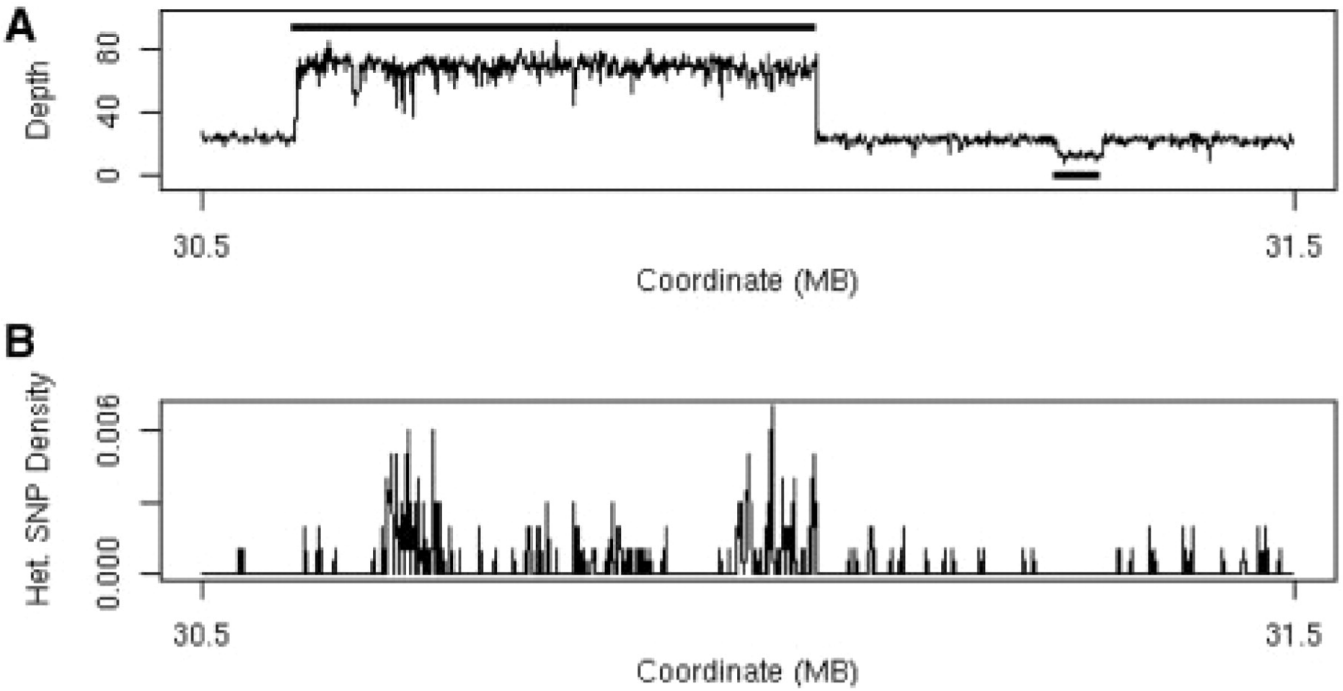
- Distinguishing SNPs from sequencing error typically a likelihood test of the coverage
  - Probability of seeing the data from a heterozygous SNP versus from sequencing error
  - However, some sequencing errors are systematic!

Identification and correction of systematic error in high-throughput sequence data
Meacham et al. (2011) *BMC Bioinformatics.* 12:451

A closer look at RNA editing.
Lior Pachter (2012) *Nature Biotechnology.* 30:246-247

