High-throughput read mapping with Burrows-Wheeler indexes

Veli Mäkinen
Department of Computer Science
University of Helsinki
Read mapping

- **Input**: Short reads extracted from donor DNA.
- **Output**: Alignment of the reads to their locations in reference genome.
- Some errors (but not many) need to be allowed in the mapping.
Solution: backtracking with suffix tree

...ACACATTATCACAGGCATCGGCATTAGCGATCGAGTGCAGTCG.....
Suffix tree

- Suffix tree is a compressed keyword trie of all *suffixes* of a sequence
- E.g. suffixes of sequence CATACT are CATACT, ATACT, TACT, ACT, CT, T.
  - suffix tree looks like:
Suffix tree
Suffix tree

C A T A C T

1 2 3 4 5 6
Back to backtracking

ACA, 1 mismatch

Same idea can be used to many other forms of approximate search, like Smith-Waterman, position-specific scoring matrices, regular expression search, etc.
Properties of suffix tree

- Suffix tree has $n$ leaves and at most $n-1$ internal nodes, where $n$ is the total length of all sequences indexed.
- Each node requires constant number of integers (pointers to first child, sibling, parent, text range of incoming edge, statistics counters, etc.).
- Can be constructed in linear time.
Properties of suffix tree... in practice

- Huge overhead due to pointer structure:
  - Standard implementation of suffix tree for human genome requires over 200 GB memory!
  - A careful implementation (using $\log n$-bit fields for each value and array layout for the tree) still requires over 40 GB.
  - Human genome itself takes less than 1 GB using 2-bits per bp.
Reducing space: suffix array
Suffix array

- Many algorithms on suffix tree can be simulated using *suffix array*...
  - ... and couple of additional arrays...
  - ... forming so-called *enhanced suffix array*...
  - ... leading to the similar space requirement as careful implementation of suffix tree
- Not a satisfactory solution to the space issue.
What we learn today?

- We learn that *backtracking* can be done using *compressed suffix arrays* requiring only 2.1 GB for the human genome.
Burrows-Wheeler transform (BWT)

- Compute a matrix $M$ whose rows are cyclic shifts of sequence $S=s_1s_2...s_n$: $s_1s_2...s_n$, $s_2s_3...s_ns_1$, $s_3s_4...s_ns_1s_2$, ..., $s_{n-1}s_n...s_{n-3}s_{n-2}$, $s_{n-1}s_n...s_{n-2}s_{n-1}$.
- Sort the rows in the lexicographic order in $M$.
- Let $L$ be the last column and $F$ the first column of $M$.
- $bwt(T)=(L,i)$, where $i$ is the row number in $M$ containing $s_1s_2...s_n=S$. 
BWT vs. suffix array

- The lexicographic order of the cyclic shifts of $S$ is essentially \textit{suffix array} $sa(S)$.

- Exercise: Given $L$ and the row number, how to compute $S$ and $sa(S)$?
$S^{-1} = \# \text{TCATAAC}$

stable sort

$$\begin{array}{c}
\text{sa}(S) \\
1: 7 \\
2: 4 \\
3: 2 \\
4: 1 \\
5: 5 \\
6: 6 \\
7: 3 \\
\end{array}$$

$$\begin{array}{c}
i \\
1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \\
LF[i] \\
6 \ 7 \ 4 \ 1 \ 2 \ 5 \ 3 \\
\end{array}$$
LF-mapping

- Let the $i$-th row of $M$ contain cyclic shift $fX^i$, and $j$-th row cyclic shift $lfX$.
- $LF[i] = j$.
- Hence, $L[i] L[LF[i]] L[LF[LF[i]]] ...$ gives the original text in reverse order, where $L[1,n]$ is the transformed text.

**Exercise:** Why the previous sorting algorithm to compute LF-mapping works correctly?
Let $C[c]$ be the amount of symbols smaller than $c$ in $T$, $c \in \{1, 2, \ldots, \sigma\}$.

**Lemma 1**: $LF[i] \in [ C[L[i]]+1, C[L[i]+1] ]$

Let $rank_c(L, i)$ be the amount of symbols $c$ in the prefix $L[1, i]$.

