Roadmap

• Application: short read mapping
• Hash tables & seeded alignment (FASTA)
• Suffix trees
• Suffix arrays
• Burrows-Wheeler Transform
• Wavelet tree
Sequencing Applications

- **DNA-Seq**
  
  - Goal: identify genomic variations (e.g., SNP, SNV)

- **RNA-Seq, ChIP-Seq, BS-Seq, Hi-C**
  
  - Goal: compute levels, detect peaks

- **De novo sequencing/assembly**

Mapping Short Reads

- Mapping reads to a reference genome is computationally challenging
  
  - Billions of reads
  - Genome size in the billion bp
  - Sequencing errors and genomic variations
Short Read Alignment

- Where in genome did each read originate? i.e., given a reference genome and a set of reads, report at least one “good” local alignment for each read if one exists
- What is “good”? For instance
  - Fewer mismatches is better
  - Failing to align a low-quality base is better than failing to align a high-quality base

Spliced-read mapping (RNA-seq)

- Need to map reads that span introns
- Tools: TopHat, HISAT (note: StringTie and Trinity are assemblers)
String matching

• Short read mapping is a version of a well-known problem: (approximate) string matching
• More than two dozen linear-time algorithms have been designed for exact string matching (KMP, BM, DFA-based, FFT-based, ...)
• Many for approximate strings matching
• ... but none of them can be used directly for this application

Approximate/exact string matching

• Fact: Given two strings $w_1$ and $w_2$ at Hamming distance $d$ from each other, they both contain an exact occurrence of a substring of length at least $\lceil m/(d+1) \rceil$. [Baeza-Yates and Perleberg, or BYP]
• Example:
  $w_1 = \text{GATTACA}$
  $w_2 = \text{GGTTACA}$
  TT and CA are occurring exactly. In fact $\lceil m/(d+1) \rceil = \lceil 7/3 \rceil = 2$
Indexing

- Genomes and reads are too large for direct approaches
- Indexing is required

```
Suffix tree

<table>
<thead>
<tr>
<th>6</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>A$</td>
</tr>
<tr>
<td>3</td>
<td>ANA$</td>
</tr>
<tr>
<td>1</td>
<td>ANANA$</td>
</tr>
<tr>
<td>0</td>
<td>BANANA$</td>
</tr>
<tr>
<td>4</td>
<td>NAS</td>
</tr>
<tr>
<td>2</td>
<td>NANA$</td>
</tr>
</tbody>
</table>

Seed hash tables

Many variants, incl. spaced seeds
```

- Choice of index is key to performance
Hashing \( k \)-mers

- \( k \)-mers are substrings of length \( k \) (also called seeds or q-grams)
- A direct-address table of all \( k \)-mers over a DNA alphabet requires \( 4^k \) entries (some could be unused)
- Each entry in the table is a pointer to the list of occurrences of that \( k \)-mer
- Hash the genome or the queries (short reads)?

Hashing \( k \)-mers

- Create a hash table of size \( 4^k \)

<table>
<thead>
<tr>
<th>0001</th>
<th>0002</th>
<th>0003</th>
<th>...</th>
<th>( 4^k )</th>
</tr>
</thead>
</table>

We choose \( k = 6 \)
Hashing $k$-mers

- Hash table of size $4^k$

\[ \begin{array}{c}
00123 @000000 \\
\vdots \\
00123 @000000 \\
\end{array} \]

\[ \begin{array}{c}
\text{AACTGTACCAGTGAG...} \\
\text{AACTGT=001323}_4=123_{10} \\
\end{array} \]

Hashing $k$-mers

- Hash table of size $4^k$

\[ \begin{array}{c}
00123 @000000 \\
\vdots \\
00492 @000001 \\
\vdots \\
\end{array} \]

\[ \begin{array}{c}
\text{AACTGTACCAGTGAG} \\
\text{ACTGTA=013230}_4=492_{10} \\
\end{array} \]
Hashing $k$-mers

• Hash table of size $4^k$

Hashing $k$-mers

• Hash table of size $4^k$ (each entry points to a list of locations)
Considerations

