Mapping Reads to Reference Genome
• DNA carries genetic information
• DNA is a double helix of two complementary strands formed by four nucleotides (bases): **Adenine, Cytosine, Guanine and Thymine**
• **Gene expression** is the process by which DNA is transcribed into mRNA (eventually translated into proteins)

• Mechanisms controlling gene expression are not fully understood yet
• New-generation sequencing technology allows fast and inexpensive DNA sequencing
• Helps biologists to study cellular processes
Example: Identify Transcription Factors binding sites

Promoters are located **upstream** from the DNA region that contains the information to be transcribed into mRNA.
Example: Identify Transcription Factors binding sites

Cell diagram adapted from LadyOfHats' Animal Cell diagram. Wikipedia.
Example: Identify Transcription Factors binding sites

1. Shear DNA strands by sonicating
2. Add bead-attached antibodies to immunoprecipitate target proteins
3. Precipitate
4. Unlink protein; purify DNA
Example: Identify Transcription Factors binding sites

Reference genome

ATGCCTGGAACCGTG

sequencing

map to genome

reads
• Mapping DNA reads back to a reference genome is the first step in the data analysis
• Mapping short sequenced reads back to a reference genome is a string search problem: given a text and a query, find all (approximate) occurrences of the query in the text

TCAAG
ATATGTAGTGTTAGTCAAGTTAAGACCTATATGTTAG
Group Work

• Assume that a human reference genome is given (a string of 3 billion characters long)
• Assume that you need to map 1 million 50bp reads to the genome
• Come up with a method to map fast the reads to the genome

Problem statement:
Given a string $S$ of length $n$ and a short string $P$ of length $m$ ($n >> m$), find all locations where $P$ occurs in $S$
Methods for Mapping Short Reads

- To speed up mapping, search space is reduced by focusing only on those regions of genome that share the same seed(s) with a read
- A seed, or $k$-mer ($q$-gram), is a substring of a read of length $k$
- Common data structures to index the data (genome) and speed-up the search:
  - hash tables
  - suffix trees
  - suffix arrays
  - Burrows-Wheeler transform (BWT) with Ferragina-Manzini (FM) index
Hash Indexing

- Hash all genome $k$-mers into a hash table using seeds of fixed length $k$ as hash keys, and corresponding genomic positions as values
- Use the $k$-mers in a read as hash keys to retrieve locations that are potential hits
- Align the entire read to the potential locations and count the number of mismatches
## Hash Indexing

<table>
<thead>
<tr>
<th>Seed Size</th>
<th>Table Size</th>
<th>Space, GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>18,446,744,073,709,600,000</td>
<td>147,573,952,590</td>
</tr>
<tr>
<td>28</td>
<td>72,057,594,037,927,900</td>
<td>576,460,752</td>
</tr>
<tr>
<td>24</td>
<td>281,474,976,710,656</td>
<td>2,251,800</td>
</tr>
<tr>
<td>20</td>
<td>1,099,511,627,776</td>
<td>8,796</td>
</tr>
<tr>
<td>18</td>
<td>68,719,476,736</td>
<td>550</td>
</tr>
<tr>
<td>16</td>
<td>4,294,967,296</td>
<td>34</td>
</tr>
<tr>
<td>12</td>
<td>16,777,216</td>
<td>134 MB</td>
</tr>
</tbody>
</table>
Hash Indexing

Disadvantages:
1. The longer seeds, the more space demanding
2. The shorter seeds, the more time consuming
1. Build a hash table for the following sequence using seeds of length 2 and 3

ATATGTTAGTCAAGTTAAGACCTATGTTAG

2. Map read TATG to the given sequence using the seed TA (TAT) and your hash tables. How many different alignments did you have to make?
1. Reads are generated from both strands of DNA
2. Reads are always sequenced from 5’ to 3’
3. Mapping is performed to only (+) strand of DNA
4. Map reverse-complement of a read: ATTGC, rc: GCAAT
### Mapping

**Output format**

<table>
<thead>
<tr>
<th>Read_ID</th>
<th>Read</th>
<th>Chromosome</th>
<th>Position</th>
<th>Strand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CTGGC</td>
<td>1</td>
<td>1</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>ATTGC</td>
<td>1</td>
<td>4</td>
<td>-</td>
</tr>
</tbody>
</table>

---

Slide 17 of 31
Encoding of Reads

A = 00
C = 01
G = 10
T = 11
ATTGC = 0011111001

Advantages:
1. Each character takes 2 bits instead of 8 bits
2. Retrieval of all positions where a seed occurs takes $O(1)$ time (use encoding of a seed as an index for a hash table’s bin)
### Suffix Array