**Lemma 2**: $LF[i] = C[L[i]] + rank_{L[i]}(L, i)$.

**Lemma 3**: When $L$ is stable sorted into $L'$, then $L[i]$ is mapped to $L'[LF[i]]$. 
Proving Lemmas 2 and 3

- Let $X^c < Y^c$. Then $X < Y$ and so is $cX < cY$.
- Let $M[i] = Y^c$, $c = L[i]$.
- Let $J = \{ j \mid M[j] = X^c \}$.
- $LF[i] = C[c] + |\{ j \mid j \in J, j \leq i \}|$
  
  
  $= C[c] + \text{Rank}_c(L, i)$.
- It is easy to see that sorting $\{(L[i], i)\}_i$ gives the same mapping.
$S^{-1} = \#TCATAC$

**stable sort**

\[ \text{sa}(S) \]

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>LF[i]</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

LF(6) = $C[C] + \text{rank}_C(L,6) = 3 + 2 = 5$
One solution is to first build suffix tree using e.g. McCreight‘s or Ukkonen‘s suffix tree construction algorithm and then read suffix array from its leaves. This takes time \( O(n \log \sigma) \).

There are also new direct constructions for suffix arrays that take linear time, when \( \sigma = O(n) \).

BW-transform \( L \) is then given by \( L[i] = S[sa[i] - 1] \), where \( S[0] = S[n] \).
Rank function

- **Lemma 4.** Given a bitvector $B[1,n]$, there is a data structure occupying $o(n)$ bits that supports $\text{rank}_1(B,i)$ and $\text{rank}_0(B,i) = i - \text{rank}_1(B,i)$ in constant time.

- **Lemma 5.** Sequence $L[1,n]$ can be replaced by a data structure (called *wavelet tree*) occupying $n \log \sigma (1+o(1))$ bits and supporting $\text{rank}_c(L,i)$ for all $c \in \Sigma$ in $O(\log \sigma)$ time.

- *Proofs.* See end of these lecture slides.
Compressed suffix array

- Suffix array $sa(S)$ occupies $|S| \log |S|$ bits.
- Next we will develop a *compressed suffix array* $csa(S)$, which occupies $2|S| + \sigma \log |S| + |S| \log \sigma (1+o(1))$ bits, and simulates $SA[i]$ computation in $O(\log \sigma \log |S|)$ time.

- Idea:
  - Store only every $\log n$:th suffix array value.
  - Use LF-mapping locally to find the nearest sampled value.
Compressed suffix array

log n = 3

<table>
<thead>
<tr>
<th>sa</th>
<th>F</th>
<th>M</th>
<th>L</th>
<th>sampling</th>
<th>B</th>
<th>sa’</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:7</td>
<td>#CATACT</td>
<td>1:7</td>
<td>#CATACT</td>
<td>1:1 → 1:7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:4</td>
<td>ACT#CAT</td>
<td>2:4</td>
<td>ACT#CAT</td>
<td>2:1 → 2:4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:2</td>
<td>ATACT#C</td>
<td>3:2</td>
<td>ATACT#C</td>
<td>3:0 → 3:1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:1</td>
<td>CATACT#</td>
<td>4:1</td>
<td>CATACT#</td>
<td>4:1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:5</td>
<td>CT#CATA</td>
<td>5:5</td>
<td>CT#CATA</td>
<td>5:0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:6</td>
<td>T#CATAC</td>
<td>6:6</td>
<td>T#CATAC</td>
<td>6:0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:3</td>
<td>TACT#CA</td>
<td>7:3</td>
<td>TACT#CA</td>
<td>7:0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compressed suffix array

- For LF-mapping table $C[1,\sigma]$ and wavelet tree of BW-transform are enough:
  - $LF[i]=C[L[i]]+\text{rank}_{L[i]}(L,i)$
  - Space $\sigma \log |S|+|S| \log \sigma(1+o(1))$ bits.
  - $LF[i]$ computation takes time $O(\log \sigma)$.

- In addition, the bitvector $B$ takes $|S|+o(|S|)$ bits, as it needs to support constant time $\text{rank}$.

- $sa'$ takes $(|S|/\log |S|)\log |S|=|S|$ bits.

