Indexing Structures for Sequences

CS234

Roadmap

- Application: short read mapping
- Hash tables & seeded alignment (FASTA)
- Suffix trees
- Suffix arrays
- Burrows-Wheeler Transform
- Wavelet tree
- Bowtie and other tools for short read mapping
Sequencing instruments

- High-throughput sequencing produces DNA sequences ("reads") in huge volumes at low cost
- Costs are decreasing exponentially fast

Strong demand

- Personal Genomics, Cancer, Metagenomics, Epigenetics, etc.

- Future growth will be multiplicative
  - Spatial: tissues, individuals, geography
  - Temporal: circadian, seasonal, lifetime
  - Disease: cancer
Metagenomics

• The application of modern genomics techniques (e.g., sequencing) to the study of communities of microbial organisms directly from environmental samples
• Applications: medicine (e.g., human microbiome initiative), biofuels, environmental remediation, agriculture, biotechnology, ecology (air, water, dirt)

Metagenomics

• The human body is composed by ~37 trillion cells
• However, additional ~100 trillion microorganisms (viruses, bacteria, fungi, archaea, and single-celled eukaryotes which comprise of 1-3% total body mass) resides on the surface and in deep layers of skin, in the mouth, on the eyes, in the lungs, in the gastrointestinal tracts (stomach and intestine), and other
Metagenomics

• The human genome includes ~20-30K genes
• The collective genomes of all the bacteria, fungi and viruses in one person’s microbiome are thought to include ~8M genes

Questions
– Is everyone’s microbiome the same?
– Does the microbiome change over time?
– What is the relationship between the microbiome, health, and disease?

Sequencing Applications

• Resequencing
  Goal: identify genomic variations (e.g. SNPs)

• RNA-seq, ChIP-seq, Methyl-seq, HiC
  Goal: classify, measure significant peaks

• De novo sequencing/assembly
Mapping Short Reads

Mapping the reads to a reference genome can be computationally challenging
- Hundreds of million of reads, genomes in the size of billion base pairs
- 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
• Examples: TopHat

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, …)
• Not as many for approximate strings matching
• But they cannot 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 $\left\lfloor \frac{m}{d+1} \right\rfloor$. [Baeza-Yates and Perleberg, or BYP]

- **Example**:
  
  $w_1 = \text{GATTCA}$
  
  $w_2 = \text{GGTACA}$
  
  **TT** and **CA** are occurring exactly. In fact $\left\lfloor \frac{m}{d+1} \right\rfloor = \left\lfloor \frac{7}{3} \right\rfloor = 2$

Indexing

- Genomes and reads are too large for direct approaches

- **Indexing** is required

  ![Suffix tree and suffix array/BWT](image)

- Choice of index is key to performance
Hashing

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 $4^k$

```
AACTGT
...
```

---

Hashing $k$-mers

- Create a hash table of size $4^k$

```
AACTGT
AACTGT
AACTGT
AACTGTaccagtgag
...
```
Hashing $k$-mers

• Create a hash table of size $4^k$
Hashing $k$-mers

- Create a hash table of size $4^k$
Considerations

- The table can be very large
- The number of entries should be as uniform as possible (avoid sparseness)
- Entries in direct-address tables tend to be non-uniform, but if one uses hashing functions (e.g., $h(x) = x \mod p$, $p$ prime) collisions need to be resolved

Seeded Alignment

![Seeded Alignment Diagram]

- Human: `CACACTGCACTGATGACTCTYCTCTGGAATACCC---GATA`
- Zebrafish: `CACAAGTACAGATGACTCTYCTCTGGAATACCC---GATA`
- Figu: `-ACAGATGCACTGGAATACCC---TGATC`
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

![Diagram](image)
Choosing 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

```
GAGTACTCAACACCAACATTAGTGCAATGGAAAAT...
||||| |||| | ||| |||
GAATACTCAACAGCAACATCAATGGCAGCAGAAAAT...
|**|***|**|****|****|
```
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 Trees
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
SUFFIX TREE FOR “GATTACA$”  

The suffix tree collects in the implicit internal nodes all the substrings of $x$

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 pos 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: Mississippi$
Brute-force construction: Mississippis
Brute-force construction: Mississippi$
Brute-force Algorithm

- We insert one suffix at a time
- Each suffix insertion takes $O(n)$ time
- Total time is $O(n^2)$

- Average complexity is $O(n \log n)$

Suffix tree for “abaababaabaababaababa$”

Inside explicit nodes we have annotated the count of the number of occurrences
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$
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 $\Rightarrow$ 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”
Applications of suffix trees

- Find the longest common substring between two strings $x$ and $y$

1. Build the suffix tree $T$ for $x$
2. Walk down the tree $T$ following $y$
3. If you hit a dead end, save the current depth, and follow the suffix link from the current node
4. When you exhaust $y$, return the longest substring found