- The table can be very large (depending on $k$)
- The number of entries should be as uniform as possible (avoid sparseness, which would waste memory)
- Entries in the tables tend to be highly non-uniform due to repeats
- One can use a smaller table, use a hashing functions (e.g., $h(x) = x \mod p$, $p$ prime), but then collisions need to be resolved
- To save space, we could store a sample of $k$-mers, but which ones? See the concept of minimizers introduced by [Roberts et al., Bioinformatics 2004]

Alignments can be “seeded” using k-mers
FASTA: seeded alignment

• Heavily used for searching databases (until advent of BLAST)
• Proposed by Lipman and Pearson (1988)
• Inputs
  • $k$ ($k$-mer) size ($k=4$-6 DNA, $k=2$ for proteins)
  • similarity matrix (scoring scheme)
• Compares query sequence pairwise with each sequence in the database
How to choose the value of $k$

- $k$ represents the tradeoff between search speed and sensitivity.
- Small $k$: the search step selects shorter regions but many of them will be discarded in the alignment step, hence an increase in computing time (slower, more sensitive).
- Large $k$: significant alignment regions without any perfect $k$ contiguous matches will be missed in the search step, hence a decrease in the sensitivity of the homology search (faster, less sensitive).

Spaced seeds

- The idea of using a single optimized spaced seed was first introduced in PatternHunter [Ma et al., 2002].
- It was empirically demonstrated that an optimal spaced seed quadruples the search speed, without sacrificing sensitivity (probability of having at least one hit in a highly similar region between query and genome).
- Several results followed to explain the advantage theoretically:

```
GAGTACTCAACACCAACATTAGTGCGAATGGAAAAAT...
11*1**1**1*1111
GAAACTGACAGCAACATCAATGGCGACAGAAAAT...
1**1**1**1**1111
```
Spaced seeds

Performance of weight 11 spaced seeds versus weight 11 and 10 consecutive seeds (from Ma et al, Bioinformatics 2002). For example, in a region of length 64 with 70% identity, Blast’s consecutive weight 11 model has a 0.30 probability of having at least one hit in the range, while a nonconsecutive model of the same weight has a 0.466 probability of getting a hit.
**Suffix trie**

- **Fact:** $y$ occurs in $x$ if $y$ is a prefix of a suffix of $x$
- Build a trie with all the suffixes of the text $x$
- Example: $x = \text{GATTACA}$ we build the tree on

```
GATTACA
ATTACA
TTACA
TACA
ACA
CA
A
```

**Suffix trie for “GATTACA”**

The suffix trie collects in the internal nodes *all* the substrings of $x$
**Suffix trie for “GATTACA$”**

The suffix trie collects in the internal nodes all the substrings of x$.

**Suffix trie**

- Construction $O(n^2)$
- Space $O(n^2)$
- Query $O(m)$
- We can do better by removing unary nodes from the tree, and coalescing the edges
- The resulting tree is called suffix tree
The suffix tree collects in the *implicit* internal nodes *all* the substrings of $x$$1$ $2$ $3$ $4$ $5$ $6$ $7$ $8$

The locus of a string is the node in the tree corresponding to it.

The label in the leaves identifies the suffix position (used to find positions all occs)

The number of leaves in the subtree corresponds to the number of occurrences.

**Space analysis**

- Every node is branching
- The number of leaves is $n$
- Therefore, the overall number of nodes is at most $2n-1$
- Use two integers (constant space) to identify labels on the arcs
- Therefore, the overall size of the tree is $O(n)$
Brute-force construction: Mississippi$
Brute-force construction: Mississippi$
Brute-force construction: Mississippi$
Brute-force construction: \textit{Mississippi}$
Brute-force construction: Mississipp$i$
Brute-force construction: Mississippi
Brute-force Algorithm

- We insert one suffix at a time
- Each suffix insertion takes $O(n)$ time (worst-case)
- Worst-case time complexity is $O(n^2)$
- Average time complexity is $O(n \log n)$