- Computation of one $SA[i]$ value requires at most $\log |S|$ LF-mappings, so the overall time is $O(\log |S| \log \sigma)$. 
Backward search

Search for "ala"

Step 1: search for "a"

Step 2: search for "la"

Step 3: search for "ala"
Backward search with LF-mapping

<table>
<thead>
<tr>
<th>i</th>
<th>SA[i]</th>
<th>L</th>
<th>suffix $T_{SA[i],n}$</th>
<th>i</th>
<th>SA[i]</th>
<th>L</th>
<th>suffix $T_{SA[i],n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>a</td>
<td>$alabar_a_la_alabarda$</td>
<td>1</td>
<td>21</td>
<td>a</td>
<td>$alabar_a_la_alabarda$</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>r</td>
<td>_a_la_alabarda$alabar</td>
<td>2</td>
<td>7</td>
<td>r</td>
<td>_a_la_alabarda$alabar</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>a</td>
<td>_alabarda$alabar_a_la</td>
<td>3</td>
<td>12</td>
<td>a</td>
<td>_alabarda$alabar_a_la</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>a</td>
<td>_la_alabarda$alabar_a</td>
<td>4</td>
<td>9</td>
<td>a</td>
<td>_la_alabarda$alabar_a</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>d</td>
<td>a$alabar_a_la_alabard</td>
<td>5</td>
<td>20</td>
<td>d</td>
<td>a$alabar_a_la_alabard</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>l</td>
<td>a_alabarda$alabar_a_l</td>
<td>6</td>
<td>11</td>
<td>l</td>
<td>a_alabarda$alabar_a_l</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>_</td>
<td>a_la_alabarda$alabar</td>
<td>7</td>
<td>8</td>
<td>_</td>
<td>a_la_alabarda$alabar</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>l</td>
<td>abar_a_la_alabarda$al</td>
<td>8</td>
<td>3</td>
<td>l</td>
<td>abar_a_la_alabarda$al</td>
</tr>
<tr>
<td>9</td>
<td>15</td>
<td>l</td>
<td>a$alabar_a_la_alabard</td>
<td>9</td>
<td>15</td>
<td>l</td>
<td>a$alabar_a_la_alabard</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>$</td>
<td>alabar_a_la_alabarda$</td>
<td>10</td>
<td>1</td>
<td>$</td>
<td>alabar_a_la_alabarda$</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>_</td>
<td>alabarda$alabar_a_la</td>
<td>11</td>
<td>13</td>
<td>_</td>
<td>alabarda$alabar_a_la</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>b</td>
<td>ar_a_la_alabarda$alab</td>
<td>12</td>
<td>5</td>
<td>b</td>
<td>ar_a_la_alabarda$alab</td>
</tr>
<tr>
<td>13</td>
<td>17</td>
<td>b</td>
<td>arda$alabar_a_la_alab</td>
<td>13</td>
<td>17</td>
<td>b</td>
<td>arda$alabar_a_la_alab</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>a</td>
<td>bar_a_la_alabarda$ala</td>
<td>14</td>
<td>4</td>
<td>a</td>
<td>bar_a_la_alabarda$ala</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>a</td>
<td>barda$alabar_a_la_ala</td>
<td>15</td>
<td>16</td>
<td>a</td>
<td>barda$alabar_a_la_ala</td>
</tr>
<tr>
<td>16</td>
<td>19</td>
<td>r</td>
<td>da$alabar_a_la_alab</td>
<td>16</td>
<td>19</td>
<td>r</td>
<td>da$alabar_a_la_alab</td>
</tr>
<tr>
<td>17</td>
<td>10</td>
<td>_</td>
<td>la_alabarda$alabar_a_</td>
<td>17</td>
<td>10</td>
<td>_</td>
<td>la_alabarda$alabar_a_</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>a</td>
<td>labar_a_la_alabarda$al</td>
<td>18</td>
<td>2</td>
<td>a</td>
<td>labar_a_la_alabarda$al</td>
</tr>
<tr>
<td>19</td>
<td>14</td>
<td>a</td>
<td>labarda$alabar_a_la_</td>
<td>19</td>
<td>14</td>
<td>a</td>
<td>labarda$alabar_a_la_</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
<td>a</td>
<td>r_a_la_alabarda$alaba</td>
<td>20</td>
<td>6</td>
<td>a</td>
<td>r_a_la_alabarda$alaba</td>
</tr>
<tr>
<td>21</td>
<td>18</td>
<td>a</td>
<td>rda$alabar_a_la_alaba</td>
<td>21</td>
<td>18</td>
<td>a</td>
<td>rda$alabar_a_la_alaba</td>
</tr>
</tbody>
</table>