$x = \text{abaaba}$
$p = \text{bbaa}$
Suffix Arrays

Suffix array

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

Let \( x = abab \)

Sort the suffixes lexicographically:

\( ab, abab, b, bab \)

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

\[
SA[i] = \begin{cases} 
3 & i = 1 \\
1 & i = 2 \\
4 & i = 3 \\
2 & i = 4 
\end{cases}
\]

Definition: \( SA[i] = k \) if 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
- Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array
- \(O(n)\) time
- Waste of space: 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 binary search

Let $x = \text{mississippi}$

Let $y = \text{issa}$

Example of binary search

Let $x = \text{mississippi}$

Let $y = \text{issa}$
Example of binary search

Let $x = \text{mississippi}$

$\begin{array}{cccccccc}
L & \rightarrow & i & \text{ippi} \\
M & \rightarrow & \text{issippi} & \text{ississippi} \\
R & \rightarrow & \text{mississippi} & \text{pi} \\
\end{array}$

Let $y = \text{issa}$

$\begin{array}{cccccccc}
L & \rightarrow & \text{ippi} \\
M & \rightarrow & \text{ississippi} \\
R & \rightarrow & \text{mississippi} \\
\end{array}$

Rightmost: IF $\text{mid} \leq \text{query}$ THEN left = mid + 1 ELSE right = mid
Leftmost: IF $\text{mid} < \text{query}$ THEN left = mid + 1 ELSE right = mid
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 \( \text{LCP}(y,M) \) is available
• We can compute \( \text{LCP}(y,M) \) from the values of \( \text{LCP}(y,L) \) and \( \text{LCP}(y,R) \) which have been already computed [omitted]

Fast search

• We still require \( \log n \) 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 $k = i+1, i+2, ..., j\}$
- 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.
Counting occurrences of $k$-mers

- **Problem:** Given a string $x$ an integer $k$, compute the number of occurrences for all distinct $k$-mers in $x$
Construction of suffix arrays

- Sorting suffixes can be expensive when suffixes have long common prefixes
- Naïve solution: $O(n^2 \log n)$
- 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 $S_{ah}$
- 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

Linear time construction

- In 2003, two groups reported a recursive algorithm to construct suffix arrays in linear time by divide-and-conquer. Here is the high-level idea:
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 \( T(n) \approx 3cn \)
- Karkkainen and Sanders [ICALP 2003], and Ko and Aluru [CPM 2003]: both need more space than the \( O(n \log n) \) construction
- There are now \( O(n^2 \log n) \) lightweight construction which require 5\( n \) or 6\( n \) 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++)
Burrows-Wheeler Transform

Indexing

• Genome indices can be big. For human:

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

\[
\begin{align*}
\text{Input: } x & \quad \text{BWT}(x) \\
a c a a c g & \rightarrow a c g s a c \rightarrow a c g s a c \\
a c g s a c & \rightarrow a c a c g & \rightarrow a c a c g \\
g s a c a a & \rightarrow g s a c a a & \rightarrow g s a c a a
\end{align*}
\]

Matrix of all circular shifts of \(x\), sorted lexicographically

- Once BWT(\(x\)) is built, all else shown here is discarded (matrix shown for illustration only)


Burrows-Wheeler Transform

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

- Invertible transform: How can we obtain \(x\) from BWT(\(x\))? In other words, how do we compute BWT^{-1}?
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

- 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 LF mapping
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

![Diagram of Burrows-Wheeler Transform](image-url)
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

![Diagram](image1.png)

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

![Diagram](image2.png)
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

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

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


Exact Matching with FM Index

- To match y in x using BWT(x), repeatedly apply rule top = LF(top, c); bot = LF(bot, c)
  - where c is the next character in y (right-to-left) and 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 $\text{top} = \text{LF}(\text{top}, c); \text{bot} = \text{LF}(\text{bot}, c)$
  - 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 $L[i]$. 

```
  a a c
  $a c a a e g$
  $a a c g a c$
  $a c a a c g$
  $a c g a c a$
  $c a c g a a$
  $c g a c a a$
  $g a c a a c$

  a a c
  $a c a a c g$
  $a c a c g a c$
  $a c a a c g$
  $a c g a c a$
  $c a c g a a$
  $c g a c a a$
  $g a c a a c$
```
Exact Matching with FM Index

• To match \( y \) in \( x \) using \( \text{BWT}(x) \), repeatedly apply rule \( \text{top} = \text{LF}(\text{top}, c); \text{bot} = \text{LF}(\text{bot}, c) \)
  – 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 \( L[i] \)

• In progressive rounds, \( \text{top} \) & \( \text{bot} \) delimit the range of rows beginning with progressively longer suffixes of \( y \)
Exact Matching with FM Index

- As soon as the range becomes empty ($top = bot$) the query suffix (and therefore the query) does not occur in the text

Rows to Reference Positions

- Once we know a row contains a legal alignment, how do we determine its position in the reference?
Rows to Reference Positions