Suffix Links

- **Fact:** If some node in a suffix tree has string $a\alpha$, for $a$ a character and $\alpha$ a string, then there exists some node in the suffix tree has string $\alpha$
Example: Mississippi$
Example: Mississippi$
Adding Suffix Links

• Adding each suffix link is a least common ancestors (LCA) computation
  • This takes $O(n)$ preprocessing + $O(1)$ per LCA
  • Adding all suffix links takes $O(n)$ time
• The first use of suffix links was to speed up suffix tree construction
  • If you keep suffix links on the partially constructed tree, you don’t have to start every insertion from the root ➔ yields a linear time construction

Building suffix trees

• Assume constant size finite alphabet
• Suffix trees can be built in $O(n)$ time and space [Weiner 1973, McCreight 1976, Ukkonen 1995, Farach 1997]
• Observe that several subtrees of the suffix tree are isomorphic, one can merge them to save space (DAWG)
Space-time trade-offs

• How to implement a node? Some choices
  • Array of pointers to children (fast, memory inefficient)
  • List of pointers to children (slow, memory efficient)
  • Global hash map (average, memory ok)
• Each has pros/cons: saving memory usually affects the time to search
  (alphabet dependent)
• The best suffix tree implementations require on average \( \approx 10n \) bytes, \( 20n \) bytes in the worst case [Kurtz 1999]
• Suffix trees are not “cache friendly”

Example: application of suffix trees

• Find the longest common substring between two strings x and y
• Build the suffix tree \( T \) for x
• Walk down the tree \( T \) following y
• If you hit a dead end, save the current depth, and follow the suffix link from the current node
• When you finish y, return the longest substring found
x = abaaba$

p = bbba

Suffix arrays
Suffix array

• Proposed by Manber and Myers (1990) as a space-efficient alternative to suffix trees

Let $x = \text{abab}$

Sort the suffixes lexicographically:

$\text{ab, abab, b, bab}$

The suffix array gives the indices of the suffixes in sorted order

Definition: $SA[i] = k$ iff the $i$-th suffix in the lexicographic order starts at position $k$ in string $x$.

How do we build a suffix array?

• Easiest solution
  • Build a suffix tree in $O(n)$ time
  • Traverse the tree in DFS, picking edges outgoing from each node in lexicographic order and fill the suffix array

• $O(n)$ time worst-case ... but suffix tree are not space-efficient

• Can we do it directly in $O(n)$ time? Unknown until 2003
How do we search for a pattern?

• **Fact:** if \( y \) occurs in \( x \) then \( y \) is a prefix of a suffix of \( x \)

• **Corollary:** the occurrences of \( y \) are consecutive in the suffix array (as prefixes)

• Perform a binary search on the suffix array

• Takes \( O(m \log n) \) time in the worst case

• We can do it faster, using an extra array (LCP)

**Example of a binary search**

Let \( x = \text{mississippi} \)

<table>
<thead>
<tr>
<th>( L )</th>
<th>( i )</th>
<th>ippi</th>
<th>issippi</th>
<th>ississippi</th>
<th>mississippi</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>( i )</td>
<td>ippi</td>
<td>issippi</td>
<td>ississippi</td>
<td>mississippi</td>
</tr>
<tr>
<td>7</td>
<td>( i )</td>
<td>ippi</td>
<td>issippi</td>
<td>ississippi</td>
<td>mississippi</td>
</tr>
<tr>
<td>4</td>
<td>( i )</td>
<td>ippi</td>
<td>issippi</td>
<td>ississippi</td>
<td>mississippi</td>
</tr>
<tr>
<td>1</td>
<td>( i )</td>
<td>ippi</td>
<td>issippi</td>
<td>ississippi</td>
<td>mississippi</td>
</tr>
<tr>
<td>0</td>
<td>( i )</td>
<td>ippi</td>
<td>issippi</td>
<td>ississippi</td>
<td>mississippi</td>
</tr>
</tbody>
</table>

Let \( y = \text{issa} \)

