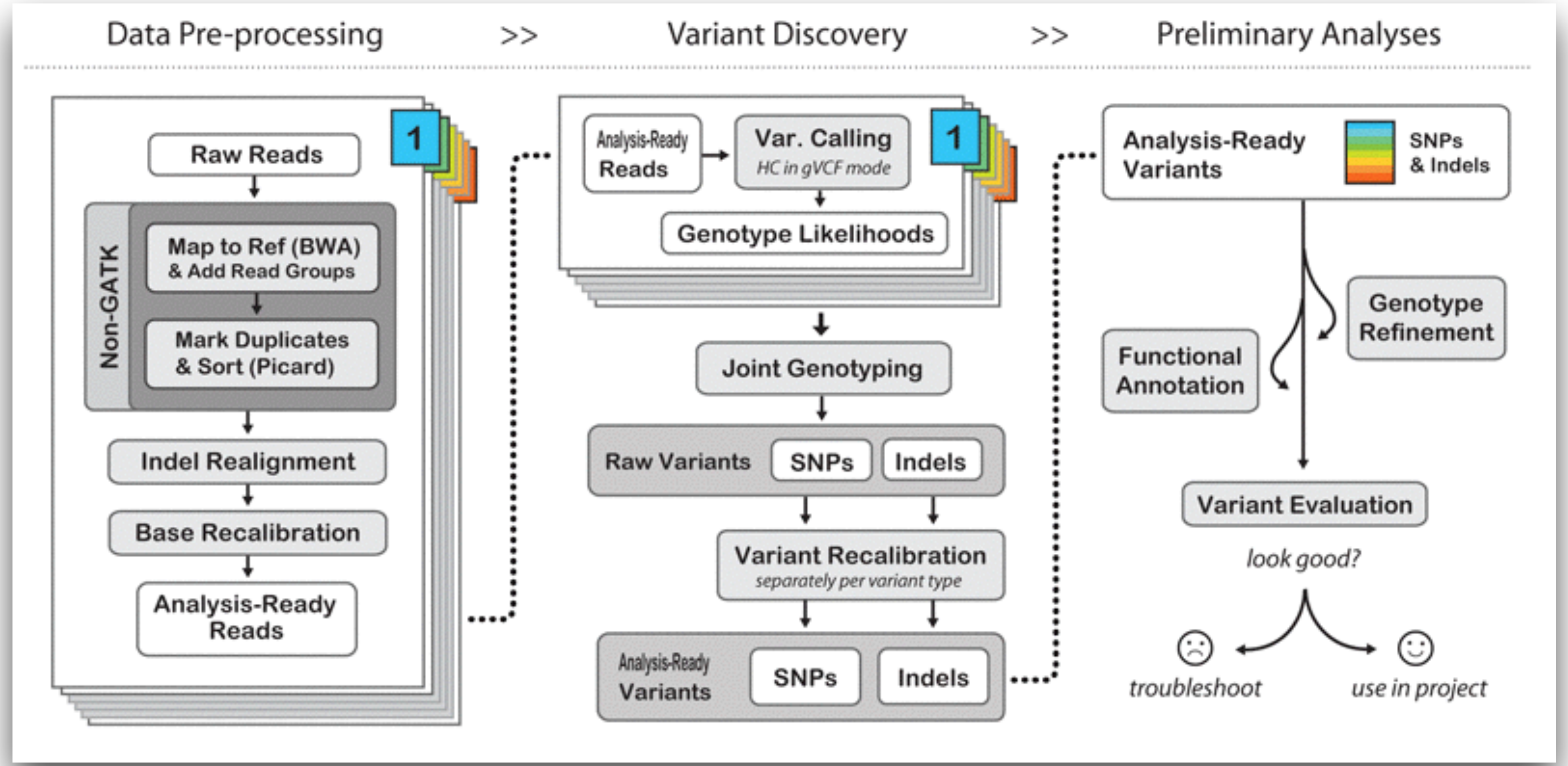
Burrows-Wheeler Alignment

Roland Krause

Objectives

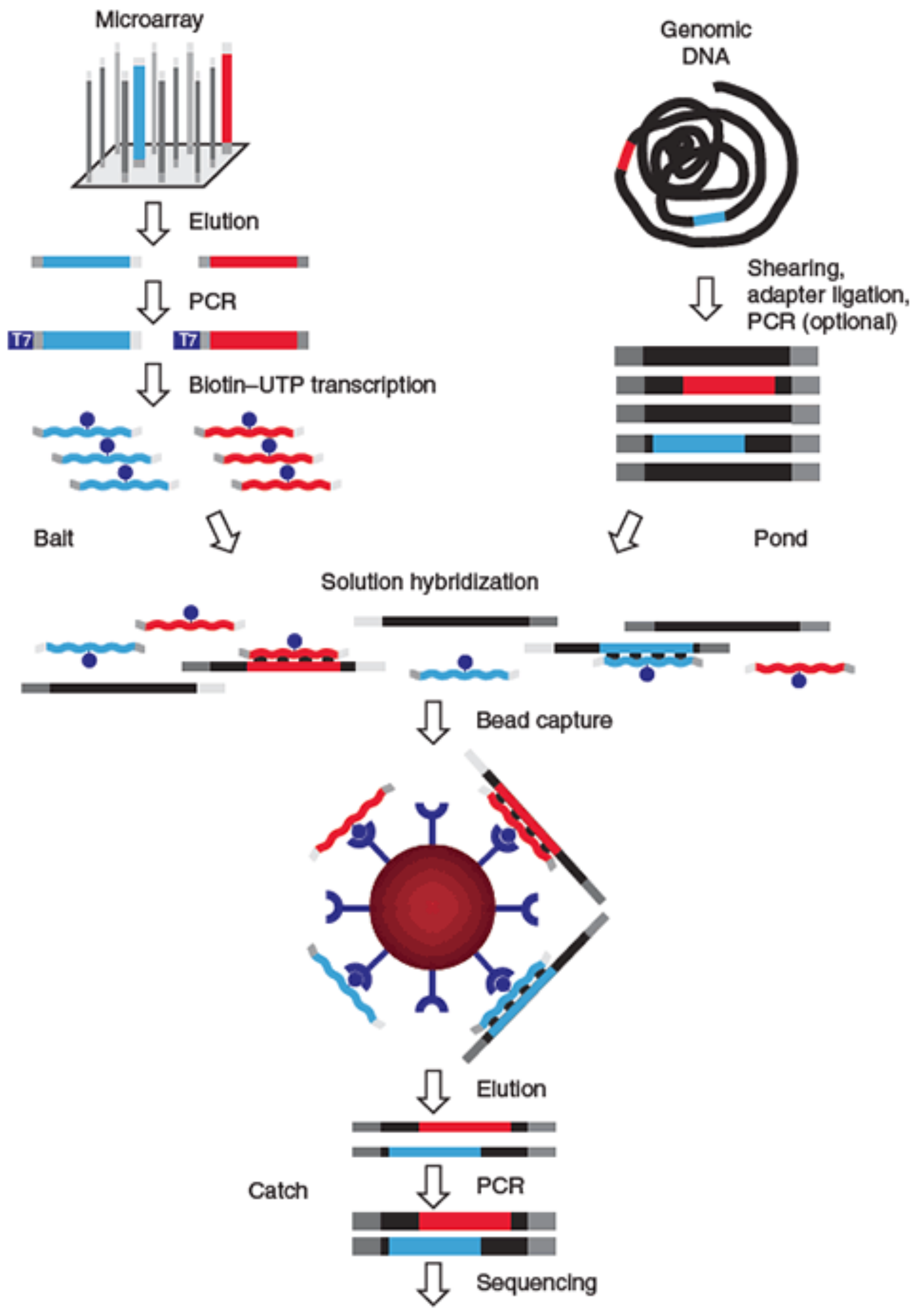
- Repeat basic algorithms for searching biosequences
- Learn about Burrows-Wheeler-Alignment (BWA)
- Transform simple BWA
- Implement a simple BWT using Perl