CNV calling

Beware of (Systematic) Errors

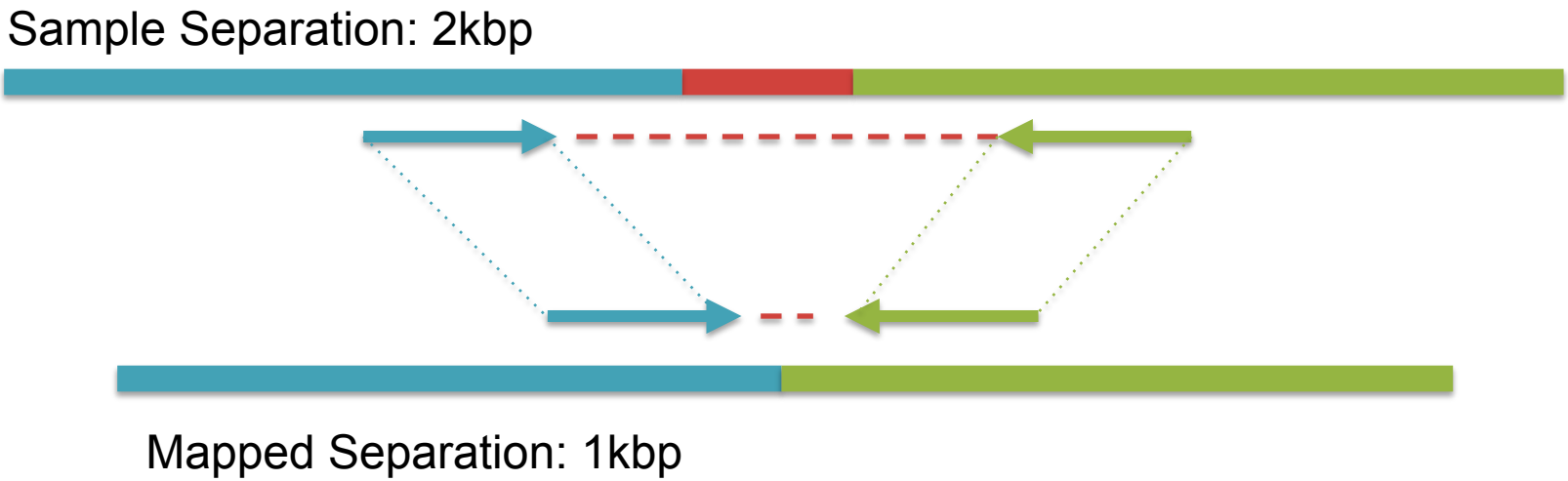


(A) Plot of sequencing depth across a one megabase region of A/J chromosome 17 clearly shows both a region of 3-fold increased copy number (30.6–31.1 Mb) and a region of decreased copy number (at 31.3 Mb).

Simpson J T et al. Bioinformatics 2010;26:565-567

- Identify CNVs through increased depth of coverage & increased heterozygosity
  - Segment coverage levels into discrete steps
  - Be careful of GC biases and mapping biases of repeats

Structural Variations

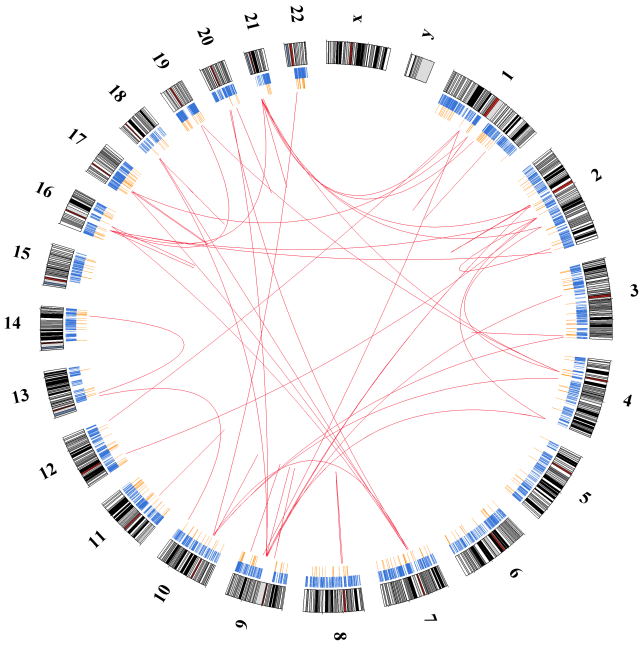


SVs tend to be flanked by repeats, making it hard to localize

- Longer reads are the key to resolving them

Circos plot of high confidence SVs specific to esophageal cancer sample

- Red: SV links
- Orange: 375 cancer genes
- Blue: 4950 disease genes



Exact Matching Review

- E-value depends on length of genome and inversely on query length
  - $E = (n-m+1)/4^m$