<table>
<thead>
<tr>
<th>( M )</th>
<th>( pi )</th>
<th>ppi</th>
<th>sippi</th>
<th>sisippi</th>
<th>ssippi</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>( pi )</td>
<td>ppi</td>
<td>sippi</td>
<td>sisippi</td>
<td>ssippi</td>
</tr>
<tr>
<td>8</td>
<td>( pi )</td>
<td>ppi</td>
<td>sippi</td>
<td>sisippi</td>
<td>ssippi</td>
</tr>
<tr>
<td>6</td>
<td>( pi )</td>
<td>ppi</td>
<td>sippi</td>
<td>sisippi</td>
<td>ssippi</td>
</tr>
<tr>
<td>3</td>
<td>( pi )</td>
<td>ppi</td>
<td>sippi</td>
<td>sisippi</td>
<td>ssippi</td>
</tr>
<tr>
<td>5</td>
<td>( pi )</td>
<td>ppi</td>
<td>sippi</td>
<td>sisippi</td>
<td>ssippi</td>
</tr>
<tr>
<td>2</td>
<td>( pi )</td>
<td>ppi</td>
<td>sippi</td>
<td>sisippi</td>
<td>ssippi</td>
</tr>
</tbody>
</table>
Example of a binary search

Let $x = \text{mississippi}$

Let $y = \text{issa}$

Example of a binary search

Let $x = \text{mississippi}$

Let $y = \text{issa}$
Fast search

- Assume that $y$ occurs between $L$ and $R$
- In order to determine whether to search in $[L,M]$ or $[M,R]$ we check $y$ against the suffix at $M$
- This check can be achieved with a constant number of comparisons if $LCP(y,M)$ is available
- We can compute $LCP(y,M)$ from the values of $LCP(y,L)$ and $LCP(y,R)$ which have been already computed [omitted]
Fast search

• Still requires $\log n$ probes
• Elements in the query are compared with $M$ at most $m + \log n$ times
• Total complexity of search $y$ in $x$ is $O(m + \log n)$

Fast search

• The fast search requires that LCP($L$, $R$), LCP($L$, $M$) and LCP($M$, $R$) are available
• These LCPs can be computed from LCP($s[i-1]$, $s[i]$) where $s[i]$ is the $i$-th suffix in the suffix array
• Fact: when $j-i>1$, LCP($s[i]$, $s[j]$) = min{LCP($s[i-1]$, $s[i]$) for $k=i+1, i+2, \ldots, j$}
• The array LCP($s[i-1]$, $s[i]$) can be built in advance in $O(n)$ time
LCP values that may be probed during the search

\[
\text{ISA}[k] = i \iff \text{SA}[i] = k
\]

Example: Counting occurrences of \( k \)-mers

- **Problem**: Given a string \( x \) an integer \( k \), compute the number of occurrences for all distinct \( k \)-mers in \( x \)
Counting occurrences of $k$-mers

### Construction of suffix arrays

- Sorting suffixes can be expensive when suffixes have long common prefixes
- Naïve solution: $O(n^2 \log n)$ time
- However, in practice most suffixes can be ordered by comparing short prefixes
- It makes sense to stop checking prefixes after $h$ comparison: this will produce an “approximate” suffix array $S_{a_h}$
- *Prefix doubling* idea based on Karp, Miller & Rosenberg (1972)
Suffix array construction

- Start from SA_1: the order of some suffixes will be final just by inspecting the first symbol
- Compute SA_2 for the “unresolved” suffixes from SA_1: this will resolve more suffixes
- Compute SA_4 for the “unresolved” suffixes from SA_2: this will resolve more suffixes
- ...
- When sorting SA_h one can use the approximate suffix array SA_{h/2} to speed up the computation
- Results in O(n log n) algorithm in the worst case, requires 8n bytes