GATK pipeline



https://www.broadinstitute.org/gatk/guide/best-practices

Exome sequencing



The human reference genome

- Is typically stored in FASTA format
- Build GRCh37: 3Gbases, corresponding to a 3Gbyte file
- Several versions exist
    - Tutorial taken from 1000 Genomes project + Modification of chrMT to be compatible with ENSEMBL

The FASTQ format (Illumina)

```
1.  @FCC189PACXX:2:1314:19975:86201/2
2.  CTCTCTTTCTCTCTTTCTCTCTTTCTTTTCTTTTTTTCTT ...
3.  +
4.  bb_eeeeegggfghhhiagfihhhhiiiihiiiiiiafg ...
```

1. Instrument name, flowcell id, coordinates within the tile, first or second read pair
2. The sequence of the read
3. Optional description
4. Quality values in ASCII (33 + Phred scaled Q)
   Phred score $Q = -10 \log_{10} P$ with P being the Base Call Accuracy (1-error rate)
   ASCII codes

Short read alignment

- Before NGS:
  - SW, FASTA, BLAST
  - MEGABLAST, SSAH2, BLAT
- NGS produces short reads in high throughput, therefore new specialized fast aligners were needed

The challenge

- Align (map) millions of short reads – 75 to 200 bp to the human genome
- Sequence quality (reads) is low

Our options

Our options

- Brute-force searches
- Local or global alignment using dynamic programming
- BLAST (index-based)

Searching options

- Brute force matching:
  - Trivial to implement
  - Extremely slow: O(n*l) naive or O(n+l) smart
  - Space efficient: (O(n+l)) 3 billion bytes for 3Gbp genome
- k-mer index
  - Simple to implement
  - Fast O(n) for k-mer, how to deal with multiple mapping?
  - Space inefficient (O(k+1*n) k+1 times

Short read aligner

- ELAND, RMAP, MAQ, ZOOM, SEQMAP, CLOUDBURST, SHRIMP: hashing reads and scan reference, flexible memory footprint, high overhead when scanning few reads

- SOAPV1, PASS, MOM, PROBEMATCH, NOVOALIGN, RESEQ, MOSAIK, BFAST: hash the genome, parallelizable, require large memory to build reference index, speed sensitive to sequence errors

- SOAPV2, BOWTIE, **BWA**: Burrows-Wheeler Transform (BWT), prefix tree with small memory footprint

BWA

- Uses the Burrows-Wheeler transform algorithm
- Fast and moderate memory footprint
- Gapped alignments
- Non-unique reads are placed randomly with a mapping quality=0
- Li, H. and Durbin, R., Fast and accurate short read alignment with Burrows- Wheeler transform. *Bioinformatics* **25** (14), 1754 (2009)

Suffix Array Search

```
G=GATTACA
   Suffixes Sorted Suffixes
 0 GATTACA  6      A
 1 ATTACA   4      ACA
 2 TTACA    1      ATTACA
 3 TACA     5      CA
 4 ACA      0      GATTACA
 5 CA       3      TACA
 6 A        2      TTACA
 SA = 6,4,1,5,0,3,2
```

This and following material from Michael Schatz (CHSL)
schatzlab.cshl.edu/teaching/2013/2013.10.24.SBU.BWT%20Notes.pdf

Binary Search: O(l lg n); can be reduced to O(lg n) by storing LCP array

Space: N integers (offsets) + N bytes (string)

15 billion bytes for 3 Gbp genome

Burrows–Wheeler

- Want compact space O(n) bytes *and* efficient search O(lg n) or O(l)
- Goal: Optimal space index is 1 byte index per byte of text (full text index)
- BWT has these properties, plus other cool properties.
- Named for Michael Burrow and David Wheeler while working at DEC in 1994
- Original algorithm by Wheeler in 1983

Construction

Sort all cyclic rotations of G'=G$ where G is genome and $ is EOF character that is lexicographically less than all other characters in G

```
Example:
G=GATTACA
G’=GATTACA$
BWT=ACTGA$TA
```

```
Rotations: Sorted (also called BWM)

GATTACA$ $GATTACA
ATTACA$G A$GATTAC
TTACA$GA ACA$GATT
TACA$GAT ATTACA$G
ACA$GATT CA$GATTA
CA$GATTA GATTACA$
A$GATTAC TACA$GAT
$GATTACA TTACA$GA

BWT (last column of BWM) -^
```

Last-first property

The magic of the BWT is the LF property: The ith occurrence of character C in the last column *is* the ith occurrence of character C in the first column.