\(i' = \text{LF}[6] = C[l'] + \text{Rank}_{i'}(L, 6) = C[l'] + \text{Rank}_{i'}(L, i-1) + 1 = 16 + 0 + 1 = 17\)

\(j' = \text{LF}[9] = C[l'] + \text{Rank}_{i'}(L, 9) = C[l'] + \text{Rank}_{i'}(L, j) = 16 + 3 = 19\)
Backward search with rank-queries

- **Observation**: If $[i,j]$ is the range in BW-matrix $M$, where all rows start with $X$, then range $[i',j']$, where all rows start with $cX$ can be computed using:

  $$i' := C[c] + \text{Rank}_c(L,i-1) + 1,$$
  $$j' := C[c] + \text{Rank}_c(L,j).$$
Backward search with rank-queries

\[ X = \text{A} \]

\[ i, j \]

\[ i', j' \]

\[ i' = C[T] + 1 + 1 = 5 + 2 = 7 \]

\[ j' = C[T] + 2 = 5 + 2 = 7 \]
Backward search - pseudocode

**Algorithm Count**\((P[1,m], L[1,n], C[1, \sigma])\)

1. \(c = P[m]; k = m;\)
2. \(i = C[c]+1; j = C[c+1];\)
3. while \((i \leq j \text{ ja } k > 1)\) do begin
4. \(c = P[k-1]; k = k-1;\)
5. \(i = C[c]+\text{Rank}_c(L, i-1)+1;\)
6. \(j = C[c]+\text{Rank}_c(L, j);\) end;
7. if \((j < i)\) then return 0 else return \((j - i + 1)\);
Algorithm Count makes $O(m)$ queries to function $\text{Rank}_c(L,i)$. Depending on the underlying structure to support $\text{Rank}_c(L,i)$, different time/space tradeoffs can be obtained.
Compressed suffix array as a self-index

Let us define a self-index $\text{csa}(S)$ as a structure that replaces a sequence $S$ with a compressed representation that supports:

- **Count(P)**: Compute the number of occurrences of a given pattern $P$ in $S$.
- **Range(P)**: Return the suffix array range $[i,j]$ containing the suffixes prefixed by the pattern.
- **Locate(i)**: Return value $SA[i]$.
- **Display(k,l)**: Return substring $S[k,l]$.
Compressed suffix array as a self-index

- Combining the earlier compressed suffix array with backward search supports directly `Count()`, `Range()` ja `Locate()` operations.

- `Display()` can be supported by sampling inverse suffix array values and using again LF-mapping.
### Display

\[ \log n = 3 \]

<table>
<thead>
<tr>
<th>sa</th>
<th>F</th>
<th>M</th>
<th>L</th>
<th>sampling</th>
<th>B</th>
<th>sa’</th>
<th>sa⁻¹’</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:7</td>
<td>#CATACT</td>
<td>1:7</td>
<td>#CATACT</td>
<td>1:1</td>
<td>1:7</td>
<td>1:4</td>
<td></td>
</tr>
<tr>
<td>2:4</td>
<td>ACT#CAT</td>
<td>2:4</td>
<td>ACT#CAT</td>
<td>2:1</td>
<td>2:4</td>
<td>2:2</td>
<td></td>
</tr>
<tr>
<td>3:2</td>
<td>ATACT#C</td>
<td>3:2</td>
<td>ATACT#C</td>
<td>3:0</td>
<td>3:1</td>
<td>3:1</td>
<td></td>
</tr>
<tr>
<td>4:1</td>
<td>CATACT#</td>
<td>4:1</td>
<td>CATACT#</td>
<td>4:1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:5</td>
<td>CT#CATA</td>
<td>5:5</td>
<td>CT#CATA</td>
<td>5:0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:6</td>
<td>T#CATA</td>
<td>6:6</td>
<td>T#CATAC</td>
<td>6:0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:3</td>
<td>TACT#CA</td>
<td>7:3</td>
<td>TACT#CA</td>
<td>7:0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ S[3,5] = TAC \]
Compressed suffix array self-index

- Overall space is $n \log \sigma (1+o(1))$ bits, when:
  - Each $(\log n)^{1+\epsilon}/ \log \sigma$:th value is sampled, making $sa'$ and $sa^{-1}'$ tables occupy $o(n \log \sigma)$ bits, with $\epsilon>0$.
  - Bitvector $B$ can be compressed into $o(n \log \sigma)$ bits (we omit the details here).
  - Locate($i$) takes time $O((\log n)^{1+\epsilon})$.
  - Display($i,j$) takes time $O((\log n)^{1+\epsilon}+(j-i)\log \sigma)$.
  - Count() / Range() take time $O(m \log \sigma)$.
High-throughput mapping in practice