**Doubling technique**

<table>
<thead>
<tr>
<th>SA_1</th>
<th>SA_2</th>
<th>SA_4</th>
<th>SA_4</th>
<th>SA_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 15</td>
<td>0 15</td>
<td>0 15</td>
<td>0 15</td>
<td>0 15</td>
</tr>
<tr>
<td>1 0 A</td>
<td>1 14 A</td>
<td>1 14</td>
<td>1 14</td>
<td>1 14</td>
</tr>
<tr>
<td>2 2 A</td>
<td>2 2 A</td>
<td>2 3 A A</td>
<td>2 3 A A</td>
<td>2 13 A</td>
</tr>
<tr>
<td>3 3 A</td>
<td>3 10 A A</td>
<td>3 2 A A A</td>
<td>3 2 A A A</td>
<td>3 10 A A A E</td>
</tr>
<tr>
<td>4 5 A</td>
<td>4 13 A A</td>
<td>4 10 A A A</td>
<td>4 10 A A A</td>
<td>4 2 A A A C A T</td>
</tr>
<tr>
<td>5 8 A</td>
<td>5 5 A C</td>
<td>5 5 A C</td>
<td>5 5 A C</td>
<td>5 5 A C</td>
</tr>
<tr>
<td>6 6 10 A</td>
<td>6 6 10 A</td>
<td>6 0 A A</td>
<td>6 0 A A</td>
<td>6 1 A A A A</td>
</tr>
<tr>
<td>7 7 11 A</td>
<td>7 7 11 A</td>
<td>7 8 A A A</td>
<td>7 8 A A A</td>
<td>7 8 A A A A</td>
</tr>
<tr>
<td>8 8 13 A</td>
<td>8 8 13 A</td>
<td>8 11 A A A</td>
<td>8 11 A A A</td>
<td>8 11 A A A A</td>
</tr>
<tr>
<td>9 9 14 A</td>
<td>9 9 14 A</td>
<td>9 3 A A C</td>
<td>9 3 A A C</td>
<td>9 3 A A C</td>
</tr>
<tr>
<td>10 6 C</td>
<td>10 6 C</td>
<td>10 6 C</td>
<td>10 6 C</td>
<td>10 6 C</td>
</tr>
<tr>
<td>11 7 G</td>
<td>11 7 G</td>
<td>11 7 G</td>
<td>11 7 G</td>
<td>11 7 G</td>
</tr>
<tr>
<td>12 1 T</td>
<td>12 1 T</td>
<td>12 1 T</td>
<td>12 1 T</td>
<td>12 1 T</td>
</tr>
<tr>
<td>13 4 T</td>
<td>13 4 T</td>
<td>13 4 T</td>
<td>13 4 T</td>
<td>13 4 T</td>
</tr>
<tr>
<td>14 9 T</td>
<td>14 9 T</td>
<td>14 9 T</td>
<td>14 9 T</td>
<td>14 9 T</td>
</tr>
<tr>
<td>15 12 T</td>
<td>15 12 T</td>
<td>15 12 T</td>
<td>15 12 T</td>
<td>15 12 T</td>
</tr>
</tbody>
</table>
Linear-time construction

• In 2003, two research groups reported on a recursive algorithm to construct suffix arrays in linear time by divide-and-conquer. Here is one of them:
  1. Divide the suffixes in two classes: suffixes starting at position $i \mod 3 \neq 0$ and the others
  2. Construct the suffix array for the first class (recursive call)
  3. Use the resulting suffix array to generate the suffix array for the second class
  4. Merge the two suffix arrays into one

Linear-time construction

• The resulting recurrence relation associated with the time complexity is $T(n)=T(2n/3)+cn$ which has solution $T(n)\approx 3cn$
• Both Karkkainen & Sanders [ICALP 2003], and Ko & Aluru [CPM 2003] need more space than the $O(n \log n)$ construction
• There are $O(n^2 \log n)$ lightweight construction which require $5n$ or $6n$ bytes
In practice

• Worst-case time complexity does not necessarily provide an accurate measure of performance in practice

• SA-IS algorithm of Nong, Zhang & Chan (DCC 2009) is the most time and space efficient algorithm (less than 100 code of C++)
Indexing

• Genome indices can be big. For human:

• Large indices necessitate compromises

  1. Require big-memory machine
  2. Use secondary storage
  3. Build new index each run
  4. Sub-index and do multiple passes

---

Burrows-Wheeler Transform

• Reversible permutation used originally in lossless data compression

\[
\begin{align*}
\text{Input: } x & \quad \rightarrow \quad \text{Matrix of all circular shifts of } x, \text{ sorted lexicographically} \\
\end{align*}
\]
Burrows-Wheeler Transform

• It turns out that $\text{BWT}(x)$ is a LOT more compressible than $x$. Why? The first column is the most compressible because it is totally sorted, the second a “little less”, ...

• Invertible transform: How can we recover $x$ from $\text{BWT}(x)$? In other words, how do we compute $\text{BWT}^{-1}$? Is $x = \text{BWT}^{-1}(\text{BWT}(x))$?

Burrows-Wheeler Transform

• Fact (“LF Mapping”): the $i$-th occurrence of a character in the Last column is same occurrence as the $i$-th occurrence in First column
Burrows-Wheeler Transform

• To recreate $x$ from $\text{BWT}(x)$, sort $x$ to obtain $F$, then set $i=1$, and repeatedly apply 1 & 2 [until $x$ is done]
  1. append symbol $\text{BWT}[i]$ to front of $x$
  2. $i = \text{LF}(i)$
where $\text{LF}(i)$ maps row $i$ to row whose first character corresponds to $i$'s last per $\text{LF}$ mapping

```
g
Sa cc a c g
ac g S ac c
ac u ac g S
ac g$ ac a a
ca a c g S a
ca $ a c a a$
g $a c a c c
```
Burrows-Wheeler Transform

- To recreate \( x \) from \( \text{BWT}(x) \), sort \( x \) to obtain \( F \), then set \( i=1 \), and repeatedly apply 1 & 2 [until \( x \) is done]
  1. append symbol \( \text{BWT}[i] \) to front of \( x \)
  2. \( i = \text{LF}(i) \)

where \( \text{LF}(i) \) maps row \( i \) to row whose first character corresponds to \( i \)'s last per \( \text{LF} \) mapping
Burrows-Wheeler Transform

• To recreate $x$ from BWT($x$), sort $x$ to obtain $F$, then set $i=1$, and repeatedly apply 1 & 2 [until $x$ is done]
  1. append symbol BWT[$i$] to front of $x$
  2. $i = LF(i)$
where $LF(i)$ maps row $i$ to row whose first character corresponds to $i$'s last per $LF$ mapping

![Diagram of Burrows-Wheeler Transform](image-url)
Equivalence BWT-Suffix Array

- The BWT requires sorting cyclic permutations of a string, not suffixes
- Note that we have appended to the string a special end-of-string character which sorts lexicographically before every other character
- Sorting cyclic permutations is then equivalent to sorting suffixes

FM Index

- The FM-index is a (compressed) index based on the BWT
- Properties
  - \textbf{LF} mapping can be used for exact matching, due to the equivalence with the suffix array
  - \textbf{LF}(i) can be made fast with checkpointing
  - and more (see papers below)

Exact Matching with FM Index

• To match $y$ in $x$ using $\text{BWT}(x)$, repeatedly apply rule
  \[ \text{top} = \text{LF}(\text{top}, c) \text{ and } \text{bot} = \text{LF}(\text{bot}, c) \]
  • where $c$ is the next character in $y$ (right-to-left) and $\text{LF}(i, c)$ is the usual $L$-
    to-$F$ mapping, as if the last character was $c$ instead of $L[i]$
Exact Matching with FM Index

• To match $y$ in $x$ using $\text{BWT}(x)$, repeatedly apply rule:

$$\text{top} = \text{LF}(\text{top}, c) \text{ and } \text{bot} = \text{LF}(\text{bot}, c)$$

• where $c$ is the next character in $y$ (right-to-left) and $\text{LF}(i, c)$ is the usual L-to-F mapping, as if the last character was $c$ instead of $L[i]$. 