Lets consider a schematic diagram of the BWM of a DNA string

```
$ _ _ _ _ _ _ _ <- By construction, first row starts with $
A _ _ _ _ _ _ _
A _ _ _ _ _ _ _ <- Followed by section for A
A _ _ _ _ _ _ _
...
C _ _ _ _ _ _ _
C _ _ _ _ _ _ _ <- Followed by C C _ _ _ _ _ _ _
...
G _ _ _ _ _ _ _
G _ _ _ _ _ _ _ <- Followed by G
G _ _ _ _ _ _ _
...
T _ _ _ _ _ _ _
T _ _ _ _ _ _ _ <- Followed by T
T _ _ _ _ _ _ _
```

Lets call those three rotations that start with C rotations X, Y, and Z

The first character of each of those rotations is x, y, z (without loss

of generality –– we don't know what those strings are, but we can
label the characters)

```
...
C x X X X X X X
C y Y Y Y Y Y Y
C z Z Z Z Z Z Z
...
```

Rotation

Now since the BWM contains every cyclic rotation, we know those 3 C strings will also be rotated like so, someplace else in the BWM

```
CxXXXXXX      xXXXXXXC
CyYYYYYY  =>  yYYYYYYC
CzZZZZZZ      zZZZZZZC
```

Key insight: Since the rotations are sorted, we know that X < Y < Z and x <= y <= z. As such their relative placement must also be in sorted order in the BWM when C is rotated to the last column.

```
$ _ _ _ _ _ _ _
A _ _ _ _ _ _ _
A X X X X X X C <- Possible location of X (x=A)
A _ _ _ _ _ _ _
...
C x X X X X X X
C y Y Y Y Y Y Y <- Original locations of X, Y, Z
C z Z Z Z Z Z Z
...
G Y Y Y Y Y Y Y <- Possible location of Y (must be below X, y=G)
G _ _ _ _ _ _ _
G _ _ _ _ _ _ _
...
T _ _ _ _ _ _ _
T _ _ _ _ _ _ _
T Z Z Z Z Z Z C <- Possible location of Z (must be below Y, z=T)
```

Last–First property is actually a statement of the *rest* of the rotation.

When they are sorted as the second character of the rotation, they are also

sorted when they are the first character of the rotation so the ranks must

be the same.

Reconstruction

- How can we use the LF-property to reconstruct G from BWT(G)?
- Say the BWT is ACTTGA$TTAA (11 characters)
- This means the genome must looks like

```
 _ _ _ _ _ _ _ _ _ _ $
 1 2 3 4 5 6 7 8 9 0 1
```

- Since the BWT is a permutation of G, we actually know a lot about how the BWM must look: 1x$, 4xA, 1xC, 1xG, 4xT