- Several tools exist for sequence mapping, e.g. Maq, BWT-SW, BWA, SOAP2, and Bowtie.
- Most are based on backtracking on BWT.
- Let us consider the $k$-mismatches problem for simplicity.
- Recall the backward search algorithm.
Backward backtracking, one step

P=AGC, k=1

\[ \text{2 exact occurrences} \]

\[ \text{sp} \]

\[ \text{ep} \]

\[ \text{rank}_A(L, sp-1) \]

\[ \text{rank}_A(L, ep) \]
Backward backtracking, one step

P=AGC, k=1

2 occurrences with 1 mismatch

rank_{C}(L,sp-1)
rank_{C}(L,ep)
Backward backtracking, one step

$P = AGC, k = 1$

$\text{rank}_G(L, sp-1)$

$\text{rank}_G(L, ep)$

2 occurrences with 1 mismatch
Backward backtracking – pseudocode

Algorithm kmismatches(P,L, k, j, sp, ep)
(1) if (j = 0) then
(2) Report occurrences Pos[sp], . . . , Pos[ep]; return;
(3) for each s ∈ Σ do
(4) sp′ ← C[s] + rank_s(L, sp − 1)+1;
(5) ep′ ← C[s] + rank_s(L, ep);
(6) if (P[j] != s) k’ ← k − 1; else k’ ← k;
(7) if (k’ ≥ 0) kmismatches(P,L, k’, j − 1, sp’, ep’);

First call: kmismatches(P,L,k,m,1,n)
Example test run

- Compressed suffix array for human genome occupied 2.1 GB.
- 10000 patterns of length 32 searched for with $k=0,1,2$ mismatches.
- Average search times (finding the ranges) were 0.3, 8.2, and 121 milliseconds per pattern, for $k=0,1,2$, respectively.
- Locating one occurrence took 0.9 milliseconds on average.
Search space pruning: BWA

- Build compressed suffix array (aka FM-index) for S and its reverse $S^r$. Call these *forward FM-index* and *reverse FM-index*.

- Compute a table $\kappa[1,m]$ such that $\kappa[i]$ gives a lower bound for the minimum amount of errors needed to match $P[1,i]$ in $T$.
  - E.g. $\kappa[i] = \text{minimum number of times } P[1,i] \text{ need to be split such that each piece occurs exactly in } T$.
  - Initialize $\kappa[0]=0$. With reverse FM-index, backward search $Pr$ from $i=m$ to $i=1$ setting $\kappa[m-i]=\kappa[m-i-1]$ until empty interval, say at $Pr[i']$, then set $\kappa[m-i']=\kappa[m-i'-1]+1$ and continue in the same way.

- Consider search space state corresponding to suffix $P[j,m]$, interval $[sp,ep]$, and $k'$ mismatches. If $k'+\kappa[j-1]>k$, no need to continue search down from current state.
Search space pruning: Bowtie

- Also uses *forward FM-index* and *reverse FM-index*.
- Splits possible occurrences in different categories and searches each category separately. For example, with 1-mismatches search, an occurrence can have the error (i) in $P[1,m/2]$ or (ii) in $P[m/2+1,m]$. For (i) category occurrences use reverse FM-index for the search; no branching in first $m/2$ characters. For (ii) category occurrences use forward FM-index for the search; no branching in first $m/2$ characters.
- For more errors, there is always a bad category, e.g. in 2-mismatches search, pattern can be split to 3 pieces and categories are all different ways to distribute 2 erros in 3 pieces. Distribution 101 is a bad one, as one has to start the search allowing branching.
Search space pruning: SOAP2

- Also uses *forward FM-index* and *reverse FM-index*.
- Solves the bad category (e.g. 101) case of bowtie: Is able to search $P[1,2m/3]$ using forward FM-index and continue directly the search from reverse FM-index with $P[2m/3+1,m]$ (see course book for details).
Appendix

PROVING RANK LEMMAS
**Constant time rank using o(n) extra bits**

- **<O(n) space, O(log n) time>:** Store answers at each \( \log n \):th position.
  - Read the rest from the original bitvector.