Brute Force (3 GB)

```
BANANA
BAN
  ANA
    NAN
      ANA
```

Naive

Slow & Easy

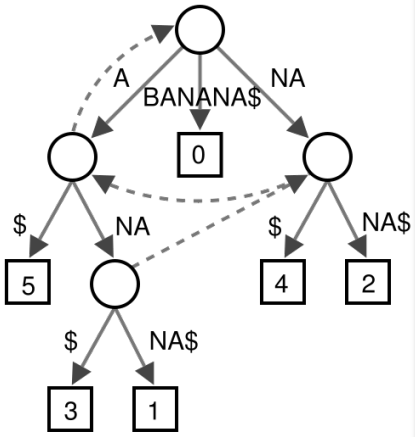
Suffix Array (>15 GB)

6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Vmatch, PacBio Aligner

Binary Search

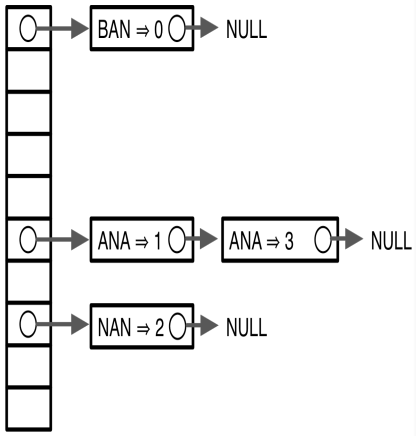
Suffix Tree (>51 GB)



MUMmer, MUMmerGPU

Tree Walking & DFS

Hash Table (>15 GB)