![Diagram](image-url)
**Exact Matching with FM Index**

- In progressive rounds, **top** & **bot** delimit the range of rows beginning with progressively longer suffixes of $y$.

**Exact Matching with FM Index**

- As soon as the range becomes empty ($\text{top} = \text{bot}$) the query suffix (and therefore the query) does not occur in the text.
Rows ➞ Reference Position(s)

• Once we determine the rows that contain $y$ as a prefix, how do we determine the position of $y$ in $x$?

Naïve solution. Walk back to the beginning of the text; number of steps = offset of hit

2 steps, so hit offset = 2

Rows ➞ Reference Position(s)

• Time complexity $O(|x|)$ – too slow
### Rows ➔ Reference Position(s)

- Other naïve solution: keep the suffix array in memory. The finding the reference position is a lookup in the array.

  ![Suffix Array Example](image)

- Time complexity is $O(1)$, but suffix array uses too much memory.

### Rows ➔ Reference Position(s)

- Hybrid solution. Store sample of suffix array; “walk left” to next sampled (“mark”) row to the left.

  ![Suffix Array Example](image)

- For instance: Bowtie samples every 32 rows by default (configurable).
Putting it all together

- Conclusion $aac$ occurs at position 2 in $acaacg$

Checkpointing in FM Index

- $LF(i,c)$ is the rank of $c$ at row $i$
- Naïve way: count occurrences of $c$ in all previous rows
- Time complexity is linear in length of text – too slow
Checkpointing in FM Index

• Solution: precompute cumulative counts for A/C/G/T up to periodic checkpoints in BWT

• LF(i, c) takes now constant-time, but space can be a concern, in particular for large alphabets

Alternative solution for large alphabets: Wavelet tree

• Example on $x$="Peter Piper picked a peck of pickled peppers"

    Peter_Piper_picked_a_peck_of_pickled_peppers:
    0010100110101101011010100010000110110101101101110

• Alphabet \{\$,P,_,a,c,d,e,f,i,k,l,o,p,r,s,t\} maps to

    \{0,0,0,0,0,0,0,1,1,1,1,1,1,1,1\}
Alternative solution for large alphabets: Wavelet tree

- Example on $x$="Peter Piper picked a peck of pickled peppers"

- Left subtree alphabet {$\$,P,_,a,c,d,e,f}$ maps to $\{0,0,0,1,1,1\}$
- Right subtree alphabet {$i,k,l,o,p,r,s,t$} maps to $\{0,0,0,1,1,1\}$

- Now we get rid of the strings (only keep the bit-vectors)
Alternative solution for large alphabets: Wavelet tree

• Example on $x$=“Peter Piper picked a peck of pickled peppers”

• This is the final wavelet tree: how do we use it to compute $LF(i, c)$ which is the rank of character $c$ in the string $x$ at position $i$?

How to use the wavelet tree to find rank

Suppose we want to compute $LF(5, e)$, we know that $e$ is encoded as 0 at this level, so we take the binary rank query of 0 at position 5 ...

... which is 4, which we use to indicate where to rank in the 0-child
How to use the wavelet tree to find rank

- We then repeat this recursively, at a leaf node we have our answer
- A rank query can be done with \( \log |A| \) binary rank queries on the bit vectors where \( A \) is the size of the alphabet – each of which can be done in \( O(1) \) if you store them in special data structures (e.g., RRR - Raman, Raman, Rao SODA 2002 or another binary rank index)

The FM Index is Small

- Entire FM-index on DNA reference consists of:
  - BWT (same size as \( x \))
  - Checkpoints (~15% size of \( x \))
  - SA sample (~50% size of \( x \))
- Total: ~1.65x the size of \( x \)
- Practical: Bowtie and BWA use the FM index
References on seeds/suffix trees/arrays

• Ma, Tromp, Li “PatternHunter: faster and more sensitive homology search”, Bioinformatics (2002)
• Kurtz, “Reducing the space requirement of suffix trees”, Software-Practice and Experience (1999)

References on the BWT