\[
\begin{array}{c}
\text{rank}=6+1=7 \\
\downarrow \\
00011010110110010011010110101010110010101100101010\ldots \\
0 \quad 1 \quad 3 \quad 6 \quad 8 \quad \ldots
\end{array}
\]
Constant time rank using $o(n)$ extra bits

- **<o(n) space, O(log n) time>:** Store answers at each $\log^2 n$:th position. Store relative answers at each $\log n$:th position.
- Read the rest from the original bitvector.

```
rank=3+3 +1=7

1

00011010110110010011010110101010110010101100101010...
0       3       8   ...
0   1   0   3   0
```
Constant time rank using $o(n)$ extra bits

- $<o(n) \text{ space}, O(1) \text{ time}>$: Same as before, but read the last answer from a precomputed table of size $o(n)$.

\[
\begin{align*}
\text{rank} &= 3 + 3 + 1 = 7 \\
00011010110110010011010110101010110010101100101010... \\
0 &\quad 3 &\quad 8 &\quad \ldots \\
0 &\quad 1 &\quad 3 &\quad 0 &\quad \ldots
\end{align*}
\]
### Precomputed table

- We would need to answer \textit{rank} in a block of length $\log n$ bits in constant time.
- Let us divide the block into two $(\log n)/2$ bits parts.
- There are $2^{(\log n)/2} = \sqrt{n}$ bitvectors of length $(\log n)/2$.
- We can store in $\sqrt{n} \log n \log \log n$ bits the answers to all possible rank-queries for all bitvectors of length $(\log n)/2$. 
Example of rank-computation

\[
B = 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 0
\]

<table>
<thead>
<tr>
<th>superblockrank</th>
<th>blockrank</th>
<th>smallrank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>01</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>11</td>
</tr>
</tbody>
</table>

\[
\text{rank}_1(B, 11) = \text{superblockrank}[0] + \text{blockrank}[2] + \\
\text{smallrank}[01, 1] + \text{smallrank}[11, 0]
\]

\[
= 0 + 4 + 1 + 1 = 6
\]
Wavelet tree

- **Rank/Select** for sequences.
- Recall LF-mapping of BW-transform:
  \[ LF[i] = C[L[i]] + \text{Rank}_{L[i]}(L,i) \]
- Wavelet tree of represents \( L = \text{bwt}(S) \) in \( n \log \sigma (1+o(1)) \) bits, such that each \( \text{Rank}_c(L,i) \) query takes \( O(\log \sigma) \) time, where \( n = |S| \).
Wavelet tree, example 1

$\Sigma = \{\#, A, C, G, T\}$

$L = T \ T \ C \ # \ A \ C \ A$

$L[6] = C$

$B = 1100010$, $B[6] = 1 \rightarrow \text{right}$
$i = \text{rank}_i(110010,6) = 4$

$B = 1100$, $B[4] = 0 \rightarrow \text{left}$
$i = \text{rank}_0(1100,4) = 2$
Wavelet tree, example 2

$\Sigma = \{\#, A, C, G, T\}$

$L = T T C \# A C A$

$1 1 1 0 0 1 0$

$A \in \{\#, A\} \rightarrow \text{left}$

$i = \text{rank}_0(1110010, 6) = 2$

$A \in \{A\} \rightarrow \text{right}$

$i = \text{rank}_1(011, 2) = 1$

$\text{Rank}_A(L, 6) = 1$
The tree has $\log \sigma$ levels, each consuming constant time for $\text{rank}_{0/1}$-queries:
- $O(\log \sigma)$ time for $\text{Rank}_c(L,i)$.

Each level has at most $n$ bits. After preprocessing each level for $\text{rank}_{0/1}$-queries, the whole tree occupies $n \log \sigma (1+o(1))$ bits.
Other alternatives

- Indicator-bitvector for each symbol:
  - $O(1)$ time in $\sigma n(1+o(1))$ bits.

- Instead of balanced tree, use Huffman tree:
  - $O(\log n)$ time in $n(H_0+1)(1+o(1))$ bits.

- Compress the bitvectors still supporting rank-queries:
  - $O(\log \sigma)$ time in $nH_0(1+o(1))$ bits, or
    $nH_k(1+o(1))$ bits if the input is BW-transform.
  - (Details omitted here)