BLAST, MAQ, ZOOM, RMAP, CloudBurst

Seed-and-extend

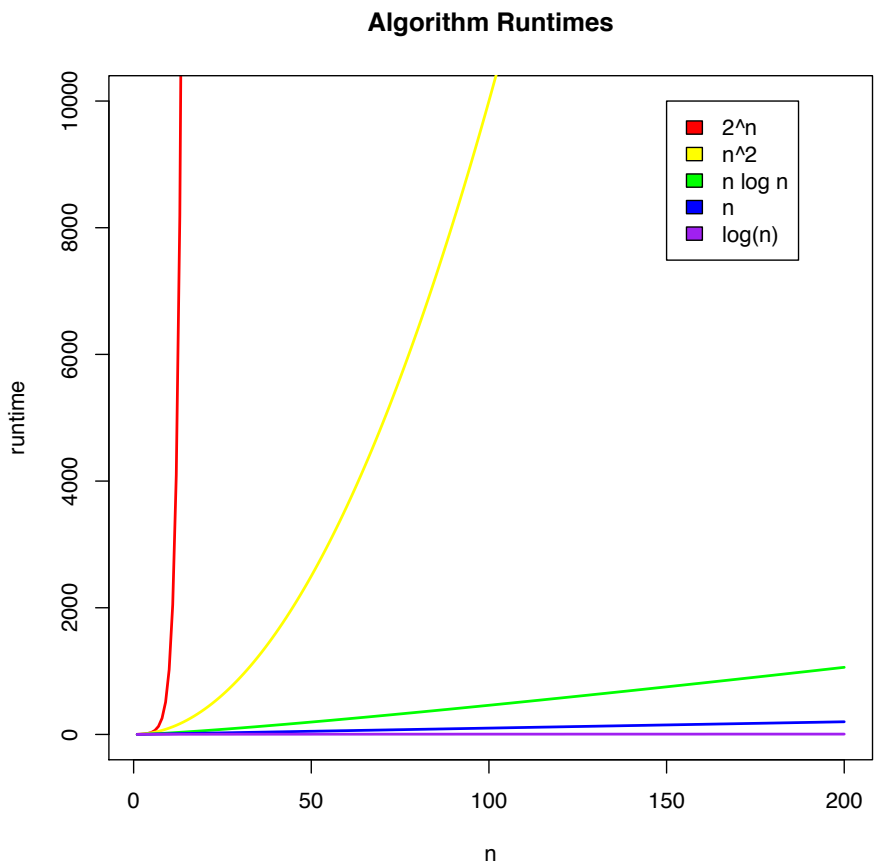
Algorithms Summary

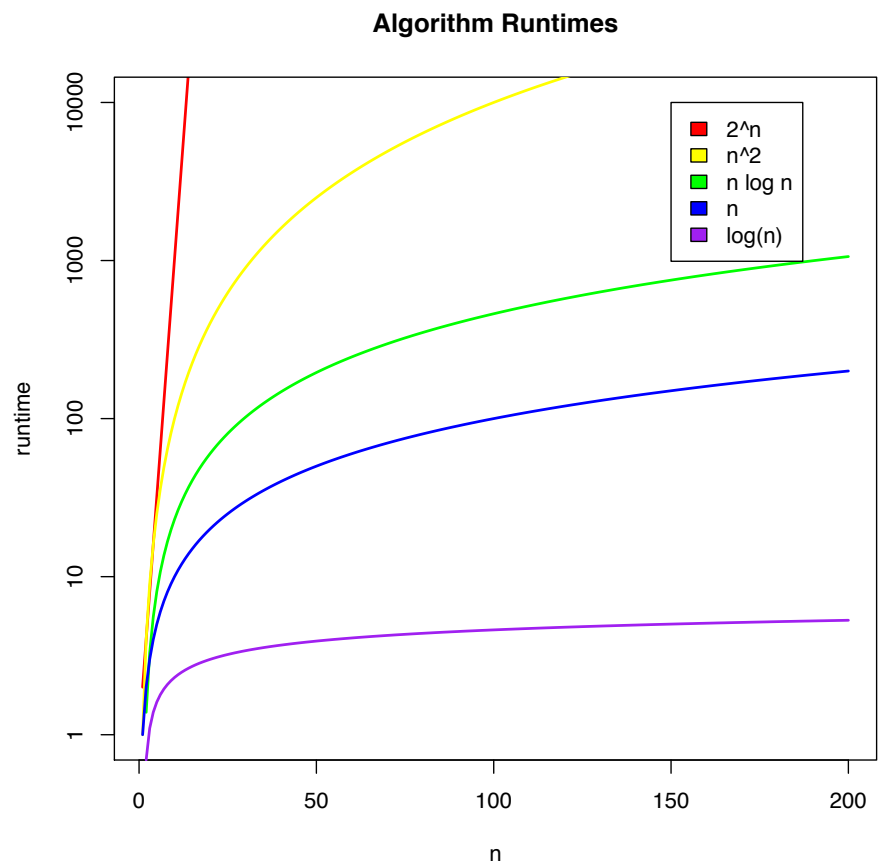
- Algorithms choreograph the dance of data inside the machine
  - Algorithms add provable precision to your method
  - A smarter algorithm can solve the same problem with much less work

- Techniques
  - Binary search: Fast lookup in any sorted list
  - Divide-and-conquer: Split a hard problem into an easier problem
  - Recursion: Solve a problem using a function of itself
  - Randomization: Avoid the demon
  - Hashing: Storing sets across a huge range of values
  - Indexing: Focus on the search on the important parts
    - Different indexing schemes have different space/time features

- Data Structures
  - Primitives: Integers, Numbers, Strings
  - Lists / Arrays / Multi-dimensional arrays
  - Trees
  - Hash Table

Algorithmic Complexity





What is the runtime as a function of the input size?

Next Time

- In-exact alignment
  - Smith & Waterman (1981) *Identification of Common Molecular Subsequences*. J. of Molecular Biology. 147:195-197.
- Sequence Homology
  - Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990). *Basic local alignment search tool*. J of Molecular Biology. 215 (3): 403–410.
- Whole Genome Alignment
  - A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg (1999) *Alignment of Whole Genomes*. Nucleic Acids Research (27):11 2369-2376.

Thank You!

http://schatzlab.cshl.edu
@mike_schatz

Supplemental

Original: GATTTACA

Cyclic Rotations	BWM
GATTTACA$	$GATTTACA
ATTTACA$G	A$GATTTAC
TTTACA$GA	ACA$GATTT
TTACA$GAT	ATTTACA$G
TACA$GATT	CA$GATTTA
ACA$GATTT	GATTTACA$
CA$GATTTA	TACA$GATT
A$GATTTAC	TTACA$GAT
$GATTTACA	TTTACA$GA

BWT: ACTGA$TTA