<table>
<thead>
<tr>
<th>Circular Shift</th>
<th>Lexicographic Sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATATATTAG$</td>
<td>$ATATATTAG</td>
</tr>
<tr>
<td>TATATTAG$A</td>
<td>AG$ATATATT</td>
</tr>
<tr>
<td>ATATTAG$AT</td>
<td>ATATATTAG$</td>
</tr>
<tr>
<td>TATTAG$ATA</td>
<td>ATATTAG$AT</td>
</tr>
<tr>
<td>ATTAG$ATAT</td>
<td>ATTAG$ATAT</td>
</tr>
<tr>
<td>TTAG$ATATA</td>
<td>G$ATATATTA</td>
</tr>
<tr>
<td>TAG$ATATAT</td>
<td>TAG$ATATAT</td>
</tr>
<tr>
<td>AG$ATATATT</td>
<td>TATATTAG$A</td>
</tr>
<tr>
<td>G$ATATATTA</td>
<td>TATTAG$ATA</td>
</tr>
<tr>
<td>$ATATATTAG</td>
<td>TTAG$ATATA</td>
</tr>
</tbody>
</table>

- Find all circular shifts of the reference genome
- Lexicographically sort the circular shifts
- All circular shifts that start with the same substring are consecutive
- Record the starting indices of the circular shifts
For a given string $S$, $\text{Pos}[i] = j$, such that $S[j...n]$ is a prefix of row $i$ in $M$.

To find a given pattern $W$ of length $m$, we know that all rows having $W$ as a prefix in $M$ are contiguous; hence, positions of $P$ in $S$ are stored in contiguous range $[L, R]$ in the suffix array $\text{Pos}$. 
Suffix arrays: A new method for on-line string searches

Udi Manber  Gene Myers
Suffix Array

Time to find all occurrences of $W$ in $S$ is $O(|W| \log(n))$, where $n = |S|$.

Space to store a suffix array is $4n$.

The authors also proposed algorithm with time $O(|W| + \log(n))$.

**Suffix arrays: A new method for on-line string searches**

Udi Manber  Gene Myers
Input Format

Reference genome is usually given as a set of files, each file per chromosome.

Each file is in **Fasta** format:

```plaintext
>Gene 4582 Conserved Region
GACTAGCATGCAGCATGCAGAGCTAGCAGCGAGCGAGCGAG
ATGCATGCAGCATGCAGCATCGAGCGAGCGAGCGAGCATGCAG
CAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGTAG
CTAGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGC
ATGCTAGCTAGCTAGCTACGTAGCAGCAGCAGCAGCAGCAGCAGATAT
CGAATACAGCGACGATGCATGCATGCATGCATGCATGCATG
CATGATGATATAT
```

```plaintext
>Region_1 taxid=9606|spec=Homo sapiens|chr=1|ctg=NC_000001|str=(+)|start=1655789|end=1655989|len=201
GATGATAATAGCGAGCTCCTGTGCGCCAGAAGCTCAGAGACAGGACAGGCTCGCTCTGTGCTGCGGCACTTCTCTGCTGTGTCGCG
GGATGATGACGCATAAAAACAGCGCTGCTCAGTCCAGGACTCCTAAAGAGCTGCGCCGTAGCTGCACTTTGGACTCTC
GCCCGGGCCCGGGCCCGGCGAGCAGCAGGCGACCTTTTG

>Region_2 taxid=9606|spec=Homo sapiens|chr=1|ctg=NC_000001|str=(-)|start=8014301|end=8014551|len=251
GCCCTGACATTGAGGCGCGTGGTGCTGCAAGGGGGAAATTCTGACACCAGTTCTGCTGAGGGCCTGCGAAGCCTTC
GCCCTGCGGCCACCGATGAGACCCCGGACCGACGACCACTCCAGCGGCGTGCGGGCGCAG
AACAATCGGACTGCCATGACGTTAGGTCACCTTCGCGCGCGCGACGATCGCCACCCGGGATTGGCGAGGTCAAG
TCGCACTGGGC
```
Input Format

Reads are usually given in **FASTQ** format:

```
@ERR030887.1 HWI-BRUNOP16X_0001:8:1:7338:1073#0/1
TNTCGATTACATGTGGATCAGGTTGATTTAATAATGGCGATAGGNNNCT
+
5#1455555555A;A84455555555>>>>.:FFFFFFFFFFFFFFFFFFFFFFFF
@ERR030887.2 HWI-BRUNOP16X_0001:8:1:10288:1073#0/1
TNAGTCTTTTCCACGCCTAACAAGAAGCAAGAATAATTGGGCACNNNGA
+
5#156+43&4(0*55CFDAF########################################################
@ERR030887.3 HWI-BRUNOP16X_0001:8:1:13787:1073#0/1
ANGTTGCTATTTCCGCGCTCTAAACCACACCACCTTTACCGCTANNNNGA
+
5#55555554GGGG?FFFFFFGGGEGGGERGGGEGGCC>CFFFFFFFFFFFFFFFFFFFFFFFF
@ERR030887.4 HWI-BRUNOP16X_0001:8:1:15388:1074#0/1
CNGTTCAAGCAGAAAGACGTCTTGCGCTCTGTATGGACACTGATCNNNGA
+
5#5555255555445EYGGGGGGGGGA@;>A<A>AFFFFFFFFFFFFFFFFFFFFFFFF
@ERR030887.5 HWI-BRUNOP16X_0001:8:1:16693:1073#0/1
CNAGTCCGTCATCCATCCTACCCCTATGGGCGAGGTAAGCCACNNNCC
+
5#555))685=<H<F@1=E:88<==55441A?AADCDBFBFFFFFFFFFFFFFFFF
```
Homework 1 discussion
Next Lecture