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

  2 steps, so hit offset = 2

• Linear in length of $x$ in general – too slow

Rows to Reference Positions

• Naïve solution. Keep the suffix array in memory: finding reference position is a lookup in the array

  hit offset = 2

• Suffix array is too big
Rows to Reference Positions

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

```
$ a c a a c g$
$ a c g s a c$
$ a c a c g$
$ a c g s a c$
$ a c g s a c$
$ a c g s a c$
$ a c g s a c$
```

```
1 step
$ a a c$
$ c a a c$
```

For instance: Bowtie samples every 32 rows by default (configurable)

Put It All Together

- Conclusion aac occurs at position 2 in acaacg
Checkpointing in FM Index

• LF$(i, c)$ determines the rank of $qc$ in row $i$

• Naïve way: count occurrences of $qc$ in all previous rows, complexity is linear in length of text – too slow

Solution: pre-calculate cumulative counts for A/C/G/T up to periodic checkpoints in BWT

• LF$(i, c)$ takes now constant-time
  (if space between checkpoints is considered constant)
Alternative solution: Wavelet tree

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

• Alphabet \{\$,$P,$,\_,a,c,d,e,f,i,k,l,o,p,r,s,t\} maps to 
\{0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1\}

Alternative solution: Wavelet tree

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

• Left subtree alphabet \{\$,$P,$,\_,a,c,d,e,f\} maps to 
\{0,0,0,0,1,1,1,1,1\}

• Right subtree alphabet \{i,k,l,o,p,r,s,t\} maps to 
\{0,0,0,0,1,1,1,1\}
Alternative solution: Wavelet tree

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

- Now we get rid of the strings (only keep the bit-vectors)

Alternative solution: Wavelet tree

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

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

Suppose we want to know $\text{LF}(5, e)$, 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

• 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 – 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)
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

Indexing Performance

• Bowtie employs builds the BWT using the algorithm by Kärkkäinen that can trade flexibly between memory usage and running time
• For human (NCBI 36.3) on 2.4 GHz AMD Opteron

<table>
<thead>
<tr>
<th>Physical memory Target</th>
<th>Actual peak memory footprint</th>
<th>Wall clock time</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 GB</td>
<td>14.4 GB</td>
<td>4h:36m</td>
</tr>
<tr>
<td>8 GB</td>
<td>5.84 GB</td>
<td>5h:05m</td>
</tr>
<tr>
<td>4 GB</td>
<td>3.39 GB</td>
<td>7h:40m</td>
</tr>
<tr>
<td>2 GB</td>
<td>1.39 GB</td>
<td>21h:30m</td>
</tr>
</tbody>
</table>

Bowtie vs. Maq vs. SOAP

<table>
<thead>
<tr>
<th></th>
<th>CPU time</th>
<th>Wall clock time</th>
<th>Reads per hour</th>
<th>Peak virtual memory footprint</th>
<th>Bowtie speedup</th>
<th>Reads aligned (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bowtie v0.9.6</td>
<td>15m:07s</td>
<td>15m:41s</td>
<td>33.8 M</td>
<td>1,149 MB</td>
<td>-</td>
<td>67.4</td>
</tr>
<tr>
<td>SOAP v0.6.6</td>
<td>91h:57m:35s</td>
<td>91h:47m:46s</td>
<td>0.08 M</td>
<td>13,619 MB</td>
<td>351x</td>
<td>67.3</td>
</tr>
<tr>
<td>Maq v0.6.6</td>
<td>16m:41s</td>
<td>17m:57s</td>
<td>29.5 M</td>
<td>1,353 MB</td>
<td>-</td>
<td>71.9</td>
</tr>
<tr>
<td>Bowtie v0.9.6</td>
<td>17h:46m:35s</td>
<td>17h:53m:07s</td>
<td>0.49 M</td>
<td>804 MB</td>
<td>59.8x</td>
<td>74.7</td>
</tr>
<tr>
<td>SOAP v0.6.6</td>
<td>17m:58s</td>
<td>18m:26s</td>
<td>28.8 M</td>
<td>1,353 MB</td>
<td>-</td>
<td>71.9</td>
</tr>
<tr>
<td>Maq v0.6.6</td>
<td>32h:56m:53s</td>
<td>32h:58m:39s</td>
<td>0.27 M</td>
<td>804 MB</td>
<td>107x</td>
<td>74.7</td>
</tr>
</tbody>
</table>

- PC: 2.4 GHz Intel Core 2, 2 GB RAM
- Server: 2.4 GHz AMD Opteron, 32 GB RAM
- Bowtie v0.9.6, Maq v0.6.6, SOAP v1.10
- SOAP not run on PC due to memory constraints
- Reads: FASTQ 8.84 M reads from 1000 Genomes (Acc: SRR001115)
- Reference: Human (NCBI 36.3, contigs)

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 BWT