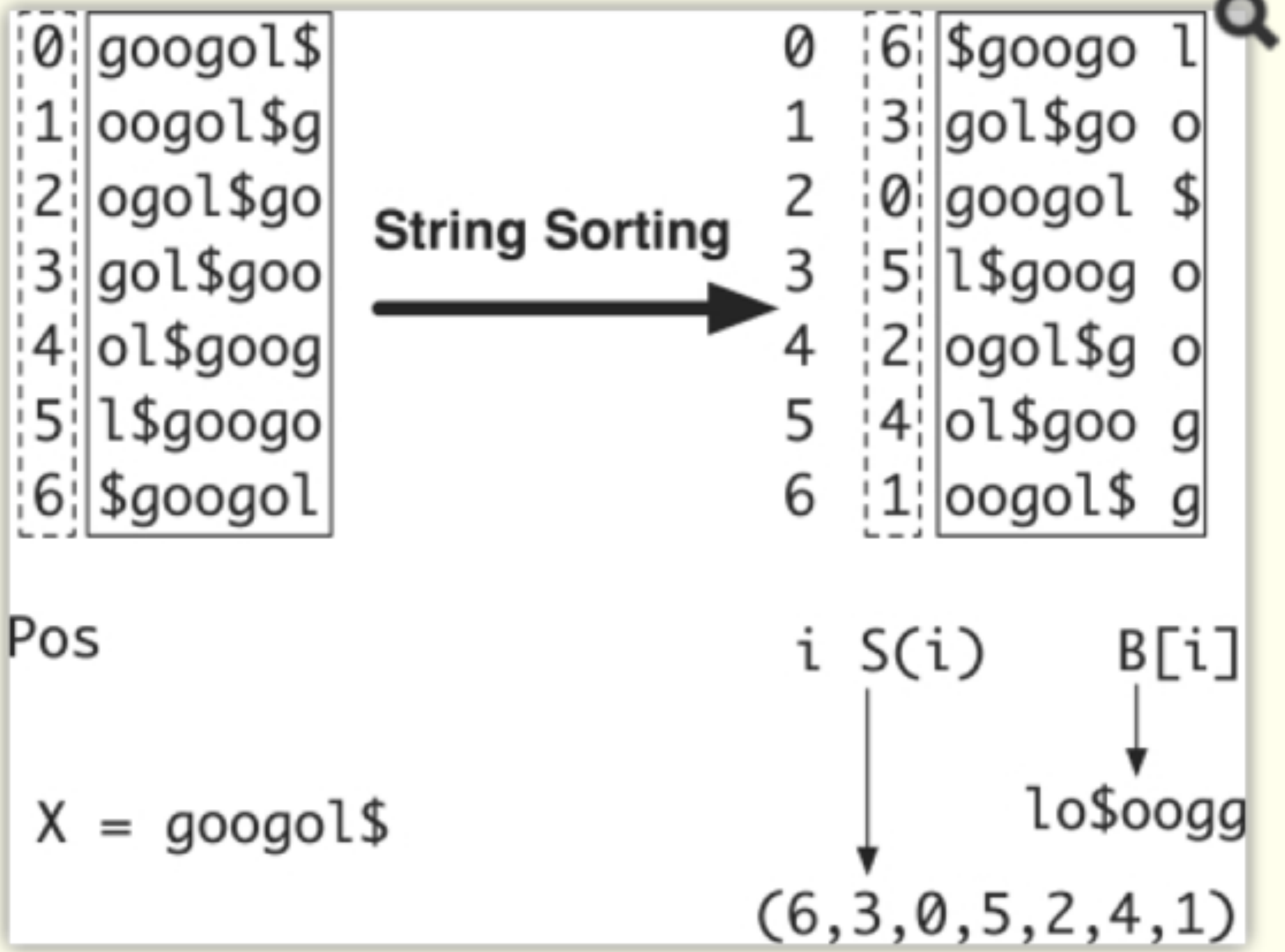
Reconstruction

```
BWT = ACTTGA$TTAA

1 2 3 4 5 6 7 8 9 0 1
$ _ _ _ _ _ _ _ _ _ A <- By construction, $ is first
A _ _ _ _ _ _ _ _ _ C <- Must have 4 A rows
A _ _ _ _ _ _ _ _ _ T "
A _ _ _ _ _ _ _ _ _ T "
A _ _ _ _ _ _ _ _ _ G "
C _ _ _ _ _ _ _ _ _ A <- 1 C row
G _ _ _ _ _ _ _ _ _ $ <- 1 G row
T _ _ _ _ _ _ _ _ _ T <- 4 T rows
T _ _ _ _ _ _ _ _ _ T "
T _ _ _ _ _ _ _ _ _ A "
T _ _ _ _ _ _ _ _ _ A "

^- Last column defined by the BWT
```

Cont. on white board

- `See Schatz notes`

Fig. 2.



Constructing suffix array and BWT string for *X*=`googol$`. String *X* is circulated to generate seven strings, which are then lexicographically sorted. After sorting, the positions of the first symbols form the suffix array (6, 3, 0, 5, 2, 4, 1) and the concatenation of the last symbols of the circulated strings gives the BWT string `lo$oogg`.

Implementation

- Write a function that returns the Burrows-Wheeler transformed string.
- sub bwt($string){

```
…
$bwt_string;
}
```