• Approximate string search
• Smith-Waterman algorithm
• Hash table, suffix array for approximate string search
Burrows-Wheeler Transform

- Build Burrows-Wheeler transform (BWT) for the reference genome
- Find positions within BWT corresponding to suffixes whose prefix is a seed of the read
- Calculate from these positions genomic positions
- Align the entire read to the potential locations and count the number of mismatches
## Methods for Mapping Short Reads

<table>
<thead>
<tr>
<th></th>
<th>Hash Indexing</th>
<th>Burrows-Wheeler transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seed length for a read</td>
<td>fixed</td>
<td>variable</td>
</tr>
<tr>
<td>Time to find genomic positions for a $k$-mer</td>
<td>$O(1)$</td>
<td>$O(k + Occ \cdot \log N)$</td>
</tr>
<tr>
<td>Time to map entire read of length $P$ to $Occ$ genomic positions, where the $k$-mer occurs</td>
<td>$O(Occ \cdot P)$</td>
<td>$O(Occ \cdot P)$</td>
</tr>
</tbody>
</table>

- The length of the seed used in hashing is fixed and usually shorter than the seed for BWT.
- Hence, $Occ$ with BWT is smaller than $Occ$ with hash indexing.
- We need to check a smaller number of full-length read alignments with BWT; thus, mapping of short reads with BWT is more time-efficient.
Find positions within BWT corresponding to suffixes whose prefix is a seed of the read

Given: $P$, a pattern of length $p$

$BW\_Search(P[1,p])$

c = P[p], i=p

sp = F[c] + 1, ep = F[c+1]

while sp < ep and i > 1

\[ c = P[i-1], i = i - 1 \]

sp = F[c] + Occ(c, 1, sp-1) + 1

ep = F[c] + Occ(c, 1, ep)

print sp and ep

**Example:** $BW\_Search(ATA)$

c = A, i = 3


i=3: c = T, sp = 6 + 0 + 1 = 7, ep = 6 + 3 = 9

i=2: c = A, sp= 1 + 1 + 1 = 3, ep = 1 + 3 = 4

At each iteration $i$, sp points to the first row of M prefixed by $P[i,p]$, and ep points to the last row of M prefixed by $P[i,p]$
Mark row of M corresponding to each 3-d genomic position
• Store explicitly these positions in array GI
• If i-th position in BWT is marked, Occ(1, 1, i) is index for genomic position in GI (e.g., GI[Occ(1,1,3)] = GI[2] = 1)
• If i-th position in BWT is not marked, do LF_mapping until encounter marked position j, BWT[j] = 1, marked
• Pos(sp) = Number_of_LF_mappings + GI[Occ(1,1,j)]

\[ LF\_mapping(sp) \]
\[ c = \text{get\_BWT\_char}(sp) \]
\[ sp = F[c] + Occ(c, 1, sp) \]

**Example:**
1. \( LF\_mapping(4) \)
   \[ c = \text{get\_BWT\_char}(4) = T \]
   \[ sp = F[T] + Occ(T, 1, 4) = 6 + 2 = 8 \text{ (not marked)} \]
2. \( LF\_mapping(8) \)
   \[ c = \text{get\_BWT\_char}(8) = A \]
   \[ sp = F[A] + Occ(A, 1, 8) = 1 + 2 = 3 \text{ (marked)} \]
   \[ \text{Pos}(sp) = 2 + 1 = 3 \text{ (total of 2 LF\_mappings and GI[Occ(1,1,3)])} \]
Why LF_mapping works?

- Why do we identify correctly the genomic position for unmarked BWT[i]?
- Given row M[i] starting with prefix P, we find the closest marked preceding genomic position.
- Since the rows of M are the circular shifts of T$, the last character of i-th row, L[i], precedes the first character F[i].
- Let L[i] = c and ri be the rank of the row M[i] among all rows ending with c. Then F[j] = c is the corresponding character to L[i] in T, where M[j] is the ri-th row of M starting with c.
- Define LF_mapping as
  \[ LF[i] = F[L[i]] + ri \]